Quality-Driven Synthesis and Optimization of Embedded Control Systems

Linköping Studies in Science and Technology

Dissertation No. 1386

Quality-Driven Synthesis and Optimization of Embedded Control Systems

by

Soheil Samii

Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköping 2011

ISBN 978-91-7393-102-1
ISSN 0345-7524

Electronic version:
http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-68641

Cover design by Ali Ardi.

Printed by LiU-Tryck, Linköping 2011.

To my mother

Abstract

THIS thesis addresses several synthesis and optimization issues for embedded control systems. Examples of such systems are automotive and avionics systems in which physical processes are controlled by embedded computers through sensor and actuator interfaces. The execution of multiple control applications, spanning several computation and communication components, leads to a complex temporal behavior that affects control quality. The relationship between system timing and control quality is a key issue to consider across the control design and computer implementation phases in an integrated manner. We present such an integrated framework for scheduling, controller synthesis, and quality optimization for distributed embedded control systems.

At runtime, an embedded control system may need to adapt to environmental changes that affect its workload and computational capacity. Examples of such changes, which inherently increase the design complexity, are mode changes, component failures, and resource usages of the running control applications. For these three cases, we present tradeoffs among control quality, resource usage, and the time complexity of design and runtime algorithms for embedded control systems.

The solutions proposed in this thesis have been validated by extensive experiments. The experimental results demonstrate the efficiency and importance of the presented techniques.

The research presented in this thesis has been funded by CUGS (the National Graduate School in Computer Science in Sweden) and by ELLIIT (Excellence Center at Linköping–Lund in Information Technology).

Acknowledgments

I would like to take the opportunity to thank several people for their important contributions to my work. I am very fortunate to have had two outstanding thesis advisors. Thank you, Petru Eles and Zebo Peng, for making this thesis possible, for providing excellent technical advice, for being encouraging and supportive, and for being great friends. I appreciate all the time you have invested in my education and the opportunities you have given me during my time as your student.

Parts of the material presented in this thesis have been developed in cooperation with Anton Cervin at Department of Automatic Control, Lund University. I thank him for his support and for the excellent feedback he has provided on my work.

I visited University of California at Los Angeles (UCLA) in the spring and summer of 2009. My gratitude goes to Paulo Tabuada at Department of Electrical Engineering and all members of his research group—Cyber-Physical Systems Laboratory—for a very rewarding and insightful visit and collaboration. The ideas presented in Chapter 7 were developed during my time at UCLA.

I thank my colleagues at Department of Computer and Information Science (IDA) for their contribution to an excellent working environment. Special thanks to past and present members of Embedded Systems Laboratory (ESLAB) for insightful discussions, for entertaining lunch and coffee breaks, for their company during many hours of work before paper deadlines, and for fun travels to conferences.

Nahid Shahmehri has been a great mentor and friend during my time at IDA. I thank her for showing interest in my work and I appreciate the valuable advice she has given me. Erik Larsson was my advisor when I joined ESLAB to work on my master's thesis. I thank him for giving me the opportunity to gain early experience in writing research papers and for his continuous support during my years at ESLAB. Traian Pop was very helpful whenever I had technical questions during my first years as

a Ph.D. candidate. I take the opportunity to thank him for all his help.

I thank Unmesh Bordoloi for the fruitful collaboration related to Chapter 6. Many thanks to Sergiu Rafiliu for always having time to help and discuss details on my work. I also thank Amir Aminifar for our discussions and collaboration, and for his efforts in proofreading drafts of this thesis.

I started my Ph.D. candidature around the same time as Shanai Ardi, Jakob Rosén, Sergiu Rafiliu, and Anton Blad. A sincere thanks to them for going through all these years of graduate studies with me.

Last, I am thankful to my family and friends for their support and encouragement over the years. I am particularly grateful for their patience and understanding when I had upcoming paper deadlines and needed to spend more time than usual at the office. My deepest appreciation goes to my brother Saman and to my mother Mansoureh for always being there for me. I dedicate this thesis to my dear mother. She has made it possible for me to succeed.

SOHEIL SAMII
Norrköping ⇄ Linköping
August 2011

Contents

List of Figures

List of Tables

1 Introduction

THE topic of this thesis is integrated design and optimization of computer-based control systems. The main contribution is synthesis and optimization of multiple control applications on embedded computing and communication platforms, considering temporal system properties, multiple operation modes, faults in computation resources, and state-based scheduling. We shall in this chapter introduce and motivate these research topics, as well as give a general introduction to embedded control systems and their research challenges. Last, we shall summarize the contributions of this thesis and outline its organization.

1.1 Motivation

Examples of prominent applications of embedded control can be found in automotive and avionics systems. Usually, such embedded control systems comprise multiple sensors, actuators and computer networks that are heterogeneous in terms of the processing and communication components [NSSW05]. Several control applications execute on such distributed platforms to control physical processes through input–output interfaces to sensors and actuators. The construction of control algorithms

is typically based on models of the physical processes or plants to be controlled, as well as models of disturbances [ÅW97]. The execution frequency of a control application is determined based on the dynamical properties of the controlled plant, characteristics of disturbances, the available computation and communication capacity of the system, and the desired level of control quality.

The development of embedded control systems comprises not only analysis and synthesis of controllers, but also scheduling of the tasks and messages of the control applications on the computation nodes and communication links, respectively, according to the scheduling policies and communication protocols of the platform. Such modern systems exhibit complex temporal behavior that potentially has a negative impact on control performance and stability—if not properly taken into account during control synthesis [WNT95, CHL+03]. In addition, real-time systems theory for scheduling and communication focuses mainly on analysis of worst-case temporal behavior, which is not a characteristic metric for control performance. The general idea to cope with these problems is to integrate temporal properties of the execution of control applications and characteristics of the platform in the synthesis of control algorithms [ÅCES00, CHL+03, ÅC05, WÅÅ02], as well as to consider the control performance in traditional system-level design problems for embedded systems [KMN+00] (e.g., scheduling, communication synthesis, and task mapping [PEP04, PEPP06]).

In the traditional design flow of control systems, the control algorithm and its execution rate are developed to achieve a certain level of control performance [ÅW97]. This is usually done independently of the development and analysis of the underlying execution platform, thus ignoring its inherent impact on control performance. The next step is implementation of the control applications on a certain execution platform. Control performance is typically not considered at this stage, although, as we have discussed, the delay characteristics of the whole system have a strong impact on the control performance and are raised as a result of the implementation [WNT95, CHL+03]. In many cases with multiple control applications that share an execution platform, the implementation phase includes system-level integration, comprising task mapping, scheduling, and communication synthesis. Worst-case delays and deadlines are often the only interface between control design and its computer implementation. Although significant and important characterizations of

real-time systems, worst-case delays occur very seldom and are thus not appropriate indicators of control quality. This leads typically to solutions with several problems: First, the controllers are not optimized for the underlying execution platform. Second, the schedule for task executions and message transmissions is not optimized with regard to control performance. Current research aims to close this gap by integrating control and computer design methods, with numerous research results already developed.

At design time, as much information as possible regarding the application and platform characteristics should be used to develop embedded systems with high quality of service. The design solutions must not only be synthesized efficiently and provide high application performance, but they should also allow the system, at runtime, to adapt to different application requirements, workload changes, and the variation of availability of computation and communication components due to temporary or permanent component failures. Another potential direction is to consider the actual state of the controlled plant at runtime to decide the amount of platform resources to be used to achieve a certain level of performance. Such an adaptive mechanism can lead to better trade-offs between resource usage and control performance, compared to traditional periodic control that merely considers the plant state to compute control signals. Event-based and self-triggered control [Åst07, VFM03, WL09, AT10] are the two classes of such adaptive control mechanisms that have been presented in literature. These two control approaches do not require periodic execution; instead, the execution is triggered based on the state of the controlled process. The development of nonperiodic control approaches is still in an early research phase. In addition, scheduling and resource-management policies of state-triggered control applications are open research problems.

1.2 Summary of Contributions

The contribution of this thesis may be viewed as four components that are treated in Chapters 4–7, respectively. In Chapter 4, we consider multiple control applications on a distributed execution platform, comprising several computation nodes and a communication bus. We consider time-triggered scheduling of tasks and messages according to static sched-

ules [XP90, Ram95, PSA97, WMF05] and TTP (Time-Triggered Protocol) [Kop97] communication. We also consider systems with priority-based scheduling [LL73] and communication according to the CAN (Controller Area Network) or FlexRay protocols [Bos91, Fle05]. The contribution is an optimization framework for the synthesis of control laws, periods, and schedules (or priorities) [SCEP09, SEPC11]. The framework is fundamental to the work presented in Chapters 5 and 6.

Chapter 5 treats control systems with multiple operation modes. At runtime, an embedded control system may switch between alternative functional modes at predetermined time instants or as a response to external events. Each mode is characterized by a certain set of active feedback-control loops. Ideally, when operating in a certain mode, the system uses schedules and controllers that are customized for that mode, meaning that the available computation and communication resources are exploited optimally in terms of control performance for that mode. The main design difficulty is caused by the exponential number of modes to be considered at design time, meaning that it is not practical to synthesize customized solutions for all possible modes. Design solutions for some few modes, however, can be used to operate the system in several modes, although with suboptimal control performance. The contribution of Chapter 5 is a synthesis method that trades control performance with the amount of spent design time and the space needed to store schedules and controllers in the memory of the underlying distributed execution platform [SEPC09].

In Chapter 6, we consider variations in the available computation capacity due to faults in computation nodes. The contribution is a design framework that ensures reliable execution and a certain level of control performance, even if some computation nodes fail [SBEP11]. When a node fails, the configuration—the set of operational computation nodes—of the underlying distributed system changes. The system must adapt to this new situation by activating and executing some new tasks on the operational nodes—tasks that were running on failed computation nodes. The task mapping, schedules, and controllers must be constructed for this new configuration at design time. The design-space complexity, which is due to the exponential number of configurations that may appear as a consequence of node failures, leads to unaffordable design time and large memory requirements to store information related to mappings, schedules, and controllers. We show that it is sufficient to

synthesize solutions for a number of minimal configurations to achieve fault tolerance. Further, we discuss an approach to generate mappings, schedules, and controllers for additional configurations of the platform to improve control quality, relative to the minimum level already provided by the minimal configurations.

The last contribution of this thesis is presented in Chapter 7. We consider self-triggered control as an alternative to periodic control implementations. Self-triggered control has been introduced recently and enables efficient use of computation and communication resources by considering the sampled plant states to determine resource requirements at runtime. To exploit these advantages in the context of multiple control applications running on a uniprocessor platform, we present a runtime-scheduler component [SEP+10, SEPC10] for adaptive resource management. The optimization objective is to find appropriate trade-offs between control performance and resource usage.

All proposed techniques in this thesis have been validated by extensive experiments, which are presented and discussed for each contribution separately in the corresponding chapter.

1.3 List of Publications

Parts of the contents of this thesis are presented in the following publications:

- Soheil Samii, Sergiu Rafiliu, Petru Eles, Zebo Peng. "A Simulation Methodology for Worst-Case Response Time Estimation of Distributed Real-Time Systems," *Design, Automation and Test in Europe Conference,* Münich, Germany, March 2008.

- Soheil Samii, Anton Cervin, Petru Eles, Zebo Peng. "Integrated Scheduling and Synthesis of Control Applications on Distributed Embedded Systems," *Design, Automation and Test in Europe Conference,* Nice, France, April 2009.

- Soheil Samii, Petru Eles, Zebo Peng, Anton Cervin. "Quality-Driven Synthesis of Embedded Multi-Mode Control Systems," *Design Automation Conference,* San Francisco, California, USA, July 2009.

- Soheil Samii, Petru Eles, Zebo Peng, Anton Cervin. "Runtime Trade-Offs Between Control Performance and Resource Usage in Embedded Self-Triggered Control Systems", *Workshop on Adaptive Resource Management,* Stockholm, Sweden, April 2010.
- Soheil Samii, Petru Eles, Zebo Peng, Paulo Tabuada, Anton Cervin. "Dynamic Scheduling and Control-Quality Optimization of Self-Triggered Control Applications," *IEEE Real-Time Systems Symposium,* San Diego, California, USA, December 2010.
- Soheil Samii, Petru Eles, Zebo Peng, Anton Cervin. "Design Optimization and Synthesis of FlexRay Parameters for Embedded Control Applications," *IEEE International Symposium on Electronic Design, Test and Applications,* Queenstown, New Zealand, January 2011.
- Soheil Samii, Unmesh D. Bordoloi, Petru Eles, Zebo Peng. "Control-Quality Optimization of Distributed Embedded Control Systems with Adaptive Fault Tolerance," *Workshop on Adaptive and Reconfigurable Embedded Systems,* Chicago, Illinois, USA, April 2011.
- Amir Aminifar, Soheil Samii, Petru Eles, Zebo Peng. "Control-Quality Driven Task Mapping for Distributed Embedded Control Systems," *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications,* Toyama, Japan, August 2011.
- Soheil Samii, Unmesh D. Bordoloi, Petru Eles, Zebo Peng, Anton Cervin. "Control-Quality Optimization of Distributed Embedded Systems with Adaptive Fault Tolerance." Under submission.
- Soheil Samii, Anton Cervin, Petru Eles, Zebo Peng. "Integrated Scheduling and Synthesis of Distributed Embedded Control Applications." Under submission.

The following publications are not covered by this thesis but are generally related to the field of embedded systems development:

- Soheil Samii, Erik Larsson, Krishnendu Chakrabarty, Zebo Peng. "Cycle-Accurate Test Power Modeling and its Application to SoC Test Scheduling," *International Test Conference,* Santa Barbara, California, USA, October 2006.

- Soheil Samii, Mikko Selkälä, Erik Larsson, Krishnendu Chakrabarty, Zebo Peng. "Cycle-Accurate Test Power Modeling and its Application to SoC Test Architecture Design and Scheduling," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* Vol. 27, No. 5, May 2008.

- Soheil Samii, Yanfei Yin, Zebo Peng, Petru Eles, Yuanping Zhang. "Immune Genetic Algorithms for Optimization of Task Priorities and FlexRay Frame Identifiers," *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications,* Beijing, China, August 2009.

1.4 Thesis Organization

This thesis is organized in nine chapters. In the next chapter, we shall discuss related research results in the area of embedded and real-time control systems. Specifically, we shall focus on methodologies that integrate control theory and computer systems design and optimization. Chapter 2 also highlights the differences and contributions of this thesis related to current state of theory and practice in the development of embedded control systems. In Chapter 3, we shall discuss preliminaries related to controller analysis and synthesis, as well as notation and models related to control applications and their distributed execution platforms. These preliminaries serve as a common foundation for the contributions presented in Chapters 4–7.

Chapter 4 presents an integrated optimization framework for the synthesis of controllers and schedules for tasks and messages in distributed control systems. In Chapter 5, we further develop the material in Chapter 4 towards synthesis of multi-mode control systems. Another direction, for the synthesis of fault-tolerant control systems, is presented in Chapter 6. The last contribution of this thesis is related to scheduling of multiple self-triggered control tasks and is presented in Chapter 7.

The conclusions are presented in Chapter 8, where we relate the material in this thesis with the general research and development of embedded control systems. In Chapter 9, we outline several directions for future research that are closely related to the material presented in this thesis. Appendix A lists selected notation with page references.

2
Background and Related Work

THE purpose of this chapter is to review the research efforts that have been made towards integrated design and optimization of control and computer systems. In this context of computer–control co-design, there are several design parameters of interest. Two of them are related directly to the controller: the control period that determines the rate of sampling, computation, and actuation; and the control law (also referred to as the control algorithm) that determines the mapping from sensor measurements to actuator values. Other synthesis and optimization issues related to the implementation of embedded control applications are, for example, task scheduling on computation nodes and management of messages on the communication bus of the platform.

In the area of integrated control and computer systems design, several previously published research results treat some of the design issues of computer-based control systems in different contexts. In addition, it is becoming increasingly important to construct systems that are adaptive in several regards—for example, related to mode changes, component failures, and states of the physical processes under control. This is due to the varying operational environment of embedded control systems.

This chapter is organized in five sections. We discuss the traditional design flow of control systems, as well as system-level optimization of

embedded computing systems in Section 2.1. In Section 2.2, we present the current state of computer–control co-design research for periodic control systems on uniprocessor and distributed computation platforms. Further, in Section 2.3, we shall review the published research results in the area of adaptive embedded systems for control applications. As a special class of adaptive embedded platforms and methodologies, we shall discuss fault-tolerant computing for control applications in Section 2.4. Last, in Section 2.5, we shall consider two execution models as alternatives to periodic control in resource-constrained systems—namely, event-based and self-triggered control.

2.1 Traditional Design Flow

Embedded control systems have during the recent decades been predominantly designed and implemented as periodic tasks that read sensors, compute control signals, and write to actuators [ÅW97]. Two main activities are performed at design time prior to deployment of the control system: First, controllers and algorithms are synthesized, and second, these are implemented on a given computation platform. In the traditional design flow of embedded control systems, these two activities are treated separately and independently. This leads to a large gap and weak interface between control design and computer implementation, subsequently limiting the control performance that can be achieved.

2.1.1 Controller Synthesis

The first design activity—controller synthesis—is twofold: First, a sampling period is chosen, and second, a control law is synthesized. From a control engineering viewpoint, the control period is decided merely based on the dynamics of the physical process to be controlled as well as the effect of the period on control performance. Rules of thumb, based on plant dynamics, are often used to select the controller period [ÅW97]. The amount of available computation and communication bandwidth may also impact the selection of controller periods. Seto et al. [SLSS96, SLSS01] presented an approach for systematic optimization and trade-off between period selection, control performance, and computation bandwidth on a uniprocessor platform with schedulability constraints.

The synthesis of the control law means that a mapping from the mea-

sured plant outputs to the actual control signal is decided. Although there are implementation factors such as delay and jitter that degrade control performance, these are often neglected, or assumed to be small and constant, during controller synthesis. The control tasks, which implement the synthesized controllers, execute according to the control law at a rate given by the chosen period.

2.1.2 Computer Implementation

The second design activity is the implementation of one or several control applications on a given computation platform, which may consist of a single computation component or of multiple computation nodes connected in a network. For the case of distributed embedded platforms, the second design activity comprises mapping and scheduling of tasks and messages on the computation nodes and communication links, respectively. This activity integrates several control applications on a platform with multiple computation and communication components. Tasks are mapped to computation nodes, and the system is scheduled and configured according to the scheduling policy and communication protocol of the platform.

Distributed execution platforms, which consist of several computation nodes connected to communication buses, are very common in, for example, automotive and avionics systems. The tasks on each node are scheduled either at design time (static cyclic scheduling [Kop97]) or at runtime (e.g., based on priorities [LL73]). Communication of messages on the bus is performed by the communication controllers according to the protocol adopted by the execution platform. Common communication protocols in automotive embedded systems are the time-triggered protocol TTP [Kop97], which is based on a TDMA[1] scheme and static schedules for message transmissions, and the event-triggered protocol CAN [Bos91], which is based on an arbitration process that, for example, can be driven by static message priorities. Especially in the automotive systems domain, the FlexRay protocol [Fle05] is replacing TTP and CAN. Some of the reasons are higher bandwidth and the combination of time-driven and event-driven communication.

The communication configuration of the system has an inherent effect on the system timing and, in turn, the quality of the control appli-

[1] Time division multiple access.

cations. The synthesis of schedule tables for messages in time-triggered communication systems, and selection of priorities or frame identifiers for event-driven communication, are therefore of critical importance during the design of embedded control systems.

Depending on the scheduling policy and communication protocol of the platform, the implementation-related activities to be performed at design time include synthesis of schedule tables, assignment of priorities to tasks and messages, and assignment of frame identifiers to messages. Pop et al. [PEPP06, PEP04] proposed a design optimization framework for such synthesis of distributed embedded systems with strict timing constraints, supporting time-triggered execution and communication with schedule tables and the TTP communication protocol, as well as event-driven scheduling with priorities and CAN-based communication. The application domain is hard real-time systems for which deadlines must be met and, if possible, worst-case delays are to be minimized. Di Natale and Stankovic [DS00] proposed a simulated annealing-based method that, given the application periods, constructs a static schedule to minimize the jitter in distributed embedded systems with precedence and timing constraints. They did not, however, address the impact of the timing jitter on the actual control performance—their work did not focus on control applications in particular but is general for application domains for which jitter is a central component of the performance metric.

Bus-access optimization has been elaborated in literature for various communication protocols in the context of hard and soft real-time systems [EDPP00, GG95, PPEP07, SYP+09]. Some other design frameworks for hard real-time systems have been presented for period optimization [DZD+07]. Design frameworks based on integer linear programming have been used to co-synthesize task mapping and priorities for distributed real-time systems with hard deadlines [ZZDS07].

The design optimization methods we have discussed in this section can potentially be used for control applications. However, control performance is not well reflected by the optimization parameters (periods, worst-case temporal behavior, and hard deadlines).

2.1.3 Problems with the Traditional Design Flow

In the traditional design flow, control laws are synthesized based on the plant dynamics, characteristics of disturbances, and the selected period.

Delays in the control loop due to computation and communication are typically not accounted for. At the very best, delays are assumed to be short and constant. Mapping and scheduling of tasks and messages is performed based on worst-case delays and deadlines. Worst-case temporal behavior and deadlines are not accurate representatives of control performance [WNT95], although these may provide the sufficient constraints regarding stability requirements [KL04]. The execution and communication of the control tasks introduce timing delays that not only are longer than assumed in the control design, but also varying during execution. This variation in each period of the control applications is due to the scheduling and communication in the system. Resource sharing by several applications generally leads to complex temporal behavior and is inherent in modern embedded systems with multiple computation and communication components. The longer and varying timing delays degrade the overall control performance, which, in the worst case, can jeopardize stability of the control loops [WNT95]. This problem is especially relevant for modern embedded systems with the increasing volume of functionality and sharing of computation and communication resources. Thus, the delay distribution is a more significant factor of control performance than worst-case delays and deadlines. The delay distribution characterizes not only the worst-case delay but also the occurrence frequency of delays in general. To optimize control quality, design tools and methodologies must consider more elaborate models of delay than worst-case delays.

The temporal behavior of embedded systems with several computation and communication units has been studied by the real-time systems community for several decades. The main focus has been analysis of worst-case timing for systems with various scheduling policies and communication protocols, starting with the paper by Liu and Layland [LL73] on schedulability conditions for rate-monotonic and earliest-deadline-first scheduling on periodic, uniprocessor systems. Sha et al. [SAÅ+04] surveyed the key results in real-time scheduling theory over three decades of research, since the results of Liu and Layland. Fixed-point iteration is one of the successful techniques to analyze response times in real-time systems [JP86, ABR+93]. Various methods for timing analysis of uniprocessor and distributed real-time systems have been proposed over the years [TC94, PG98, Fid98, CKT03a, CKT+03b, PEP03, RJE03, HHJ+05, PEPP06]. Response-time analysis of real-time com-

munication has also been proposed for systems with CAN [DBBL07] and FlexRay [PPE+08, HBC+07]. Relatively little work has been done to make use of knowledge and analysis of system timing in the design phase of controllers.

In addition to the computation delay of a control application itself, resource sharing and communication contribute to the delay in the control loop. It is already well known that not only the average delay impacts the control performance, but also the delay variance or jitter [WNT95, Tör98, ÅCES00, CHL+03]. The variance, in particular, is difficult to compensate for in the actual control law. When such compensations can be made, the control law is typically time varying [NBW98, MFFR01]. The restriction that the delays must be smaller than the sampling period is also common [NBW98], but is usually not applicable in the context of distributed control with communication in the control loop. In addition, time stamps are needed to implement jitter-compensating controllers [Lin02].

To achieve high performance in distributed embedded control systems, it is important to consider the system timing during controller synthesis, and to consider the control performance during system scheduling. Such control–scheduling co-design problems [ÅCES00, ÅC05, ÅCH05, ACSF10] have become important research directions in recent years. Furthermore, the interaction between control and computation has been emphasized and acknowledged as an important area of research in the context of cyber-physical systems with tight interaction between the computer system and its operational environment [Wol09, Lee09].

2.2 Integrated Control and Computing

The periodic application model with hard deadlines is a convenient interface between controller design and its computer implementation but it also abstracts away the implementation-induced delays and jitters that affect the control performance significantly. Nevertheless, until recently, hard deadlines have been the interface between control design and computer implementation of control applications. As has been pointed out by Wittenmark et al. [WNT95], delay and jitter in sampling, as well as between controller execution and actuation, have a negative impact on control performance and may—in the worst-case—lead to an unstable

closed-loop system. This problem has to be addressed by considering the time-varying delays and their impact on control performance both during controller design and during scheduling of computations and communications. This has opened up a relatively new research area on the border between control and computer engineering.

In their seminal work, Seto et al. [SLSS96, SLSS01] studied uniprocessor systems that run several control tasks. They solved the problem of optimal period assignment to each controller, considering schedulability constraints for rate-monotonic and earliest-deadline-first scheduling [LL73]. The optimization objective is to maximize the performance of the running controllers. Their approach does not consider the delays in the control loop and their impact on the control performance. Recently, Bini and Cervin [BC08] proposed an approximate response-time analysis as a method for the estimation of the control delay in a uniprocessor system with multiple control tasks. Using a control-performance function that is linear in the sampling periods and the estimated, constant control delays, the authors used analytical methods to assign controller periods that give optimal control performance. Bini and Di Natale [BD05] extended the work by Seto et al. [SLSS96, SLSS01] towards priority-based systems with arbitrary fixed priorities. Wu et al. [WBBC10] presented a solution to the period-selection problem for uniprocessor control systems with earliest-deadline-first scheduling. With the motivation of large execution-time variations, Sha et al. [SLCB00] extended the work on period assignment by Seto et al. [SLSS96], leading to better CPU utilization at runtime based on elastic scheduling [CBS00]. Related work on integrated control and uniprocessor scheduling by Palopoli et al. [PPS+02] addresses robustness optimization of multiple control applications on a uniprocessor platform. The optimization results are the periods and the control laws. Another step was taken by Zhang et al. [ZSWM08] for period assignment to multiple control tasks with the objective to optimize robustness and power consumption under rate-monotonic, uniprocessor scheduling.

Other important results in the context of control–scheduling co-design for uniprocessor systems make it possible to divide the control tasks into several subtasks to improve control performance [Cer99]. Furthermore, Cervin et al. [CLE+04] combined stability conditions related to sampling–actuation jitter and response-time analysis for uniprocessor systems for the assignment of periods to multiple control tasks. Cervin and

Eker proposed a complementary scheduling policy—the control server—to eliminate jitter in uniprocessor control systems [CE05]. For static cyclic scheduling on uniprocessor execution platforms, Rehbinder and Sanfridson [RS00] proposed a method that computes an offline schedule and the corresponding control actions for multiple independent control tasks. The result of the optimization is a static cyclic schedule and time-varying controllers. Buttazzo and Cervin [BC07] discuss three approaches to tackle the problem of timing jitter (the difference between worst-case and best-case delay). The first approach is to cancel timing jitter completely by actuating the control input at the end of the task periods. The second method reduces jitter and delay by appropriate assignment of task deadlines in uniprocessor systems with deadline-monotonic or earliest-deadline-first scheduling. The third method is to adopt non-preemptive execution. The second approach is also taken by Balbastre et al. [BRVC04] to reduce control-delay jitter.

For control loops that are closed over computer networks, stability requirements based on communication delays and packet dropouts were provided by Zhang et al. [ZBP01]. Research results on integrated control and computer systems design for distributed control applications have started to appear recently. For example, Ben Gaid et al. [BCH06] considered static scheduling of control-signal transmission on a communication channel for a single control loop with given sampling period. The results of the optimization are a finite sequence of control signals and start times for their transmissions over the communication channel to the actuator. Based on state feedback and a resource-allocation method [MLB+04], Velasco et al. [VMC+06] presented a bandwidth-management and optimization framework for CAN-based networked control systems, without considering delay characteristics and their impact on control performance. Goswami et al. [GSC11a] presented a framework for the design of controllers and communication schedules that satisfy stability conditions. The communication schedules are chosen such that the number of unstable samples is upper-bounded in an arbitrary time interval. Such communication schedules were found experimentally through control–communication co-simulations for FlexRay. Voit et al. [VSG+10] presented an optimization framework for distributed control systems, where multiple performance metrics are considered under the influence of the communication schedule. For each performance metric, a subset of the schedule parameters is of importance during optimization. Further, the

authors derived closed-form expressions of worst-case message delays as functions of the schedule parameters. Such design frameworks are relevant for control applications for which worst-case temporal behavior is a major factor in the control-performance metric.

As we have noticed, a significant amount of research has been already conducted towards integrated control–computer design methods. Especially for the case of distributed control systems, however, several parameters have not been considered by the design and optimization methodologies we have mentioned in this chapter. Most works in this context have focused on co-design of controllers and communication schedules, under the assumption that periods are given. The global task and message schedule has not been considered in the existing frameworks. Also, there is still a gap between the details of the delay characteristics and the delay information that is used to characterize the control performance during optimization. Chapter 4 presents an integrated design framework towards the optimization of periods, schedules, and controllers that execute on distributed platforms with static cyclic or priority-based scheduling.

2.3 Adaptive Platforms for Control

Although many control systems are implemented on systems with static configuration of schedule parameters (e.g., priorities and schedule tables) and other parameters related to resource usage of the running control applications, there are situations in which appropriate mechanisms are needed to adapt to workload and environmental changes during system operation. Some examples of such situations are temporary overloads in computation and communication resources, large variations in execution times, and variations in external disturbances. To adapt to such changes, the system needs to be prepared to solve resource-management problems at runtime. Several such solutions have been proposed to maintain an operational and stable control system with a certain level of quality. In addition to performance and quality of service, it is important to consider the time overhead of adaptive mechanisms and to keep it as low as possible—for example, by solving most parts of the resource-management problem at design time.

The controller period is one of the parameters that can be adjusted at runtime to adapt to changes in the computational demand of the running

control applications. Runtime methods have been proposed [EHÅ00, CEBÅ02] to adjust controller periods and schedule the execution of control tasks on uniprocessor platforms. The optimization objective is given by a quadratic cost function, representing the control performance of multiple control tasks. The solution is based on feedback from the controlled plants, as well as feedforward information on the workload to the task scheduler that adjusts periods. In various contexts, several other approaches for adaptive period assignment have been presented [HC05, CVMC10, CMV+06]. Shin and Meissner [SM99] proposed algorithms for task reallocation and runtime assignment of task periods to be used in multiprocessor control systems with transient overloads and failures—however, with no consideration of delay and jitter effects on control performance.

Martí et al. [MLB+09] presented a resource-management framework, where the main contribution is feedback from controlled plants to the resource manager. The system allocates resources to the control tasks at runtime based on the measured state of the controlled plants. Buttazzo et al. [CBS02, BVM07] followed a similar line of motivation, extended towards control-quality optimization, for resource management in overloaded real-time systems. Martí et al. [MYV+04] formulated a message-scheduling problem to be solved at runtime, where the goal is to maximize the overall control quality in networked control systems. The optimization variables are the network bandwidths to be assigned to the individual control loops. Ben Gaid et al. [BCH09] presented a method that schedules control tasks based on feedback of plant states. The method is based on a runtime-scheduling heuristic that aims to optimize control performance by reacting to disturbances and selecting appropriate task schedules; a set of such static schedules are generated at design time [BCH06] and stored in the memory of the underlying execution platform. Henriksson et al. [HCÅÅ02a, HCÅÅ02b] presented solutions for feedback scheduling of multiple model-predictive control tasks on uniprocessor platforms. Model-predictive control is an approach in which control signals are optimized iteratively at runtime. An intrinsic property of such control approaches is the possibility to find trade-offs between execution time of the control task and the quality of the produced control signal. The feedback scheduler determines the order of execution of multiple periodic control tasks and decides when to terminate the optimization of a running control task based on its CPU usage

and control performance.

A multi-mode implementation of control applications is presented by Goswami et al. [GSC11b]. The authors characterize a control application by two modes: a transient phase and a steady-state phase. It is argued that a control application in a transient phase is more prone to performance degradations due to large and varying delays, compared to when it is in steady-state phase and the controlled plant is close to equilibrium. Considering distributed execution platforms with communication in both time-driven and event-driven communication segments (e.g, TTCAN [LH02] or FlexRay [Fle05]), Goswami et al. [GSC11b] proposed to implement the message communication implied by the control applications during their transient phase in the time-driven TDMA-based communication segment, which provides predictability. The steady-state phase is typically more robust to large delay variations and is thus implemented in the event-driven segment, which typically leads to more complex temporal behavior than the time-driven segment of FlexRay.

We have discussed several methods for runtime management of resources in embedded control systems. For the related approaches we have discussed in this section, the need for adaptation is motivated by variations in workload, overload scenarios, and disturbances. In Chapter 5, the resource-management problem is raised by the runtime-variation of the number of controlled plants. Appropriate mechanisms are needed to implement such multi-mode control systems for which each mode is characterized by a set of running control applications and needs a customized resource-allocation solution for optimized control quality.

2.4 Fault-Tolerant Computing and Control

The aggressive shrinking of transistor sizes and the environmental factors of modern embedded computing systems make electronic devices increasingly prone to faults [Bor05, BC11]. Broadly, faults may be classified as transient or permanent. Transient faults (e.g., caused by electromagnetic interference) manifest themselves for a short time and then disappear without causing permanent damage to devices. Permanent faults, however, sustain for much longer time intervals, or—in the worst case—for the remaining lifetime of the system. Permanent faults can occur due to aging, wear out, design defects, or manufacturing defects [KK07].

Because of the feedback mechanism in control loops, temporary performance degradations caused by transient faults are often automatically remedied in the next sampling instants. In addition, controllers are often synthesized with a certain amount of robustness to modeling errors and disturbances; transient faults may be considered as one possible motivation for synthesis of robust controllers. Integrated real-time scheduling and robust control have been studied by Palopolo et al. [PPS+02]. They investigated stability of control systems in the presence of perturbations in the control law. Recent research has also dealt with transient faults explicitly when analyzing and synthesizing control systems. Control over computer networks with noisy communication channels has received attention in literature [HNX07, SSF+07] with recent results by Sundaram et al. [SCV+11]. Robust control in the context of unspecified disturbances has been developed by Majumdar et al. [MRT11]. The authors applied their framework to control synthesis for robustness against transient faults. Another approach to fault tolerance in the context of embedded systems is to first assess the impact of faults on the application performance, and then take appropriate fault-tolerance measures. Towards such assessments, Skarin et al. [SBK10a, SBK10b] developed fault injection-based tools for experimental evaluation and assessment of system operation in the presence of different types of faults.

In the context of distributed control systems, the proposed analysis and synthesis methods in literature assume that computation nodes operate without permanent or long-term faults. In case a computation node fails due to a permanent fault, applications that are controlled by tasks running on this node will potentially become unstable. To avoid such situations, these tasks must now be activated and executed on other nodes. In addition to appropriate fault-detection mechanisms [Kop97], the system must adapt to situations in which nodes fail, as well as to reintegrate nodes in the system when these are operational again. It is thus important to construct control systems that are resilient to node failures and, in addition, provide as high control performance as possible with the available computation and communication resources. We shall address this problem in Chapter 6 by appropriate synthesis and optimization methods, as well as task replication and migration as reconfiguration mechanisms to be used at runtime in case nodes fail. Related approaches for embedded systems design and permanent faults, relying on task remapping by replication and migration, have been proposed by Pinello et al. [PCS08]

and Lee et al. [LKP+10]. These works differ significantly as compared to the material presented in Chapter 6. First, none of them focused on control-quality optimization. Further, the related methods consider a set of predefined fault scenarios. We present methods that guarantee proper operation in fault scenarios that are derived directly based on the inherent design constraints for embedded control systems. Finally, the proposed heuristic in Chapter 6 explores the design space judiciously to improve control quality in an incremental manner at design time.

2.5 Paradigms for Nonperiodic Control

For many control systems, it may be important to minimize the number of controller executions, while still providing a certain level of control performance. This is due to various costs related to sensing, computation, and actuation (e.g., computation time, communication bandwidth, and energy consumption). In addition to optimize control performance, it is thus important for the system to minimize the resource usage of the control tasks, in order to accommodate several control applications on a limited amount of computation and communication resources and, if needed, provide a certain amount of bandwidth to other applications. Although integrated computing and periodic control has been elaborated in literature—at least to a certain extent—periodic implementations can result in inefficient resource usage in many execution scenarios. The control tasks are triggered and executed periodically merely based on the elapsed time and not based on the states of the controlled plants. This may be inefficient for two reasons: First, resources are used unnecessarily much when a plant is close to equilibrium. Second, depending on the selected execution period, the resources might be used too little to provide a reasonable level of control quality when a plant is far from the desired state in equilibrium. The two inefficiencies also arise in situations with varying amount of external disturbance and noise on the system.

Event-based and self-triggered control are the two main approaches that have been proposed recently to address inefficient resource usage in periodic control systems. The main idea is that execution is not only triggered by the elapsed time—this is the only triggering parameter in periodic control—but also, and more importantly, by the actual state of the controlled process. It has been demonstrated that such adaptive, state-

based control approaches can lead to better control performance than their periodic counterpart, given a certain level of resource usage [Åst07]. Another viewpoint is that a certain level of control performance can be achieved with an event-based or self-triggered implementation with lower resource usage than a corresponding periodic implementation. As a result, compared to periodic control, event-based and self-triggered control can result in implementations with less number of sensing and actuation instants, lower usage of computation and communication resources, and lower energy consumption of the system at runtime. In addition, other application tasks (e.g., best-effort tasks) on the same platform as the control tasks can be allocated more resources.

2.5.1 Event-Based Control

Event-based control [Åst07] is an approach that can result in similar control performance as periodic control but with relaxed requirements on computation capacity. Several such approaches have been presented in literature in recent years [ÅB99, Årz99, Tab07, HJC08, CH08, HSv08, VMB09]. In event-based control, plant states are measured continuously to generate control events when needed, which then activate the control tasks that perform sampling, computation, and actuation. Periodic control systems can be considered as a special class of event-based systems that generate control events with a constant time period that is chosen independently of the states of the controlled plant.

Åström and Bernhardsson [ÅB99] motivated the use of event-based control for linear systems with one state perturbed by random noise. It was assumed that control events are generated whenever the plant state leaves a given region of the state space. The control action, which is executed as a response to events, is an impulse that resets the plant state to zero instantaneously. In the worst-case, infinite number of events may be generated in some time interval, making it impossible to bound the resource requirements of the underlying computation platform on which the applications execute.

To overcome the problem of unbounded resource requirements of event-based control, several researchers proposed to use sporadic control, by defining a minimum time interval between two control events [HJC08, CJ08]; this is a combination of event-based impulse control with sporadic real-time tasks. The sporadic control paradigm has also been applied

to scheduling of networked control systems [CH08]. Other event-based control strategies have been proposed over the years [HSv08, Cog09, LL10], although no resource-management framework to accommodate multiple event-based control loops on a shared platform has been presented in literature. Tabuada [Tab07] presented an event-triggered control method and further presented co-schedulability requirements for multiple real-time tasks with hard deadlines and a single event-triggered control task that executes with highest priority on a uniprocessor platform with fixed-priority scheduling.

While reducing resource usage, event-based control loops typically require specialized hardware—for example, ASIC (application-specific integrated circuit) implementations for continuous measurement or very high-rate sampling of plant states to generate control events. In event-based control systems, a control event usually implies that the control task has immediate or very urgent need to execute—and to communicate, if the control loop is closed over a network. This imposes very tight constraints on resource management and scheduling components in case multiple control applications execute on a shared platform.

2.5.2 Self-Triggered Control

Self-triggered control [VFM03, WL09, AT10, VMB08] is an alternative that leads to similar reduced levels of resource usage as event-based control, but without dedicated event-generator resources. A self-triggered control task computes deadlines on its future executions, by using the sampled states and the dynamical properties of the controlled system, thus canceling the need of specialized hardware components for event generation. The deadlines are computed based on stability requirements or other specifications of minimum control performance. Because the deadline of the next execution of a task is computed already at the end of the latest completed execution, a resource manager has, compared to event-based control systems, a larger time window and more options for task scheduling and optimization of control performance and resource usage.

Anta and Tabuada developed a method for self-triggered control of nonlinear systems [AT10, AT08a, AT08b]. An implementation of this self-triggered control approach on CAN-based embedded platforms has also been proposed [AT09]. Wang and Lemmon [WL09] developed a

theory for self-triggered control of linear systems, considering worst-case bounds on state disturbances. The main issues addressed in their work are release and completion times of control tasks, and, more specifically, conditions on the temporal behavior of self-triggered control systems to meet stability and performance requirements. These conditions on release and completion times depend on the sampled plant state, the model of the plant, and a given performance index. The timing requirements can thus be computed by the self-triggered control task itself. Further, Almeida et al. [ASP10] presented model-based control with self-triggered execution to further improve control performance compared to the already presented approaches in literature.

Although several methods for event-based and self-triggered control have been presented recently, the interaction and resource-sharing among multiple control loops has not received much attention in literature. In Chapter 7, we present a resource-management framework that can be used to accommodate multiple self-triggered control tasks on a given uniprocessor platform. We shall in that chapter investigate appropriate trade-offs between control performance and resource usage.

3 Preliminaries

THIS chapter presents preliminaries related to models and theory for control applications and their underlying distributed execution platform. We shall also introduce notation that is common for the remainder of this thesis. In Section 3.1, we shall discuss control design, modeling, and performance. Further, we present the application and platform model in Section 3.2.

3.1 Control Model and Design

In this section, we shall introduce the plant and disturbance models to be considered throughout this thesis. We continue with a discussion of the structure and performance of a feedback-control loop, including a physical plant and a controller implementation. The section ends with a discussion of controller synthesis in relation to the introduced metric of control performance.

3.1.1 Feedback-Control Loop

Figure 3.1 shows a block diagram of a typical feedback-control loop, including a controller and a plant, or physical process, to be controlled.

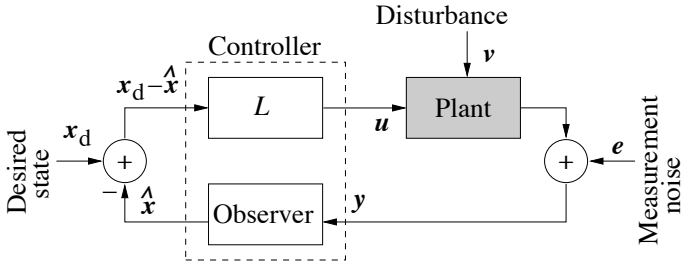

Figure 3.1: Structure of a feedback-control loop comprising a controller and the physical plant to be controlled. The controller communicates with sensors and actuators to implement a state observer and a state-feedback component.

The control problem is to control the state $\boldsymbol{x}$ of the plant to the desired state $\boldsymbol{x}_\mathrm{d}$ in the presence of continuous disturbances $\boldsymbol{v}$ acting on the plant. Sensors are connected to the plant in order to measure its outputs. These measurements, including measurement errors of the sensors, are inputs to the controller block, depicted in Figure 3.1 with a dashed rectangle. In practice, it is not possible to measure all plant states. Therefore, the controller includes an observer component that produces an estimate $\widehat{\boldsymbol{x}}$ of the plant state, based on the measured plant outputs $\boldsymbol{y}$. The feedback-gain block, denoted with L in the figure, computes the control signal $\boldsymbol{u}$ to be actuated based on the estimated error $\boldsymbol{x}_\mathrm{d} - \widehat{\boldsymbol{x}}$. The dashed rectangle in Figure 3.1 thus shows the control law

$$\boldsymbol{u} : \mathbb{R}^{n_{\boldsymbol{y}}} \longrightarrow \mathbb{R}^{n_{\boldsymbol{u}}},$$

where $n_{\boldsymbol{y}}$ and $n_{\boldsymbol{u}}$, respectively, are the number of measured plant outputs and the number of control signals applied to the plant. We shall in the continuation, depending on the context, use the notation $\boldsymbol{u}$ to refer to the control law and to refer to the actual plant input as well. When $\boldsymbol{u}$ is used to denote the plant input, it is a function of time

$$\boldsymbol{u} : [0, \infty) \longrightarrow \mathbb{R}^{n_{\boldsymbol{u}}}.$$

The desired state $\boldsymbol{x}_\mathrm{d}$ is typically given or is computed by a separate control component based on reference signals and the model of the controlled plant. In the continuation, without loss of generality for the linear time-invariant systems considered in this thesis, we shall consider

$\boldsymbol{x}_\mathrm{d} = 0$; the plant model is described in Section 3.1.2. In addition to state feedback, control applications can include additive feedforward information in the control signal. Feedforward is typically used to reduce measurable disturbances or to improve response to command signals. The feedback and feedforward components are usually designed separately and are evaluated against different performance metrics.

Control applications are typically implemented as periodic activities comprising computation and communication. Thus, a controller of a plant is characterized by a sampling period and a control law. Periodic implementations are very common in most embedded control systems (adaptive, event-based control strategies are emerging in application domains in which computation and communication resources are very limited [Åst07]; this is the topic of Chapter 7). The controller is implemented by one or several tasks that sample and process the plant outputs $\boldsymbol{y}$ periodically, and subsequently compute and update the control signal $\boldsymbol{u}$ according to the control law.

3.1.2 Plant Model

Throughout this thesis, we shall denote the set of plants to be controlled by $\mathbf{P}$. Let us also introduce its index set $\mathcal{I}_\mathbf{P}$. The synthesis problem is to construct controllers (periods and control laws) and to accommodate these on a given execution platform. Each plant P_i ($i \in \mathcal{I}_\mathbf{P}$) is modeled as a continuous-time linear system [ÅW97]. Specifically, this model is given by a set of differential equations

$$\dot{\boldsymbol{x}}_i(t) = A_i \boldsymbol{x}_i(t) + B_i \boldsymbol{u}_i(t) + \boldsymbol{v}_i(t), \tag{3.1}$$

where the vector functions of time $\boldsymbol{x}_i$ and $\boldsymbol{u}_i$ are the plant state and controlled input, respectively, and the vector $\boldsymbol{v}_i$ models plant disturbance as a continuous-time white-noise process with given variance R_{1i}. The matrices A_i and B_i model how the plant state evolves in time depending on the current plant state and provided control input, respectively. Typically, not all plant states can be measured by the available sensors. The measurable plant outputs are denoted with $\boldsymbol{y}_i$ and are modeled as

$$\boldsymbol{y}_i(t) = C_i \boldsymbol{x}_i(t) + \boldsymbol{e}_i(t), \tag{3.2}$$

where $\boldsymbol{e}_i$ is an additive measurement noise. The continuous-time output $\boldsymbol{y}_i$ is measured and sampled periodically and is used to produce the

control signal $\boldsymbol{u}_i$. The matrix C_i, which often is diagonal, indicates those plant states that can be measured by available physical sensors. If all states can be measured, the matrix C_i is the identity matrix and the linear model is only given by Equation 3.1. Because the plant outputs are sampled at discrete time instants, the measurement noise $\boldsymbol{e}_i$ is modeled as a discrete-time white-noise process with variance R_{2i}. The control signal is actuated at discrete time instants and is held constant between two updates by a hold circuit in the actuator [ÅW97].

As an example of a system with two plants, let us consider a set of two inverted pendulums $\mathbf{P} = \{P_1, P_2\}$. Each pendulum P_i ($i \in \mathcal{I}_{\mathbf{P}} = \{1, 2\}$) is modeled according to Equations 3.1 and 3.2, with

$$A_i = \begin{bmatrix} 0 & 1 \\ g/l_i & 0 \end{bmatrix},$$

$$B_i = \begin{bmatrix} 0 \\ g/l_i \end{bmatrix},$$

and

$$C_i = \begin{bmatrix} 1 & 0 \end{bmatrix},$$

where $g \approx 9.81$ m/s^2 and l_i are the gravitational constant and length of pendulum P_i, respectively ($l_1 = 0.2$ m and $l_2 = 0.1$ m). The two states are the pendulum position and speed. For the plant disturbance and measurement noise we have $R_{1i} = B_i B_i^{\mathsf{T}}$ and $R_{2i} = 0.1$. The inverted pendulum model appears often in literature as an example of control problems for unstable processes.

3.1.3 Control Quality

Considering one of the controlled plants P_i in isolation, the goal is to control the plant states in the presence of the additive plant disturbance $\boldsymbol{v}_i$ and measurement error $\boldsymbol{e}_i$. We use quadratic control costs [ÅW97] to measure the quality of a control application and its implementation. This includes a cost for the error in the plant state and the cost of changing the control signals to achieve a certain state error (the latter cost can be related to the amount of energy spent by the actuators). Specifically, the quality of a controller for plant P_i is given by the quadratic cost

$$J_i = \lim_{T \to \infty} \frac{1}{T} \mathrm{E} \left\{ \int_0^T \begin{bmatrix} \boldsymbol{x}_i \\ \boldsymbol{u}_i \end{bmatrix}^{\mathsf{T}} Q_i \begin{bmatrix} \boldsymbol{x}_i \\ \boldsymbol{u}_i \end{bmatrix} dt \right\}. \tag{3.3}$$

This stationary cost indicates the ability of the controller to reduce the disturbances in the system (i.e., to have a small control error) relative to the magnitude of the control signal (i.e., the amount of spent control energy). The cost J_i is decided partly by the controller period and the control law $\boldsymbol{u}_i$. The weight matrix Q_i is a positive semi-definite matrix (usually a diagonal matrix) that is used by the designer to assign weights to the individual components of the state $\boldsymbol{x}_i$ and the inputs $\boldsymbol{u}_i$ ($\mathrm{E}\{\cdot\}$ denotes the expected value of a stochastic variable). The weight of a state, which is given by the corresponding position in the matrix Q_i, indicates the importance of a small variance in the state relative to the other states. The controlled inputs are also given weights by appropriate elements in Q_i. In this way, the designer can indicate the desired trade-off between the state cost and a small variance in the controlled input.

For the example with the inverted pendulums (Section 3.1.2), a possible weight matrix is

$$Q_i = \mathrm{diag}\left(C_i^{\mathsf{T}} C_i, 0.002\right) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0.002 \end{bmatrix}.$$

This indicates the importance of having a control that achieves small variance in the pendulum position. The last element (0.002) indicates that it is also of some importance to have small variance in the controlled input (e.g., because of energy consumption in changing the value of the input), however with less importance than the pendulum state. For systems with several control loops, the matrix Q_i can also be used to transform the cost to a common baseline or to indicate importance relative to other control loops.

The quadratic cost in Equation 3.3 is a very common performance metric in the literature of control systems [ÅW97]. Note that a small control cost J_i indicates high control performance, and vice versa. The control cost is a function of the sampling period of the controller, the control law, and the characteristics of the delay between sampling and actuation. As we have discussed, this delay is complex and is induced not only by the computations and communications of the controller but also, and more important, by the interference experienced due to computation and communication delays of other tasks and messages on the platform. To compute the cost J_i for the examples and experimental results presented in this thesis, the Jitterbug toolbox [LC02, CHL+03, CL10]

has been used with the controller and characteristics of the sampling–actuation delay as inputs.

3.1.4 Controller Synthesis

The synthesis of a controller for a plant P_i comprises the determination of the sampling period h_i and the control law $\boldsymbol{u}_i$. The period h_i decides the rate of execution of the tasks that implement the control law. This period is typically decided based on the dynamics of the plant and the available resources, as well as trade-offs between the control performance and the performance of other applications running on the same execution platform. The control law determines how the sampled plant output is used to compute the control signal to be applied to the plant through actuator interfaces.

For a given sampling period h_i and a given constant sensor–actuator delay δ_i^{sa} (i.e., the time between sampling the output $\boldsymbol{y}_i$ and updating the controlled input $\boldsymbol{u}_i$), it is possible to find the control law $\boldsymbol{u}_i$ that minimizes the control cost J_i [ÅW97]. Thus, optimal delay compensation can be achieved if the delay of a certain control application is constant. It is, in general, difficult to synthesize an optimal control law for the case of varying delays in the control loop. As mentioned earlier, for those cases when an optimal controller can be synthesized, the control law is typically time varying and complicated to implement (e.g., time stamps may be required to find optimal delay compensation laws at runtime). A recent feature in the Jitterbug toolbox makes it possible to design a control law that is optimal for a given constant delay with a certain built-in robustness against given delay variations.

The control-design command in Jitterbug [CL10] produces a controller with the same structure as depicted in Figure 3.1. The controller is given by the following equations [CL10]:

$$\boldsymbol{u}[k] = -L\widehat{\boldsymbol{x}}_e[k \mid k] \tag{3.4}$$

$$\widehat{\boldsymbol{x}}_e[k \mid k] = \widehat{\boldsymbol{x}}_e[k \mid k-1] + K_f\left(\boldsymbol{y}[k] - C_e\widehat{\boldsymbol{x}}_e[k \mid k-1]\right) \tag{3.5}$$

$$\begin{aligned}\widehat{\boldsymbol{x}}_e[k+1 \mid k] = {} & \Phi_e\widehat{\boldsymbol{x}}_e[k \mid k-1] + \Gamma_e\boldsymbol{u}[k] \\ & + K\left(\boldsymbol{y}[k] - C_e\widehat{\boldsymbol{x}}_e[k \mid k-1]\right)\end{aligned} \tag{3.6}$$

The observer state vector $\widehat{\boldsymbol{x}}_e$ combines an estimate of the plant state and

the previous control signals as defined by

$$\widehat{\boldsymbol{x}}_e[k] = \begin{bmatrix} \widehat{\boldsymbol{x}}[k] \\ \boldsymbol{u}[k-1] \\ \vdots \\ \boldsymbol{u}[k-\ell] \end{bmatrix},$$

where

$$\ell = \max\left(1, \left\lceil \frac{\delta^{\text{sa}}}{h} \right\rceil\right).$$

As we have already mentioned, the period h and constant sampling–actuation delay δ^{sa} are given as inputs to the control-law synthesis procedure. In Equations 3.4–3.6, we use the notation $[\,\cdot\,]$ to denote the value of a variable at a certain sampling instant. For example, $\boldsymbol{y}[k]$ denotes the output of the plant at the k^{th} sampling instant, whereas $\boldsymbol{u}[k]$ denotes the controlled input at actuation k. The notation $\widehat{\boldsymbol{x}}_e[k \mid k-1]$ means the estimate of the plant state at sampling instant k, when computed at sampling instant $k-1$. At a certain sampling instant k, the controller measures the output $\boldsymbol{y}[k]$ and uses it to compute an estimate $\widehat{\boldsymbol{x}}_e[k \mid k]$ of the plant state at the current sampling instant (Equation 3.5). This computation is also based on an estimate $\widehat{\boldsymbol{x}}_e[k \mid k-1]$ of the current state for which the estimation was performed at the previous sampling instant (Equation 3.6). Following that, the controlled input $\boldsymbol{u}[k]$ to be actuated is computed according to Equation 3.4. Last, the state at the next sampling instant $k+1$ is estimated according to Equation 3.6. The parameters L, K_f, K, C_e, Φ_e, and Γ_e in Equations 3.4–3.6 are outputs of the controller synthesis and are optimized for the given period and constant sensor–actuator delay.

The quality of a controller is degraded (its cost J_i is increased) if the sensor–actuator delay is different from what was assumed during the control-law synthesis, or if this delay is not constant (i.e., there is jitter). Considering that the sensor–actuator delay is represented as a stochastic variable Δ_i^{sa} with probability function $\xi_{\Delta_i}^{\text{sa}}$, we can compute the cost J_i with the Jitterbug toolbox [LC02]. This is done for the cost computations related to the results presented in this thesis. The delay characteristics of the system depend on the scheduling policy and communication protocol, as well as their parameters and configuration. Optimization of scheduling and communication with regard to control delays and performance of multiple distributed control applications is the subject of Chapter 4.

3.2 System Model

This section contains models and notation for the execution platform and their running control applications. We shall also discuss mapping and scheduling of the tasks and messages in the system.

3.2.1 Platform Model

The control applications execute on a distributed execution platform with multiple computation nodes that communicate on a bus. The computation nodes, which are also called electronic control units in the automotive domain, have input–output interfaces to the sensors and actuators of the controlled plants. In addition, a computation node comprises a CPU, several memories (e.g., RAM or ROM memories), an operating system, and a communication controller. The operating system implements a scheduling policy (e.g., static cyclic scheduling or priority-based scheduling) for the executions on the CPU. The communication of messages between computation nodes is conducted by the communication controllers according to a certain communication protocol (e.g., TTP [Kop97], FlexRay [Fle05], or CAN [Bos91]). Let us denote the set of computation nodes with $\mathbf{N}$ and its index set with $\mathcal{I}_{\mathbf{N}}$. Figure 3.2 shows two computation nodes ($\mathbf{N} = \{N_1, N_2\}$) that are connected to a bus (the communication controllers are denoted CC). The figure also shows that the computation nodes are connected to sensors and actuators. The nodes have input–output interfaces that are used by the control applications to measure and read plant outputs, and to write control signals to actuators.

3.2.2 Application Model

The applications that execute and communicate on the distributed platform comprise a set of tasks and messages. Let us denote the set of all applications with $\mathbf{\Lambda}$. Each application represents the controller of a certain plant. We shall use the set $\mathcal{I}_{\mathbf{P}}$—the same index set as for the set of plants $\mathbf{P}$—to index applications in $\mathbf{\Lambda}$. Thus, application Λ_i is the controller for plant P_i ($i \in \mathcal{I}_{\mathbf{P}}$). An application $\Lambda_i \in \mathbf{\Lambda}$ is modeled as a directed acyclic graph

$$\Lambda_i = (\mathbf{T}_i, \mathbf{\Gamma}_i).$$

The vertices in the set $\mathbf{T}_i$, which are indexed by $\mathcal{I}_i$, represent computation tasks, whereas the edges in the set $\mathbf{\Gamma}_i \subset \mathbf{T}_i \times \mathbf{T}_i$ represent messages

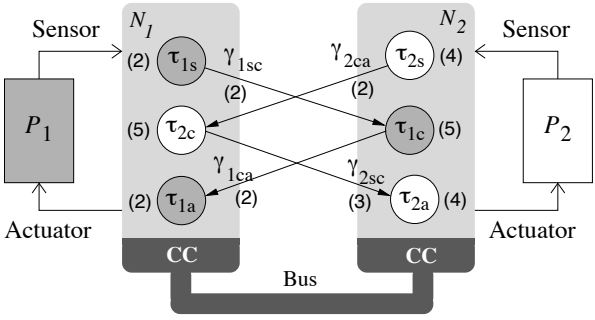

Figure 3.2: Example of an execution platform. Two computation nodes transmit and receive messages on the bus through the communication controllers (CC). Two control applications, comprising three tasks each, execute on the two nodes.

between tasks (data dependencies). We also introduce the set of all tasks in the system as

$$\mathbf{T}_{\mathbf{\Lambda}} = \bigcup_{i \in \mathcal{I}_{\mathbf{P}}} \mathbf{T}_i.$$

The specification of an application as a set of tasks is based on the structure of the application and its different types of computations. In Figure 3.2, we show two applications $\mathbf{\Lambda} = \{\Lambda_1, \Lambda_2\}$ ($\mathcal{I}_{\mathbf{P}} = \{1, 2\}$), which are controllers for the two pendulums P_1 and P_2 in the example in Section 3.1.2. For $i \in \mathcal{I}_{\mathbf{P}}$, the task set of application Λ_i is $\mathbf{T}_i = \{\tau_{is}, \tau_{ic}, \tau_{ia}\}$ with index set $\mathcal{I}_i = \{s, c, a\}$. For example, task τ_{1s} on node N_1 is the sensor task that measures the position of the pendulum by reading and processing data from the sensor. A message γ_{1sc} is sent on the bus to node N_2, whereafter the control task τ_{1c} computes the control signal to be actuated. The actuation is done by task τ_{1a} on node N_1, which is the only node connected to the actuator of P_1. An arbitrary task shall in the continuation be denoted by τ. A task of application Λ_i is denoted τ_{ij}, where $i \in \mathcal{I}_{\mathbf{P}}$ and $j \in \mathcal{I}_i$. An arbitrary message between tasks τ_{ij} and τ_{ik} in application Λ_i is denoted γ_{ijk}.

Each application $\Lambda_i \in \mathbf{\Lambda}$ has a period h_i, which decides the rate of execution of Λ_i. Jobs of the application tasks are released for execution periodically and are scheduled for execution according to the scheduling policies of the computation nodes. Thus, at time $(q-1)h_i$, a job of each task in the application is released for execution. Job q of

task τ_{ij} is denoted $\tau_{ij}^{(q)}$ and is released at time $(q-1)h_i$. For a message $\gamma_{ijk} = (\tau_{ij}, \tau_{ik}) \in \mathbf{\Gamma}_i$, the message instance produced by job $\tau_{ij}^{(q)}$ is denoted $\gamma_{ijk}^{(q)}$. An edge $\gamma_{ijk} = (\tau_{ij}, \tau_{ik}) \in \mathbf{\Gamma}_i$ means that the earliest start time of a job $\tau_{ik}^{(q)}$ is when $\tau_{ij}^{(q)}$ has completed its execution and the produced message instance $\gamma_{ijk}^{(q)}$ has been communicated to the computation node that hosts task τ_{ik}. Let us also define the hyper period $h_{\mathbf{\Lambda}}$ of all control applications as the least common multiple of all application periods. Further, a task can have a deadline, which means that any job of that task must finish within a given time relative to its release. Control applications do not typically have hard timing constraints, but instead the goal is to achieve a certain level of control quality. Additional tasks, representing other application domains (e.g., safety-critical applications with strict timing constraints, or best-effort applications), may coexist with the control applications on the distributed execution platform.

3.2.3 Mapping and Scheduling

Each task in the system is mapped to a computation node. The mapping is given by a function

$$\mathsf{map} : \mathbf{T}_{\mathbf{\Lambda}} \longrightarrow \mathbf{N}$$

that returns the computation node of a certain task in the system. Typically, a certain task may be mapped only to a subset of the computation nodes of the system. For example, tasks that read sensors or write to actuators can only be mapped to computation nodes that provide input–output interfaces to the needed sensors and actuators. Also, some tasks may require specialized instructions or hardware accelerators that are available only on some nodes in the platform. To model such mapping constraints, we consider given a function

$$\Pi : \mathbf{T}_{\mathbf{\Lambda}} \longrightarrow 2^{\mathbf{N}}$$

that, for each task $\tau \in \mathbf{T}_{\mathbf{\Lambda}}$ in the system, gives the set of computation nodes $\Pi(\tau) \subseteq \mathbf{N}$ that task τ can be mapped to. We shall treat mapping constraints further in Chapter 6 for the remapping problem that arises as a consequence of failed computation nodes. Let us proceed by introducing the function

$$\mathsf{map}^* : \mathbf{N} \longrightarrow 2^{\mathbf{T}_{\mathbf{\Lambda}}}$$

that, for a certain node N_d, returns the set of tasks that are mapped to N_d. Thus, we have

$$\mathsf{map}^*(N_d) = \{\tau_{ij} \in \mathbf{T}_{\mathbf{\Lambda}} : \mathsf{map}(\tau_{ij}) = N_d\}\,.$$

A message between tasks mapped to different nodes is sent on the bus. Thus, the set of messages that are communicated on the bus is

$$\mathbf{\Gamma}_{\text{bus}} = \left\{ (\tau_{ij}, \tau_{ik}) \in \bigcup_{i \in \mathcal{I}_{\mathbf{P}}} \mathbf{\Gamma}_i : \mathsf{map}(\tau_{ij}) \neq \mathsf{map}(\tau_{ik}) \right\}.$$

For a message instance $\gamma_{ijk}^{(q)}$, we denote with c_{ijk} the communication time when there are no conflicts on the bus. For our example with the mapping in Figure 3.2, we note that the set of messages on the bus is $\mathbf{\Gamma}_{\text{bus}} = \mathbf{\Gamma}_1 \cup \mathbf{\Gamma}_2$. Given a mapping of the tasks to the computation nodes, we have, for each task, a specification of possible execution times during runtime. We model the execution time of task τ_{ij} as a stochastic variable c_{ij} with probability function $\xi_{c_{ij}}$. It is assumed that the execution time is bounded by given best-case and worst-case execution times, denoted c_{ij}^{bc} and c_{ij}^{wc}, respectively. These properties are obtained with tools and methods for simulation, profiling, and program analysis [WEE+08]. In Figure 3.2, the execution times (constant in this example) and communication times for the tasks and messages are given in milliseconds inside parentheses.

In this thesis, we consider systems with static cyclic scheduling or priority-based scheduling of tasks and messages. For static cyclic scheduling, tasks are executed on each computation node according to schedule tables that are constructed at design time. Similarly, the communication controllers conduct the communication on the bus according to tables with start times for the transmissions. An example of such a statically scheduled system is the time-triggered architecture with the TTP communication protocol [Kop97]. Examples of priority-based scheduling policies with static priorities are rate-monotonic and deadline-monotonic scheduling, whereas the earliest-deadline-first policy is a representative of a scheduling algorithm with dynamic priorities [LL73, But97, KS97, SSDB95]. The most prominent example of a communication protocol for priority-based scheduling of messages is CAN [Bos91]. The design frameworks that are proposed in this thesis support systems with static-cyclic scheduling according to schedule tables and priority-based

scheduling with static priorities that are synthesized at design time. These parameters related to scheduling and communication are synthesized at design time and influence the delay characteristics (delay and jitter) of the system at runtime. As we have discussed in Section 3.1.3, the delay is an important and influential parameter that decides the control performance. Thus, optimization of parameters related to scheduling and communication in embedded control systems is an important design activity, in addition to period selection and control-law synthesis. We shall in the beginning of the next chapter motivate such optimizations with an example (Section 4.1).

4

Synthesis of Distributed Control Systems

We shall in this chapter present the formulation and solution of a control–scheduling co-design problem for distributed embedded systems. The objective is to optimize the overall performance of several control loops in the presence of plant disturbance and measurement noise. We shall consider the temporal properties of the system and their influence on controller design and performance. In addition, we consider control performance as a driving factor in system-level optimization of embedded systems. As part of the optimization, we synthesize a controller (sampling period and control law) for each plant. Further, considering both static cyclic scheduling and priority-based scheduling of the tasks and messages in the system, we schedule the execution and communication of the control applications on the given distributed execution platform. The resulting optimization method integrates controller design with system-level design and leads to design solutions with significant performance improvements relative to traditional design methods for distributed embedded control systems.

The remainder of this chapter is organized as follows. The motivation and formal statement of the co-design problem are presented in

Sections 4.1 and 4.2, respectively. The integrated design approach is presented in Section 4.3. Details of the solution related to static cyclic scheduling and priority-based scheduling, respectively, are presented in Sections 4.4 and 4.5. We present an experimental validation of the proposed approach in Section 4.6, followed by a case study on an automotive control application in Section 4.7. The contribution of the chapter is summarized and discussed in Section 4.8.

4.1 Motivational Example

In Section 3.1.2, we introduced an example in which two inverted pendulums are controlled over a network of two computation nodes. Let us consider that the two pendulums are controlled by the two applications with three communicating tasks each as depicted in Figure 3.2 on page 33. The somewhat unrealistic mapping of tasks to computation nodes is chosen such that the example has communication of messages on the bus. This enables us to illustrate the key issues treated in this chapter by a small and comprehensible example. Towards this, we shall consider static cyclic scheduling as the scheduling policy of tasks and messages, although the discussions and conclusions of this section are valid also for other scheduling policies and communication protocols. All time quantities are given in milliseconds throughout this section.

4.1.1 Ideal Execution Scenario

Let us now consider that the periods of Λ_1 and Λ_2 are $h_1 = 20$ and $h_2 = 30$, respectively. We shall consider that optimal control design has been used to synthesize the control laws $\boldsymbol{u}_1 : \mathbb{R} \longrightarrow \mathbb{R}$ and $\boldsymbol{u}_2 : \mathbb{R} \longrightarrow \mathbb{R}$ for the chosen periods and constant delays between sampling and actuation of each of the two control loops. The delay that has been considered during control-law synthesis is equal to the sum of task execution times; this isolates control design from the particularities of the underlying execution platform and communication protocol. Thus, we have a control law $\boldsymbol{u}_1$ that gives optimal control performance in the sense that it minimizes the control cost defined in Equation 3.3 on page 28 and compensates optimally for a constant delay. The control performance is optimal if the control law $\boldsymbol{u}_1$ is executed periodically without jitter every 20 milliseconds and with a constant delay of 9 milliseconds between

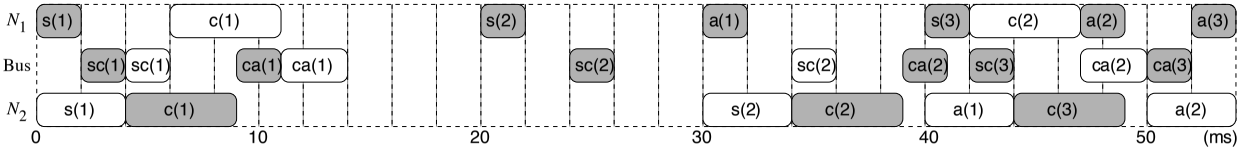

Figure 4.1: Task and message schedule for the periods $h_1 = 20$ ms and $h_2 = 30$ ms. The construction of the schedule does not consider control performance.

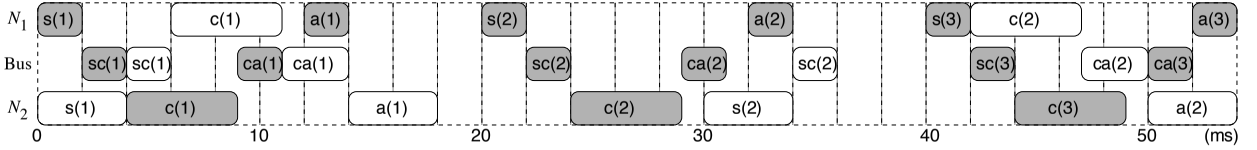

Figure 4.2: Improved schedule for the periods $h_1 = 20$ ms and $h_2 = 30$ ms. The schedule has been constructed to reduce sampling–actuation delays and their variance.

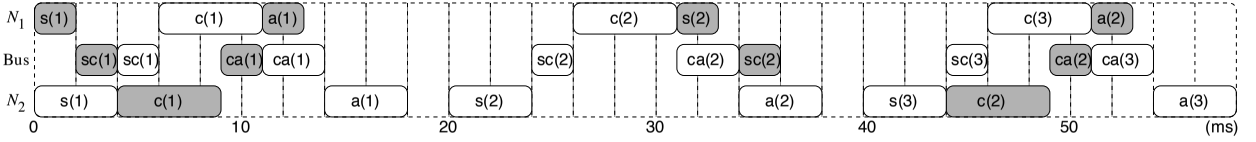

Figure 4.3: Schedule for the periods $h_1 = 30$ ms and $h_2 = 20$ ms.

sampling and actuation. Similarly, we have an optimal control law $\boldsymbol{u}_2$ for the period 30 and the constant delay 13 for the second control loop. The individual control costs are computed with Jitterbug to $J_1 = 0.9$ and $J_2 = 2.4$, giving a total control cost of $J_{\text{tot}} = J_1 + J_2 = 3.3$ for the entire system. These costs indicate the control performance that is achieved if control delays are constant during execution.

4.1.2 Effect of Execution and Communication

In practice, because the control applications share computation and communication resources, it is usually not possible to schedule the task executions and message transmissions such that all control loops execute with strict periodicity and with the constant sampling–actuation delay assumed during control-law synthesis. The control performance is degraded if there are sampling jitters or delay variations. The control cost for such scenarios is computed with Jitterbug by providing as inputs the synthesized control loop together with the specification of the actual delays during controller execution. Let us consider the system schedule in Figure 4.1. The schedule is shown with three rows for node N_1, the bus, and node N_2, respectively. The boxes depict the task executions and message transmissions. The grey boxes show the execution of control application Λ_1, whereas the white boxes show the execution of Λ_2. Each box is labeled with an index that specifies the corresponding task or message, and with a number that specifies the job or message instance. For example, the white box labeled $a(2)$ shows the execution of job $\tau_{2a}^{(2)}$ of the actuator task τ_{2a}. The job starts and finishes at times 50 and 54, respectively. The grey box labeled $sc(1)$ shows the first message $\gamma_{1sc}^{(1)}$ between the sensor and controller task of Λ_1. The schedule of execution and communication is repeated with the period 60 (the hyper period $h_{\boldsymbol{\Lambda}}$ of the application set $\boldsymbol{\Lambda}$).

Let us now focus on the delays between sampling and actuation. We consider that the outputs of the plants are sampled periodically without jitter (e.g., by some dedicated hardware mechanism that stores the sampled data in buffers, which are read by the sensor tasks). The sampling–actuation delay of control application Λ_i is denoted Δ_i^{sa}. In the schedule in Figure 4.1, we have three instances of Λ_1. The three actuations $\tau_{1a}^{(1)}$, $\tau_{1a}^{(2)}$, and $\tau_{1a}^{(3)}$ complete at times 32, 49, and 54, respectively. By considering the sampling period 20, we observe that the sampling–actuation

delays are 32, 29, and 14, which are repeated during the execution of the control applications. Each of these delays occurs with the same frequency during execution. By modeling the sampling–actuation delay as a stochastic variable Δ_i^{sa}, we note that the nonzero values of the probability function are given by $\xi^{\text{sa}}_{\Delta_1}(14) = \xi^{\text{sa}}_{\Delta_1}(29) = \xi^{\text{sa}}_{\Delta_1}(32) = 1/3$. Thus, if the control system is implemented according to the schedule in Figure 4.1, the delay of Λ_1 is different from 9 (the delay assumed during controller synthesis), and, moreover, it is not constant. By using the Jitterbug toolbox and providing as input the probability function $\xi^{\text{sa}}_{\Delta_1}$, we obtained a much larger cost $J_1 = 4.2$, which indicates a worse control performance than an ideal implementation that does not violate the assumptions regarding delays during control-law synthesis. Similarly, the two instances of application Λ_2 have the delays 44 and 24. We thus have $\xi^{\text{sa}}_{\Delta_2}(24) = \xi^{\text{sa}}_{\Delta_2}(44) = 1/2$ and the corresponding control cost is $J_2 = 6.4$. The total cost of the whole system is $J_{\text{tot}} = 10.6$, and is increased significantly from a total of 3.3 as a result of the implementation in Figure 4.1.

4.1.3 Integrated Control and Scheduling

Scheduling To obtain a better control performance, it is important to reduce delay and jitter. The delay characteristics and their impact on control performance are important to consider when constructing the execution schedule. Considering the same periods and control laws as before, let us study the schedule in Figure 4.2. With this schedule, the sensor–actuator delay is 14 for all the three instances of Λ_1; that is, $\xi^{\text{sa}}_{\Delta_1}(14) = 1$. Similarly, we have $\xi^{\text{sa}}_{\Delta_2}(18) = \xi^{\text{sa}}_{\Delta_2}(24) = 1/2$. The second control loop has a smaller delay with less jitter than in Figure 4.1. When the system operates with the schedule in Figure 4.2, the control costs are $J_1 = 1.1$ and $J_2 = 5.6$. This results in a total cost of $J_{\text{tot}} = 6.7$, which indicates a better overall control performance compared to the previous schedule; as already discussed, the previous schedule incurs a very high control cost of 10.6. An important observation in the context of integrated control and computing is thus that the control performance may be improved if the tasks and messages are scheduled properly to reduce delay and jitter.

Delay Compensation While it is important to consider control performance during system scheduling, it is possible to achieve further im-

provements by synthesizing the control laws to compensate for the sensor–actuator delays. Let us consider the same schedule (Figure 4.2) and redesign the control laws. We have synthesized the control law $\boldsymbol{u}_1$ for the same period as before but for the delay 14, which is the constant sensor–actuator delay in the schedule. The control law for P_2 is redesigned for the average sensor–actuator delay $\mathrm{E}\{\Delta_2^{\mathrm{sa}}\} = 21$. With the schedule in Figure 4.2 and the new control laws, which are customized for the schedule, the costs of the implementation are $J_1 = 1.0$ and $J_2 = 3.7$. The total cost is $J_{\mathrm{tot}} = 4.7$ and it shows that performance improvements can be obtained by considering the system schedule when synthesizing control laws. We observe that the negative influence of delays can be compensated for both by appropriate scheduling and by appropriate delay compensation during control design.

Period Selection As a last step, let us change the periods of the two control applications to $h_1 = 30$ and $h_2 = 20$. Figure 4.3 shows a schedule with two instances of Λ_1 and three instances of Λ_2 (the period of the schedule is 60). It can be seen that the first control loop has some delay jitter, whereas the second control loop executes periodically with a constant delay between sampling and actuation. The delays in the first control loop are 13 and 23, which gives $\xi^{\mathrm{sa}}_{\Delta_1}(13) = \xi^{\mathrm{sa}}_{\Delta_1}(23) = 1/2$. We have redesigned the control law $\boldsymbol{u}_1$ for the period 30 and a constant delay of $\mathrm{E}\{\Delta_1^{\mathrm{sa}}\} = 18$. The delay in the second control loop is 14 (constant). The control law $\boldsymbol{u}_2$ is synthesized for the period 20 and the delay 14. The evaluation resulted in the costs $J_1 = 1.3$ and $J_2 = 2.1$. The total control cost is thus $J_{\mathrm{tot}} = 3.4$, which is a significant quality improvement relative to the previous design solution. Appropriate period optimization is important for several reasons. First, the plants have different dynamics (some are faster than others), which should be reflected in the choice of periods. Second, some selection of periods may enable the possibility of finding a schedule with good delay characteristics (small and low-varying delays), which in turn may favor the delay compensation during control-law synthesis. To conclude, the example in this section highlights several design trade-offs and illustrates the importance of a proper integration of period selection, scheduling, and control synthesis.

4.2 Problem Formulation

Let us define the control–scheduling co-design problem treated in this chapter by specifying the required inputs, the decision parameters, and the optimization objective. The inputs are

- a set of plants $\mathbf{P}$ to be controlled, where each plant $P_i \in \mathbf{P}$ ($i \in \mathcal{I}_{\mathbf{P}}$) is a continuous-time linear system;
- a set of periodic applications $\mathbf{\Lambda}$, each modeled as a set of data-dependent tasks that control a plant;
- a finite set of available sampling periods $\mathbf{H}_i$ for each control application Λ_i ($i \in \mathcal{I}_{\mathbf{P}}$);
- a set of computation nodes $\mathbf{N}$ connected to a single bus;
- a mapping function $\mathsf{map} : \mathbf{T}_{\mathbf{\Lambda}} \longrightarrow \mathbf{N}$ of the whole task set;
- a scheduling policy for the tasks and the communication protocol for messages; and
- execution-time distributions of the tasks and communication times of the messages.

The set of available sampling periods $\mathbf{H}_i$ of a control application Λ_i is specified by the designer and should cover a reasonable range of different sampling rates to enable a proper exploration of trade-offs among the sampling periods of the multiple control applications. The inputs to the design problem are essentially described in detail in Sections 3.1.2 and 3.2. In addition, we consider that other application tasks with hard deadlines and given release periods may coexist with the control applications. For static cyclic scheduling of the system, we consider non-preemptive execution of tasks, whereas for priority-based scheduling, we consider preemptive task execution. The communication on the bus is non-preemptive.

The outputs related to the controller synthesis are the period $h_i \in \mathbf{H}_i$ and the control law $\boldsymbol{u}_i$ (the controller in Figure 3.1) for each plant P_i. The outputs related to scheduling depend on the scheduling policy and communication protocol adopted by the computation platform. For static cyclic scheduling of tasks and messages, the output is a schedule table

with start times of the job executions and the communications on the bus. In the case of priority-based scheduling, the output is a priority for each task and message. In both cases, it must be guaranteed that the specified task deadlines for the hard real-time applications are met at runtime. The cost function to be minimized is the overall control cost

$$J = \sum_{i \in \mathcal{I}_{\mathbf{P}}} J_i, \tag{4.1}$$

where J_i is given by Equation 3.3. Note that this formulation indicates optimization of control quality as defined by the control cost in Equation 3.3 on page 28. As we discussed in more detail in Section 3.1.3, appropriate weights can be embedded in the Q_i matrices in Equation 3.3 to indicate relative importance of the different control applications.

4.3 Scheduling and Synthesis Approach

This section starts with a general description of our proposed solution. The overall description is valid independently of the underlying scheduling policy of the computation and communication platform. The details of the solution related to static cyclic scheduling are described in Section 4.4, whereas the case of priority-based scheduling is treated in Section 4.5.

Figure 4.4 illustrates the overall approach. The dashed box in the figure shows the portion of the flowchart that is specific for static cyclic scheduling (Section 4.4). If priority-based scheduling is the scheduling policy of tasks and messages, this box is replaced by the flowchart in Figure 4.5 on page 55 (Section 4.5). In the outer loop, we iterate over several assignments of the controller periods. In each iteration, we choose a period for each control application in the set of available periods. Thus, a solution candidate for period assignment is a tuple

$$\boldsymbol{h} = \left(h^{(1)}, \ldots, h^{(|\mathcal{I}_{\mathbf{P}}|)}\right) \in \mathbf{H}_{\sigma_{\mathbf{P}}(1)} \times \cdots \times \mathbf{H}_{\sigma_{\mathbf{P}}(|\mathcal{I}_{\mathbf{P}}|)} = \prod_{p=1}^{|\mathcal{I}_{\mathbf{P}}|} \mathbf{H}_{\sigma_{\mathbf{P}}(p)}, \tag{4.2}$$

where

$$\sigma_{\mathbf{P}} : \{1, \ldots, |\mathcal{I}_{\mathbf{P}}|\} \longrightarrow \mathcal{I}_{\mathbf{P}}$$

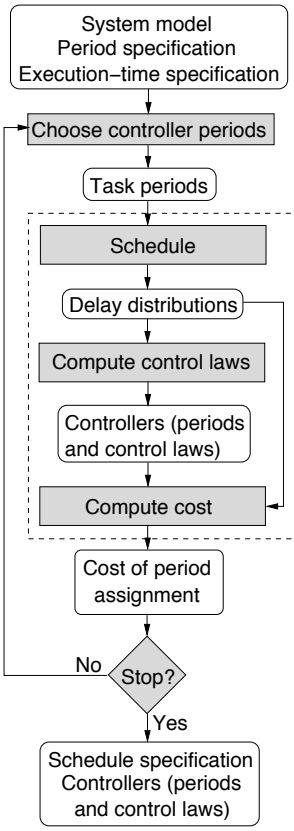

Figure 4.4: Overall approach with iterative assignment of controller periods. The dashed block for scheduling and control-law synthesis is specific to platforms with static cyclic scheduling.

is any bijection, which gives an order of the control applications, and $\prod$ denotes the Cartesian product of sets. The period h_i of controller Λ_i ($i \in \mathcal{I}_{\mathbf{P}}$) is thus

$$h_i = h^{\left(\sigma_{\mathbf{P}}^{-1}(i)\right)}.$$

For each period assignment $\boldsymbol{h}$ that is considered in the period-exploration phase, the following steps are performed (the dashed rectangle in Figure 4.4):

1. Schedule all tasks and messages of the system,

2. synthesize the control law $\boldsymbol{u}_i$ of each plant P_i, and

3. compute the cost J_i of the controller for each plant P_i.

These synthesis steps and cost computations result in a final cost $J_{\boldsymbol{h}}$ of the period assignment $\boldsymbol{h}$ under consideration. The details of the first step depend on the underlying scheduling policy of the platform and are described in Sections 4.4 and 4.5 for static cyclic scheduling and priority-based scheduling, respectively. In the remainder of this section, we shall consider that the results of the first step are the delay characteristics of the controllers. The sampling–actuation delay of controller Λ_i ($i \in \mathcal{I}_{\mathbf{P}}$) is represented as a stochastic variable Δ_i^{sa}. The probability function $\xi_{\Delta_i}^{\text{sa}}$ of this delay is determined by the execution-time distributions $\xi_{c_{ij}}$ of the tasks and the scheduling of the whole system. In Sections 4.4 and 4.5, we shall discuss the determination of $\xi_{\Delta_i}^{\text{sa}}$ for the case of static cyclic scheduling and priority-based scheduling, respectively. Considering the delay distributions for a certain period assignment $\boldsymbol{h}$ given, let us proceed by describing the last two steps: synthesis of control laws and computation of control costs.

4.3.1 Synthesis and Cost Computation for Given Periods

Given is an assignment of controller periods $\boldsymbol{h} = (h^{(1)}, \ldots, h^{(|\mathcal{I}_{\mathbf{P}}|)})$, where $h^{(p)}$ is the period of controller $\Lambda_{\sigma_{\mathbf{P}}(p)}$ ($1 \leqslant p \leqslant |\mathcal{I}_{\mathbf{P}}|$). From the task and message scheduling step, we also know the sensor–actuator delay Δ_i^{sa}—with given probability distribution—for each control application Λ_i. Let us first describe the synthesis of control laws. Each control law $\boldsymbol{u}_i$ is synthesized to minimize the cost J_i for the sampling period $h^{\left(\sigma_{\mathbf{P}}^{-1}(i)\right)}$ and a constant sampling–actuation delay $\delta_i^{\text{sa}} = \mathrm{E}\{\Delta_i^{\text{sa}}\}$, which is the expected value of the delay. Thus, the controllers are designed to compensate optimally for average delays. For each plant P_i ($i \in \mathcal{I}_{\mathbf{P}}$), we now have a controller Λ_i with a period h_i and a control law $\boldsymbol{u}_i$ synthesized for a constant delay of $\mathrm{E}\{\Delta_i^{\text{sa}}\}$.

In general, the actual implementation of the synthesized controllers results in a varying sensor–actuator delay. As we have discussed, this delay is modeled as a stochastic variable Δ_i^{sa} for which the probability function $\xi_{\Delta_i}^{\text{sa}}$ is given by the system schedule for tasks and messages (Sections 4.4 and 4.5). We compute the cost J_i of each control loop with Jitterbug [LC02, CHL$^+$03] based on the stochastic delay Δ_i^{sa}. The total

cost associated with the period assignment $\boldsymbol{h}$ is given by Equation 4.1 as

$$J_{\boldsymbol{h}} = \sum_{i \in \mathcal{I}_{\mathbf{P}}} J_i.$$

4.3.2 Period Optimization Based on Genetic Algorithms

The period-exploration process is based on genetic algorithms [Ree93, Mic96, Hol75]. An initial population

$$\boldsymbol{\Psi}_1 \subset \prod_{p=1}^{|\mathcal{I}_{\mathbf{P}}|} \mathbf{H}_{\sigma_{\mathbf{P}}(p)}$$

is generated randomly and comprises several solution candidates for the period assignment problem. At each iteration $k > 0$, the cost $J_{\boldsymbol{h}}$ of each member $\boldsymbol{h} \in \boldsymbol{\Psi}_k$ in the population is computed by performing the three steps discussed previously: system scheduling, control-law synthesis, and cost computation (Section 4.3.1). Following that, the crossover and mutation operators are applied to the members of $\boldsymbol{\Psi}_k$ to generate the next population $\boldsymbol{\Psi}_{k+1}$ to be evaluated in the next iteration.

Let us discuss the crossover and mutation implementation in more detail. Crossover is performed on the current population

$$\boldsymbol{\Psi}_k \subset \prod_{p=1}^{|\mathcal{I}_{\mathbf{P}}|} \mathbf{H}_{\sigma_{\mathbf{P}}(p)},$$

to obtain a set of offsprings

$$\boldsymbol{\Psi}_k^{\text{offspr}} \subset \prod_{p=1}^{|\mathcal{I}_{\mathbf{P}}|} \mathbf{H}_{\sigma_{\mathbf{P}}(p)}.$$

The members in $\boldsymbol{\Psi}_k^{\text{offspr}}$ are created by applying the crossover operator to distinct parent members successively. A crossover operation is initiated by selecting two distinct parents $\boldsymbol{h}_1, \boldsymbol{h}_2 \in \boldsymbol{\Psi}_k$. The first parent $\boldsymbol{h}_1$ is selected from $\boldsymbol{\Psi}_k$ randomly, where the probability for a member to be selected is induced by its control cost. The second parent $\boldsymbol{h}_2$ is selected randomly with a uniform, cost-independent probability distribution over the members of the current population $\boldsymbol{\Psi}_k$. Let us consider

$$\boldsymbol{h}_1 = \left(h_1^{(1)}, \ldots, h_1^{(|\mathcal{I}_{\mathbf{P}}|)}\right)$$

and any bijection

$$\sigma_{\mathbf{P}} : \{1, \ldots, |\mathcal{I}_{\mathbf{P}}|\} \longrightarrow \mathcal{I}_{\mathbf{P}}.$$

Similar to our discussion around Equation 4.2, we note that the period assignment $\boldsymbol{h}_1$ assigns the period $h_1^{(p)}$ to control application $\Lambda_{\sigma_{\mathbf{P}}(p)}$. In a similar manner for the second parent, we have

$$\boldsymbol{h}_2 = \left(h_2^{(1)}, \ldots, h_2^{(|\mathcal{I}_{\mathbf{P}}|)}\right).$$

The crossover operator is applied to $\boldsymbol{h}_1$ and $\boldsymbol{h}_2$ to generate two offsprings $\boldsymbol{h}_a^{\text{offspr}}$ and $\boldsymbol{h}_b^{\text{offspr}}$ to be included in $\boldsymbol{\Psi}_k^{\text{offspr}}$. The two offsprings are created by generating randomly a crossover point κ for which $1 < \kappa < |\mathcal{I}_{\mathbf{P}}|$. The two offsprings are then given by

$$\boldsymbol{h}_a^{\text{offspr}} = \left(h_1^{(1)}, \ldots, h_1^{(\kappa)}, h_2^{(\kappa+1)}, \ldots, h_2^{(|\mathcal{I}_{\mathbf{P}}|)}\right)$$

and

$$\boldsymbol{h}_b^{\text{offspr}} = \left(h_2^{(1)}, \ldots, h_2^{(\kappa)}, h_1^{(\kappa+1)}, \ldots, h_1^{(|\mathcal{I}_{\mathbf{P}}|)}\right).$$

After the crossover step, we proceed by generating randomly a subset $\boldsymbol{\Psi}'_k \subset \boldsymbol{\Psi}_k$ with $|\boldsymbol{\Psi}'_k| = \left|\boldsymbol{\Psi}_k^{\text{offspr}}\right|$. The members of this subset shall be replaced by the generated offsprings. The probability for a member $\boldsymbol{h} \in \boldsymbol{\Psi}_k$ to be included in $\boldsymbol{\Psi}'_k$ is directly proportional to its cost $J_{\boldsymbol{h}}$; that is, the probability of replacing a member is high if the member has a high control cost (i.e., low control quality). Finally, with a given probability, mutation is performed on each member of the set

$$\left(\boldsymbol{\Psi}_k \setminus \boldsymbol{\Psi}'_k\right) \cup \boldsymbol{\Psi}_k^{\text{offspr}}$$

to obtain the population $\boldsymbol{\Psi}_{k+1}$ to be evaluated in the next iteration. Mutation of a member

$$\boldsymbol{h} = (h^{(1)}, \ldots, h^{(|\mathcal{I}_{\mathbf{P}}|)})$$

is done by generating randomly an index $p \in \{1, \ldots, |\mathcal{I}_{\mathbf{P}}|\}$ and replacing $h^{(p)}$ with a randomly selected period in $\mathbf{H}_{\sigma(p)} \setminus \{h^{(p)}\}$.

Genetic algorithms have several parameters that are decided based on the optimization problem at hand. We have tuned the parameters of the genetic algorithm experimentally according to rules of thumb and typical

ranges for the parameters [Ree93, Mic96, Hol75]. The population size is constant (i.e., $|\mathbf{\Psi}_k| = |\mathbf{\Psi}_{k+1}|$) and is chosen to be

$$2\,|\mathcal{I}_{\mathbf{P}}| \max_{i \in \mathcal{I}_{\mathbf{P}}} |\mathbf{H}_i|\,.$$

The number of offsprings $\left|\mathbf{\Psi}_k^{\text{offspr}}\right|$ that are generated in each iteration is the closest even integer to $0.25\,|\mathbf{\Psi}_k|$, where 0.25 is the crossover rate. The mutation probability is 0.01. The period exploration terminates when the average cost

$$J_{\text{avg}} = \frac{1}{|\mathbf{\Psi}_k|} \sum_{\boldsymbol{h} \in \mathbf{\Psi}_k} J_{\boldsymbol{h}}$$

of the current population is sufficiently close to the cost

$$J_{\text{min}} = \min_{\boldsymbol{h} \in \mathbf{\Psi}_1 \cup \cdots \cup \mathbf{\Psi}_k} J_{\boldsymbol{h}}$$

of the current best solution; in our implementation and experiments, the period-exploration process is stopped when $J_{\text{avg}} < 1.05 J_{\text{min}}$. This indicates a deviation of less than 5 percent between the average cost of the current population and the cost of the best solution found in the optimization process.

4.4 Solution for Static Cyclic Scheduling

In this section, we start by defining a static cyclic schedule for a given set of applications. We proceed by deriving the sensor–actuator delays for a given schedule. Last, we present an approach to construct the task and message schedule. We shall consider that a period assignment

$$\boldsymbol{h} \in \prod_{p=1}^{|\mathcal{I}_{\mathbf{P}}|} \mathbf{H}_{\sigma_{\mathbf{P}}(p)}$$

is given. This means that the period h_i of each application $\Lambda_i \in \mathbf{\Lambda}$ is given ($i \in \mathcal{I}_{\mathbf{\Lambda}}$). This is also the assumption for the material presented in Section 4.5, which treats task and message scheduling for priority-based systems.

4.4.1 Definition of a Static Cyclic Schedule

For each node index $d \in \mathcal{I}_{\mathbf{N}}$, we define

$$\mathbf{\Theta}_d = \bigcup_{\tau_{ij} \in \mathsf{map}^*(N_d)} \left\{ \tau_{ij}^{(q)} : q = 1, \ldots, h_{\mathbf{\Lambda}}/h_i \right\}$$

to be the set of jobs that are released for execution on node N_d in the half-open time interval $[0, h_{\mathbf{\Lambda}})$. Note that map^* is defined in Section 3.2.3. Let us also define the set of message instances $\mathbf{\Theta}_{\text{bus}}$ that are communicated on the bus in the time interval $[0, h_{\mathbf{\Lambda}})$ as

$$\mathbf{\Theta}_{\text{bus}} = \bigcup_{\gamma_{ijk} \in \mathbf{\Gamma}_{\text{bus}}} \left\{ \gamma_{ijk}^{(q)} : q = 1, \ldots, h_{\mathbf{\Lambda}}/h_i \right\}.$$

A static cyclic schedule $\mathbf{\Omega}$ is a set of schedules

$$\mathbf{\Omega} = \left(\bigcup_{i \in \mathcal{I}_{\mathbf{N}}} \{\Omega_i\} \right) \cup \{\Omega_{\text{bus}}\}$$

for each computation node and the bus. For each $d \in \mathcal{I}_{\mathbf{N}}$, the schedule for node N_d is an injective function

$$\Omega_d : \mathbf{\Theta}_d \longrightarrow [0, h_{\mathbf{\Lambda}})$$

that gives the start time of each job. The bus schedule is an injective function

$$\Omega_{\text{bus}} : \mathbf{\Theta}_{\text{bus}} \longrightarrow [0, h_{\mathbf{\Lambda}})$$

that gives the start time of each message instance.

The schedule $\mathbf{\Omega}$ is executed regularly with a period equal to the hyper period $h_{\mathbf{\Lambda}}$ of the application set $\mathbf{\Lambda}$. Let

$$q' = 1 + \left((q-1) \bmod \frac{h_{\mathbf{\Lambda}}}{h_i} \right).$$

Then, the periodicity of the schedule means that, for each $\tau_{ij} \in \mathsf{map}^*(N_d)$, the start time of job $\tau_{ij}^{(q)}$ is

$$\left\lfloor (q-1) \Big/ \frac{h_{\mathbf{\Lambda}}}{h_i} \right\rfloor h_{\mathbf{\Lambda}} + \Omega_d \left(\tau_{ij}^{(q')} \right),$$

whereas, for each message $\gamma_{ijk} \in \mathbf{\Gamma}_{\text{bus}}$, the start time of instance $\gamma_{ijk}^{(q)}$ is

$$\left\lfloor (q-1) \Big/ \frac{h_{\mathbf{\Lambda}}}{h_i} \right\rfloor h_{\mathbf{\Lambda}} + \Omega_{\text{bus}} \left(\gamma_{ijk}^{(q')} \right).$$

We shall now introduce a set of constraints that a schedule $\mathbf{\Omega}$ must satisfy.

Periodic task releases The start time of any job $\tau_{ij}^{(q)} \in \mathbf{\Theta}_d$ must be after its release time. This means that

$$\Omega_d \left(\tau_{ij}^{(q)} \right) \geqslant (q-1) h_i$$

for $d \in \mathcal{I}_{\mathbf{N}}$, $i \in \mathcal{I}_{\mathbf{\Lambda}}$, $j \in \mathcal{I}_i$, and $q = 1, \ldots, h_{\mathbf{\Lambda}}/h_i$.

Data dependencies If a job needs data that are produced by other jobs, it can start executing only after that data is available. For each $i \in \mathcal{I}_{\mathbf{\Lambda}}$ and $\gamma_{ijk} = (\tau_{ij}, \tau_{ik}) \in \mathbf{\Gamma}_i$, let $N_c = \mathsf{map}(\tau_{ij})$ and $N_d = \mathsf{map}(\tau_{ik})$ be the computation nodes that host tasks τ_{ij} and τ_{ik}, respectively. If $N_c = N_d$ (i.e., the two tasks are executed on the same node), then

$$\Omega_c \left(\tau_{ij}^{(q)} \right) + c_{ij}^{\text{wc}} \leqslant \Omega_d \left(\tau_{ij}^{(q)} \right)$$

must hold. If the two tasks execute on different computation nodes, we require that the following two constraints hold:

$$\Omega_c \left(\tau_{ij}^{(q)} \right) + c_{ij}^{\text{wc}} \leqslant \Omega_{\text{bus}} \left(\gamma_{ijk}^{(q)} \right)$$
$$\Omega_{\text{bus}} \left(\gamma_{ijk}^{(q)} \right) + c_{ijk} \leqslant \Omega_d \left(\tau_{ik}^{(q)} \right)$$

Resource constraints On each computation node, at most one job can execute at any given time instant. This means that, for each $d \in \mathcal{I}_{\mathbf{N}}$, there must exist a bijection

$$\sigma_d : \{1, \ldots, |\mathbf{\Theta}_d|\} \longrightarrow \mathbf{\Theta}_d$$

such that, for each job $\tau_{ij}^{(q)} = \sigma_d(p)$ with $p \in \{1, \ldots, |\mathbf{\Theta}_d| - 1\}$,

$$\Omega_d(\sigma_d(p)) + c_{ij}^{\text{wc}} \leqslant \Omega_d(\sigma_d(p+1))$$

holds. For the last job $\tau_{ij}^{(q)} = \sigma(|\mathbf{\Theta}_d|)$, we require

$$\Omega_d(\sigma(|\mathbf{\Theta}_d|)) + c_{ij}^{\text{wc}} \leqslant h_{\mathbf{\Lambda}}.$$

The bijection σ_d gives the order of job executions on node N_d.

The resource constraint for the bus is formulated in a similar manner as for the computation nodes. For the bus schedule Ω_{bus}, there must exist a bijection

$$\sigma_{\text{bus}} : \{1, \ldots, |\mathbf{\Theta}_{\text{bus}}|\} \longrightarrow \mathbf{\Theta}_{\text{bus}}$$

such that, for each instance $\gamma_{ijk}^{(q)} = \sigma_{\text{bus}}(p)$ with $p \in \{1, \ldots, |\mathbf{\Theta}_{\text{bus}}| - 1\}$,

$$\Omega_{\text{bus}}(\sigma_{\text{bus}}(p)) + c_{ijk} \leqslant \Omega_{\text{bus}}(\sigma_{\text{bus}}(p+1))$$

holds. The bijection σ_{bus} gives the communication order on the bus.

Timing constraints If a task τ_{ij} on node $N_d = \mathsf{map}(\tau_{ij})$ has a hard relative deadline D_{ij}, the timing constraint

$$\Omega_d\left(\tau_{ij}^{(q)}\right) + c_{ij}^{\text{wc}} \leqslant (q-1)h_i + D_{ij}$$

must hold for all $q \in \{1, \ldots, h_{\mathbf{\Lambda}}/h_i\}$.

4.4.2 Sensor–Actuator Delay

For a given a schedule $\mathbf{\Omega}$, let us now discuss the computation of the sensor–actuator delay Δ_i^{sa} of each control application Λ_i ($i \in \mathcal{I}_{\mathbf{P}}$). For a control application Λ_i of plant P_i, let us denote the actuator task of Λ_i with τ_{ia} ($a \in \mathcal{I}_i$). The probability function of Δ_i^{sa} is determined by the start times of the actuator task τ_{ia} and the stochastic execution time c_{ia} with given probability function $\xi_{c_{ia}}$ (these execution times are introduced in Section 3.2.3). Let $N_d = \mathsf{map}(\tau_{ia})$ denote the computation node for the actuator task of controller Λ_i. For $q = 1, \ldots, h_{\mathbf{\Lambda}}/h_i$, in the q^{th} instance of Λ_i, the sensors are read at time $(q-1)h_i$. The start time of the corresponding actuation is $\Omega_d\left(\tau_{ia}^{(q)}\right)$. Letting

$$\phi_{ia}^{(q)} = \Omega_d\left(\tau_{ia}^{(q)}\right) - (q-1)h_i,$$

we observe that the sampling–actuation delay in the q^{th} controller instance is distributed between a minimum and maximum delay of $\phi_{ia}^{(q)} +$

c_{ia}^{bc} and $\phi_{ia}^{(q)} + c_{ia}^{\text{wc}}$, respectively. The probability function of the delay in the q^{th} instance is thus

$$\xi_{\Delta_i}^{\text{sa}(q)}(\delta) = \xi_{c_{ia}}\left(\delta - \phi_{ia}^{(q)}\right).$$

By considering all jobs in the schedule, we observe that the probability function of the delay Δ_i^{sa} is

$$\xi_{\Delta_i}^{\text{sa}}(\delta) = \frac{h_i}{h_{\mathbf{\Lambda}}} \sum_{q=1}^{h_{\mathbf{\Lambda}}/h_i} \xi_{\Delta_i}^{\text{sa}(q)}(\delta).$$

This information is used to synthesize a control law and, subsequently, to compute the control cost of a control application and its implementation.

4.4.3 Schedule Construction

Our goal is to find a schedule $\mathbf{\Omega}$ that minimizes the cost

$$\sum_{i \in \mathcal{I}_{\mathbf{P}}} J_i.$$

We use a constraint logic programming (CLP) formulation [AW07] of the scheduling problem. CLP solvers have been used successfully in the past for various scheduling problems [Kuc03]. We have formulated the constraints in Section 4.4.1 in the CLP framework of ECL^{i}PS^e [AW07]. The control cost J_i defined in Equation 3.3 (page 28) cannot be incorporated in the CLP framework. For this reason, we have considered an approximate cost to be used during the construction of the static cyclic schedule; the control cost J_i is, however, considered during the other design activities in our integrated control and scheduling framework.

As stated before, two timing parameters, characterizing the sensor–actuator delay, affect the control performance: the average delay and its variance. During the construction of the schedule, we minimize therefore the quadratic cost

$$\sum_{i \in \mathcal{I}_{\mathbf{P}}} \left(\alpha_i \mathrm{E}\left\{\Delta_i^{\text{sa}}\right\}^2 + \beta_i \mathrm{D}\left\{\Delta_i^{\text{sa}}\right\}^2\right),$$

where α_i and β_i are given parameters ($\mathrm{D}\{\cdot\}$ denotes the standard deviation of a stochastic variable). We have used $\alpha_i = \beta_i = 1$ for the experiments presented in Section 4.6.

The CLP formulation, comprising the constraints and optimization objective, are inputs to the CLP solver, which is based on branch-and-bound search and constraint propagation [AW07]. Because of computational complexity of finding optimal solutions to the scheduling problem, we cannot, for large problem sizes, afford a complete search of the set of possible schedules. Therefore, the CLP solver is configured for heuristic search based on the characteristics of the problem at hand. We have configured the CLP solver to use limited-discrepancy search [AW07], which is an incomplete branch-and-bound search, and to exclude solutions that only lead to less than 10 percent of the cost of the current best solution during the search process.

4.5 Solution for Priority-Based Scheduling

In this section, we consider preemptive scheduling of tasks and non-preemptive scheduling of messages, both based on fixed priorities that are determined at design time. The overall flowchart of the integrated control and scheduling solution for the case of priority-based scheduling is obtained by replacing the dashed box in Figure 4.4 on page 45 with the flowchart in Figure 4.5. Given from the outer loop in Figure 4.4 are the periods $\boldsymbol{h}$ of all applications. The goal is to minimize the overall cost by deciding task and message priorities, and by computing the control laws. In the outer loop in Figure 4.5, we explore and compute costs for several priority assignments. In the continuation of this section, we describe the computation flow that leads to the cost of a priority assignment. Last, we describe the priority-exploration approach.

4.5.1 Definition of Priority Assignment and Schedulability

Let us define a priority assignment as a set

$$\boldsymbol{\rho} = \left(\bigcup_{d \in \mathcal{I}_{\mathbf{N}}} \{\rho_d\} \right) \cup \{\rho_{\text{bus}}\},$$

where $\rho_d : \mathsf{map}^*(N_d) \longrightarrow \mathbb{N}$ and $\rho_{\text{bus}} : \mathbf{\Gamma}_{\text{bus}} \longrightarrow \mathbb{N}$ are injective functions that give the priorities of the tasks on node N_d ($d \in \mathcal{I}_{\mathbf{N}}$) and the priorities of the messages on the bus, respectively (a larger value indicates a higher priority). Let us recall that $\mathsf{map}^*(N_d)$ denotes the set of

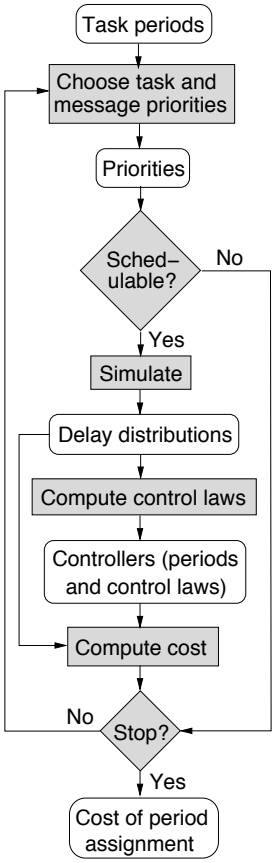

Figure 4.5: Integrated control-law synthesis and assignment of task and message priorities. Schedulability is verified by formal timing analysis, whereas the delay distributions for control synthesis and cost computation are obtained through simulation.

tasks that are mapped to node N_d (Section 3.2.3). Given a priority assignment $\boldsymbol{\rho}$, we are interested in the temporal behavior of the system.

For a given period and priority assignment, it is needed to perform timing analysis to verify schedulability. We have used the holistic timing analysis method by Pop et al. [PPEP08, PPE+08] to obtain the worst-case response time of each task. The system is schedulable if all worst-case response times exist—this means that no task has an unbounded response time—and they are smaller than or equal to any imposed task deadline.

Note that, in this step, we check the satisfaction of the imposed timing constraints by formal response-time analysis. In the next step, we synthesize the control laws and assess their quality based on delay distributions obtained with simulation; we have used our simulation framework for distributed real-time systems [SREP08].

4.5.2 Estimation of Sensor–Actuator Delay

For a given assignment of periods and priorities, we use a simulation environment for distributed real-time systems [SREP08] to obtain an approximation $\widehat{\Delta}_i^{\text{sa}}$ of each sensor–actuator delay Δ_i^{sa}. The probability function of the discrete stochastic variable $\widehat{\Delta}_i^{\text{sa}}$, which approximates Δ_i^{sa}, is denoted $\widehat{\xi}_{\Delta_i}^{\text{sa}}$ and is an output of the simulation. During simulation, we compute the average sensor–actuator delays regularly with the period $h_{\mathbf{\Lambda}}$. Let $\mathbf{\Delta}_i^{(k)}$ denote the set of sensor–actuator delays for application Λ_i ($i \in \mathcal{I}_{\mathbf{P}}$) that, during simulation, occur in the time interval $[0, kh_{\mathbf{\Lambda}}]$ ($k > 0$). Further, let

$$\eta_{\mathbf{\Delta}_i}^{(k)} : \mathbf{\Delta}_i^{(k)} \longrightarrow \mathbb{N} \setminus \{0\}$$

be a function for which $\eta_{\mathbf{\Delta}_i}^{(k)}(\delta)$ is the number of times the delay δ occurred in the time interval $[0, kh_{\mathbf{\Lambda}}]$. The total number of delays for Λ_i in the simulated time interval is

$$\eta_{i,\text{tot}}^{(k)} = \sum_{\delta \in \mathbf{\Delta}_i^{(k)}} \eta_{\mathbf{\Delta}_i}^{(k)}(\delta).$$

At times $kh_{\mathbf{\Lambda}}$ during simulation, we compute an average delay

$$\delta_{i,\text{avg}}^{(k)} = \frac{1}{\eta_{i,\text{tot}}^{(k)}} \sum_{\delta \in \mathbf{\Delta}_i^{(k)}} \delta \cdot \eta_{\mathbf{\Delta}_i}^{(k)}(\delta)$$

for each control application Λ_i. The simulation is terminated at the first simulated time instant $k'h_{\mathbf{\Lambda}}$ ($k' > 1$) where the condition

$$\max_{i \in \mathcal{I}_{\mathbf{P}}} \left(\frac{\left| \delta_{i,\text{avg}}^{(k')} - \delta_{i,\text{avg}}^{(k'-1)} \right|}{\delta_{i,\text{avg}}^{(k'-1)}} \right) < \zeta_{\text{sim}}$$

is satisfied. The parameter ζ_{sim} is given and is used to regulate the stopping condition and the runtime of the simulation. We have tuned this

parameter to $\zeta_{\text{sim}} = 0.05$ experimentally. This means that the simulation is stopped when the average delay has changed with less than 5 percent. After the simulated time $k' h_{\mathbf{\Lambda}}$, the approximation of each Δ_i^{sa} is given by the probability function

$$\widehat{\xi}_{\Delta_i}^{\text{sa}}(\delta) = \eta_{\mathbf{\Delta}_i}^{(k')}(\delta) \Big/ \eta_{i,\text{tot}}^{(k')}.$$

Given the approximate delay $\widehat{\Delta}_i^{\text{sa}}$ for each control loop, we proceed by computing the control law $\boldsymbol{u}_i$ for a constant delay of $\mathrm{E}\left\{\widehat{\Delta}_i^{\text{sa}}\right\}$. Finally, the control cost J_i is computed, using the approximate probability function $\widehat{\xi}_{\Delta_i}^{\text{sa}}$. The cost of the priority assignment $\boldsymbol{\rho}$, given the periods $\boldsymbol{h}$, is

$$J_{\boldsymbol{\rho}|\boldsymbol{h}} = \sum_{i \in \mathcal{I}_{\mathbf{P}}} J_i.$$

4.5.3 Optimization of Priorities

The outer loop in Figure 4.5, which explores different priority assignments, is based on a genetic algorithm, similar to the period exploration in Section 4.3. We generate a population randomly and, in the iterations, we evaluate the cost of priority assignments that are generated with the crossover and mutation operators [Ree93, Mic96, Hol75]. The population size is constant and equal to the number of tasks and messages in the system. The crossover rate is 0.25 and the mutation probability is 0.01. The exploration of priority assignments terminates when $J_{\text{avg}} < 1.1 J_{\text{min}}$, where J_{avg} is the average cost of the current population and J_{min} is the cost of the current best priority assignment. After the priority optimization has terminated, the cost that is associated to the period assignment $\boldsymbol{h}$ is $J_{\boldsymbol{h}} = J_{\text{min}}$.

4.6 Experimental Results

We have run experiments to study the performance improvements that can be achieved with our control–scheduling co-design technique. Our method is compared to a straightforward approach, which is described in Section 4.6.1. The experimental results are presented and discussed in Section 4.6.2.

4.6.1 Straightforward Design Approach

We have defined a straightforward approach as a baseline for comparison against our proposed optimization technique. The objective is to compare the integrated control and scheduling method with an approach in which control design and implementation are separated. In the straightforward design approach, each controller Λ_i is synthesized for a sampling period equal to the average of the set of available periods $\mathbf{H}_i$ and for a delay equal to the cumulative execution time of the control application. For the implementation, the following steps are performed:

1. Assign the period of each control application Λ_i to the smallest period in the set of available periods $\mathbf{H}_i$.

2. Schedule the system (depends on the scheduling policy).
   (a) Static cyclic scheduling: Schedule the tasks and messages, respectively, for execution and communication as early as possible, taking into account the schedule constraints. In this step, start times of tasks and messages are chosen to be as small as possible.
   (b) Priority-based scheduling: Assign priorities rate monotonically. Priorities of tasks and messages are inversely proportional to the periods; a task with a small period has a higher priority than a task with a large period, running on the same computation node. Each computation node is treated separately during period assignment. The messages on the bus are assigned priorities in a similar manner.

3. If the system is not schedulable, perform the following steps:
   (a) If, for each application, the current period is the largest in the original set of available periods, the straightforward approach terminates. In this case, the straightforward design method could not find a solution.
   (b) For each of the control applications with highest utilization, if the current period is not the largest in the original set of available periods, then remove it from $\mathbf{H}_i$. Go to Step 1.

Thus, the straightforward approach takes the designed controllers and produces a schedulable implementation for as small controller periods as possible, but does not further consider the control quality.

To investigate the efficiency of period exploration and appropriate scheduling separately, we defined two semi-straightforward approaches. For the first approach—straightforward period assignment—periods are assigned as in the straightforward approach. The scheduling, however, is performed according to our proposed approach, which integrates control-law synthesis. For the second approach—straightforward scheduling—periods are chosen according to our proposed genetic algorithm-based approach, but the scheduling is done according to Step 2 in the straightforward approach; the control laws in this case are synthesized according to the straightforward approach.

4.6.2 Setup and Results

For the evaluation, we created 130 test cases with varying number of plants to be controlled. We used test cases with 2 to 9 plants that were chosen randomly from a set of inverted pendulums, ball and beam processes, DC servos, and harmonic oscillators [ÅW97]. For each plant, we generated a control application with 3 to 5 tasks with data dependencies. Thus, the number of tasks for the generated test cases varies from 6 to 45. The tasks were mapped randomly to platforms consisting of 2 to 7 computation nodes. For each controller, we generated 6 available periods (i.e., $|\mathbf{H}_i| = 6$). Based on the average values of these periods, we generated the execution and communication times of the tasks and messages to achieve maximum node and bus utilization between 40 and 80 percent. For the tasks, we considered uniform execution-time distributions.

For each test case, the proposed integrated design approach has been executed for both static cyclic scheduling and priority-based scheduling of the tasks and messages. In the experiments, we have considered a TTP bus [Kop97] for the static cyclic scheduling case and a CAN bus [Bos91] for the priority-based scheduling case. Our implementation also supports the FlexRay protocol [Fle05] through optimization of frame identifiers for messages [SEPC11]. We have also run the two semi-straightforward approaches. For each of the three approaches (the integrated approach and the two semi-straightforward approaches), we obtained a final overall cost J_{approach}. We were interested in the relative cost improvements

$$\frac{J_{\text{SF}} - J_{\text{approach}}}{J_{\text{SF}}},$$

where J_{SF} is the cost obtained with the straightforward approach. This

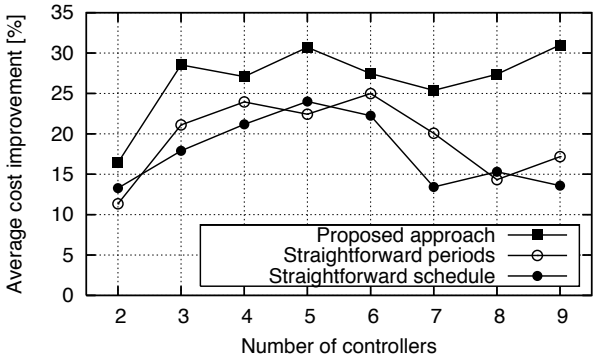

Figure 4.6: Improvements for static cyclic scheduling. Our integrated approach and the two semi-straightforward approaches are compared to the straightforward design approach in which control design and system scheduling are separated.

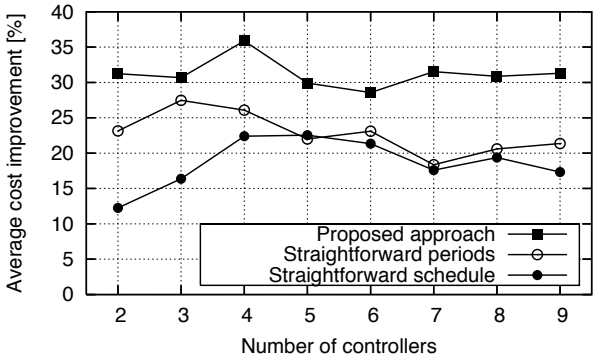

Figure 4.7: Improvements for priority-based scheduling. The same comparison as in Figure 4.6 is made but for systems with priority-based scheduling and communication.

improvement factor characterizes the quality improvement achieved with a certain design approach compared to the straightforward method in Section 4.6.1.

The average quality improvements for static cyclic scheduling and priority-based scheduling are depicted in Figures 4.6 and 4.7, respectively. In each of the two figures, the vertical axis is the average cost improvement in percent for all test cases, with the number of controllers

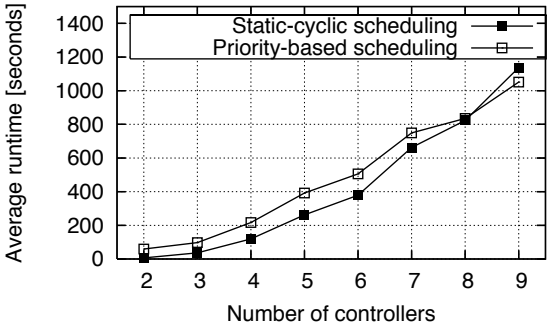

Figure 4.8: Runtime for the optimization. The runtime of the integrated design approach is shown for various problem sizes. Optimization of systems with static cyclic scheduling and priority-based scheduling can be performed with comparable runtime.

given on the horizontal axis. In Figure 4.6, for example, the average relative cost improvements of the straightforward period-exploration and straightforward scheduling for 9 controllers are 17.2 and 13.6 percent, respectively. For the same case, the integrated synthesis and scheduling approach gives a quality improvement of 31.0 percent. For a small number of controllers, we note that the semi-straightforward approaches give improvements close to the improvements by the integrated approach. For a larger number of controllers, however, the design space becomes larger, and thus the semi-straightforward approaches perform worse. The results show that it is important to combine period exploration with integrated scheduling and control-law synthesis to obtain high-quality solutions.

We have measured the runtime for the proposed integrated approach; all experiments were run on a PC with the CPU frequency 2.2 GHz, 8 Gb of RAM, and running Linux. In Figure 4.8, we show, for both static and priority-based scheduling, the average runtime in seconds as a function of the number of controllers. It can be seen that the complex optimization, involving period assignment, scheduling (priority assignment), controller design, and cost computation, can be run in our framework in less than 19 minutes for large systems of 9 controllers and 45 tasks.

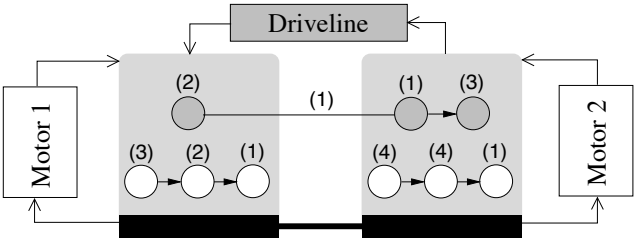

Figure 4.9: Benchmark comprising a driveline model and two motors. Several tasks implement control functionality and execute on a two-node platform.

4.7 Automotive Example

To further evaluate the performance of our optimization tool, and to validate its practicality, we performed a case study on a benchmark with automotive control applications. We performed the optimization and synthesis for a benchmark consisting of a driveline model and two motor models that are controlled by a system with two computation nodes. We have considered static cyclic scheduling of the system.

Figure 4.9 shows an overview of the considered benchmark. The driveline is controlled through sensors and actuators with interfaces to both computation nodes. Two motors are stabilized by two control applications, each running on one computation node. The driveline-control tasks are depicted with grey circles, whereas the other two controllers are shown with white circles. The worst-case execution times of the tasks and communication times of the messages on the bus are indicated in parentheses. We considered the best-case execution time of each task to be a quarter of its worst-case (the execution-time distributions are uniform). We have used sampling periods of 10, 20, 30, or 40 milliseconds.

The driveline model and the details of its parameters are provided by Kiencke and Nielsen [KN05]. The model contains states that represent the angle of the engine flywheel, the engine speed, the wheel position, and the wheel speed. Sensors are available for measurements of the wheel speed. The control problem at hand for the driveline is to control the wheel speed by applying and actuating engine torque. The motor model [ÅW97] describes the dynamics of the velocity and position of the motor shaft. The input to the motor is a voltage that is used to control the shaft position, which is the only variable that can be measured with the

Table 4.1: Control costs for the two motors and the driveline. Our integrated approach is compared to the straightforward design.

	Integrated	Straightforward	Relative improvement
Motor 1	0.63	1.13	44.2%
Motor 2	0.72	0.99	27.3%
Driveline	1.51	2.01	25.0%
Total	2.86	4.13	30.6%

available sensors.

Our optimization tool and the straightforward design approach both found solutions with the assigned periods 10, 20, and 10 milliseconds, respectively, for the two motors and the driveline. The individual and total costs are shown in Table 4.1. Let us now focus on the driveline controller and discuss the characteristics of the obtained solution. With the straightforward approach, the control law is synthesized for the average period of 25 milliseconds of the available sampling periods (10, 20, 30, and 40 milliseconds). The controller compensates for a delay of 7 milliseconds, which is the cumulative delay of the grey control application in Figure 4.9. With the straightforward design approach, we obtained a schedule for which the delay in the driveline-control loop varies between 5 and 8 milliseconds. With our proposed integrated approach, the delay varies between 6 and 7 milliseconds, and, in addition, the control laws are synthesized to compensate for the delays in the produced schedules. This results in the significant performance improvements presented in Table 4.1.

4.8 Summary and Discussion

Scheduling and communication in embedded control systems lead to complex temporal behavior and influence the control quality of the running applications. To address this problem at design time, we have developed a control–scheduling co-design approach that integrates task and message scheduling with controller design (control-period selection and control-law synthesis). Our solution is to consider a control-performance metric as an optimization objective for the scheduling of tasks and mes-

sages, as well as to integrate information regarding the system timing in the traditional control-synthesis framework. The experimental results and a case study show that such an integrated design flow is essential to achieve high control performance for distributed embedded control systems.

5
Synthesis of Multi-Mode Control Systems

EMBEDDED computing systems that control several physical plants may switch between alternative modes at runtime either as a response to external events or at predetermined moments in time. In an operation mode, the system controls a subset of the plants by executing the corresponding control applications according to precomputed solutions (control laws, periods, and schedules). In the context of this chapter, a mode change means that some of the running control loops are deactivated, some new controllers are activated, or both. This leads to a change in the execution and communication demand of the system. Consequently, new design solutions must be used at runtime to achieve best possible control performance by an efficient use of the computation and communication resources.

In this chapter, we shall extend the framework presented in Chapter 4 towards synthesis and optimization of multi-mode control systems. The number of possible modes is exponential in the number of control loops. This design-space complexity leads to unaffordable synthesis time and memory requirements to store the synthesized controllers and schedules. With the objective of optimizing the overall control performance,

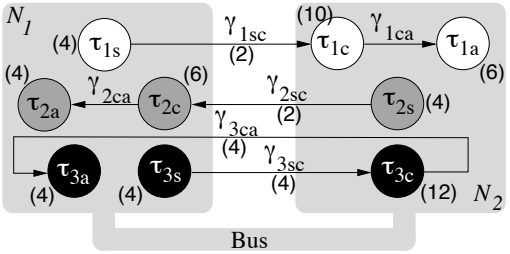

Figure 5.1: Three control applications running on a platform with two nodes. Execution and communication times are given inside parentheses.

we shall address these problems by a limited exploration of the set of modes and a careful selection of the produced schedules and controllers to store in memory.

This chapter is organized as follows. Section 5.1 presents models and formalisms related to multi-mode systems. We discuss a motivational example in Section 5.2, leading to a problem statement in Section 5.3. The synthesis approach for multi-mode control systems is presented in Section 5.4, and its experimental evaluation is discussed in Section 5.5. In Section 5.6, we discuss and summarize the contributions of this chapter.

5.1 Multi-Mode Systems

We shall consider the same system and notation as in Chapter 3: A set of plants $\mathbf{P}$ are controlled by a set of applications $\mathbf{\Lambda}$ that execute on a platform with a set of nodes $\mathbf{N}$. Application Λ_i implements the controller for plant P_i ($i \in \mathcal{I}_\mathbf{P}$).

Figure 5.1 shows an example with two nodes $\mathbf{N} = \{N_1, N_2\}$ that are connected to a bus. We shall consider this example system throughout this chapter. Three applications $\mathbf{\Lambda} = \{\Lambda_1, \Lambda_2, \Lambda_3\}$ are mapped to the nodes and control the three pendulums P_1, P_2, and P_3. For this example, the task set of application Λ_i is $\mathbf{T}_i = \{\tau_{is}, \tau_{ic}, \tau_{ia}\}$ with index set $\mathcal{I}_i = \{s, c, a\}$ ($\mathcal{I}_\mathbf{P} = \{1, 2, 3\}$). The data dependencies are given by the edges

$$\mathbf{\Gamma}_i = \{\gamma_{isc} = (\tau_{is}, \tau_{ic}), \gamma_{ica} = (\tau_{ic}, \tau_{ia})\}\,.$$

The sensor task is τ_{is}, the controller task is τ_{ic}, and the actuator task is τ_{ia}.

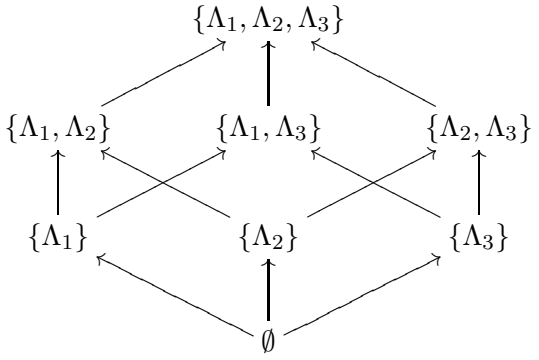

Figure 5.2: Hasse diagram of modes. Eight possible modes for the system in Figure 5.1 are shown. The vertices are operation modes and the edges order modes by inclusion.

We shall consider that the system can run in different operation modes, where each mode is characterized by a set of running applications. A mode is therefore defined as a subset $\mathbf{M} \subseteq \mathbf{\Lambda}$. For a mode $\mathbf{M} \neq \emptyset$, let us denote the index set of $\mathbf{M}$ by $\mathcal{I}_{\mathbf{M}}$. When the system is operating in mode $\mathbf{M}$, only the applications $\Lambda_i \in \mathbf{M}$ ($i \in \mathcal{I}_{\mathbf{M}}$) are executing. The complete set of modes is the power set $\mathcal{M} = 2^{\mathbf{\Lambda}}$ and its cardinality is $|\mathcal{M}| = 2^{|\mathbf{\Lambda}|}$. The set of modes $\mathcal{M}$ is a partially ordered set under the subset relation. For our example with the three applications in Figure 5.1, we illustrate this partial order with the Hasse diagram [Gri04] in Figure 5.2 for the case

$$\mathcal{M} = 2^{\mathbf{\Lambda}} = 2^{\{\Lambda_1, \Lambda_2, \Lambda_3\}}.$$

Let us consider two modes $\mathbf{M}, \mathbf{M}' \in \mathcal{M}$ with $\mathbf{M}' \subset \mathbf{M}$. Mode $\mathbf{M}'$ is called a *submode* of $\mathbf{M}$. Similarly, mode $\mathbf{M}$ is a *supermode* of mode $\mathbf{M}'$. We define the set of submodes of $\mathbf{M} \in \mathcal{M}$ as

$$\underline{\mathcal{M}}(\mathbf{M}) = \left\{ \mathbf{M}' \in \mathcal{M} : \mathbf{M}' \subset \mathbf{M} \right\}.$$

Defined similarly, the set of supermodes of $\mathbf{M}$ is denoted $\overline{\mathcal{M}}(\mathbf{M})$. The idle mode is the empty set $\emptyset$, which indicates that the system is inactive, whereas mode $\mathbf{\Lambda}$ indicates that all applications are running. It can be the case that certain modes do not occur at runtime—for example, because certain plants are never controlled concurrently. We shall refer to modes that may occur during execution as *functional* modes. The other modes

do not occur at runtime and are referred to as *virtual* modes. Let us therefore introduce the set of functional modes $\mathcal{M}^{\text{func}} \subseteq \mathcal{M}$ that can occur during execution. The virtual modes $\mathcal{M}^{\text{virt}} = \mathcal{M} \setminus \mathcal{M}^{\text{func}}$ do not occur at runtime. In a given mode $\mathbf{M} \in \mathcal{M}$, an application $\Lambda_i \in \mathbf{M}$ ($i \in \mathcal{I}_{\mathbf{M}}$) releases jobs for execution periodically with the period $h_i^{\mathbf{M}}$. We define the hyper period $h_{\mathbf{M}}$ of $\mathbf{M}$ as the least common multiple of the periods

$$\{h_i^{\mathbf{M}}\}_{i \in \mathcal{I}_{\mathbf{M}}}.$$

An *implementation* of a mode $\mathbf{M} \in \mathcal{M}$ comprises the period $h_i^{\mathbf{M}}$ and control law $\boldsymbol{u}_i^{\mathbf{M}}$ of each control application $\Lambda_i \in \mathbf{M}$ ($i \in \mathcal{I}_{\mathbf{M}}$), and the schedule (or priorities) for the tasks and messages in that mode. We denote with $\mathsf{mem}_d^{\mathbf{M}}$ the memory consumption on node $N_d \in \mathbf{N}$ ($d \in \mathcal{I}_{\mathbf{N}}$) of the implementation of $\mathbf{M}$. An implementation of a mode can serve as an implementation of all its submodes. This means that the system can run in a submode $\mathbf{M}' \subset \mathbf{M}$ with the same controllers and schedule as for mode $\mathbf{M}$. This is performed, for example, by not running the applications in $\mathbf{M} \setminus \mathbf{M}'$ or by not writing to their outputs. To achieve better performance in mode $\mathbf{M}'$, however, a customized set of controllers and schedule for submode $\mathbf{M}'$ can exploit the available computation and communication resources that are not used by the other applications $\mathbf{M} \setminus \mathbf{M}'$. To have a correct implementation of the whole multi-mode system, for each functional mode $\mathbf{M} \in \mathcal{M}^{\text{func}}$, there must exist an implementation in memory, or there must exist an implementation of at least one of its supermodes. Let us denote the set of implemented modes with $\mathcal{M}^{\text{impl}}$. We know that $\mathcal{M}^{\text{impl}} \subseteq \mathcal{M} \setminus \{\emptyset\}$. Thus, only the modes in $\mathcal{M}^{\text{impl}}$ have implementations stored in memory. In Section 5.4, we shall describe how to select the modes to be implemented by the system with the objective to synthesize a functionally correct system with optimized control quality.

We use the framework presented in Chapter 4 to obtain an implementation of a certain mode $\mathbf{M} \in \mathcal{M} \setminus \{\emptyset\}$. After this optimization process, we obtain a design solution with a cost

$$J^{\mathbf{M}} = \sum_{i \in \mathcal{I}_{\mathbf{M}}} J_i^{\mathbf{M}}, \tag{5.1}$$

where $J_i^{\mathbf{M}}$ is the cost of controller Λ_i when executing in mode $\mathbf{M}$ with the synthesized implementation. The produced mode implementation comprises the periods $h_i^{\mathbf{M}}$, control laws $\boldsymbol{u}_i^{\mathbf{M}}$, and system schedule (sched-

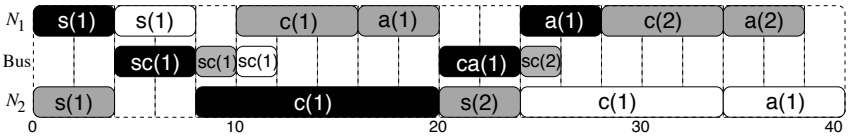

Figure 5.3: Schedule for mode $\{\Lambda_1, \Lambda_2, \Lambda_3\}$ with periods $h_1 = 40$, $h_2 = 20$, and $h_3 = 40$. The schedule, periods, and control laws are optimized for the situation in which all three plants are controlled.

ule table or priorities). As an output, we also obtain the memory consumption $\mathsf{mem}_d^{\mathbf{M}} \in \mathbb{N}$ of the implementation on each computation node $N_d \in \mathbf{N}$ ($d \in \mathcal{I}_{\mathbf{N}}$).

5.2 Motivational Example

In this section, we shall highlight the two problems that are addressed in this chapter—the time complexity of the synthesis of embedded multi-mode control systems and the memory complexity of the storage of produced mode implementations. Let us consider our example in Section 5.1 with three applications $\mathbf{\Lambda} = \{\Lambda_1, \Lambda_2, \Lambda_3\}$ that control three inverted pendulums $\mathbf{P} = \{P_1, P_2, P_3\}$. We shall use static cyclic scheduling in our examples; the conclusions, however, are general and valid also for priority-based scheduling and communication.

By using our framework for controller scheduling and synthesis in Chapter 4, we synthesized an implementation of mode $\mathbf{M}_{123} = \mathbf{\Lambda}$ with the periods $h_1^{\mathbf{M}_{123}} = 40$, $h_2^{\mathbf{M}_{123}} = 20$, and $h_3^{\mathbf{M}_{123}} = 40$ and the schedule in Figure 5.3. In Figure 5.1, the execution times (constant in this example) and communication times for the tasks and messages are given in milliseconds in parentheses. All times are given in milliseconds in the discussion that follows. The schedule is constructed for mode $\mathbf{M} = \mathbf{\Lambda}$—the mode in which all three applications are running—and for the periods $h_1^{\mathbf{M}} = 40$, $h_2^{\mathbf{M}} = 20$, and $h_3^{\mathbf{M}} = 40$. The period of the schedule is $h_{\mathbf{M}} = 40$, which is the hyper period of $\mathbf{M}$. Considering the periods of the applications, the jobs to be scheduled on node N_1 are $\tau_{1s}^{(1)}$, $\tau_{2c}^{(1)}$, $\tau_{2c}^{(2)}$, $\tau_{2a}^{(1)}$, $\tau_{2a}^{(2)}$, $\tau_{3s}^{(1)}$, and $\tau_{3a}^{(1)}$. The jobs on node N_2 are $\tau_{1c}^{(1)}$, $\tau_{1a}^{(1)}$, $\tau_{2s}^{(1)}$, $\tau_{2s}^{(2)}$, and $\tau_{3c}^{(1)}$. The message transmissions on the bus are $\gamma_{1sc}^{(1)}$, $\gamma_{2sc}^{(1)}$, $\gamma_{2sc}^{(2)}$, $\gamma_{3sc}^{(1)}$, and $\gamma_{3ca}^{(1)}$. The schedule in Figure 5.3 is shown with three rows for

Table 5.1: Individual control costs when running the system in Figure 5.1 in different modes. Each row shows an operation mode and the corresponding control costs for the control applications in that mode.

Mode $\mathbf{M}$	$J_1^{\mathbf{M}}$	$J_2^{\mathbf{M}}$	$J_3^{\mathbf{M}}$
$\{\Lambda_1, \Lambda_2, \Lambda_3\}$	2.98	1.19	2.24
$\{\Lambda_1, \Lambda_2\}$	1.60	1.42	-
$\{\Lambda_1, \Lambda_3\}$	1.60	-	1.95
$\{\Lambda_2, \Lambda_3\}$	-	1.36	1.95
$\{\Lambda_1\}$	1.60	-	-
$\{\Lambda_2\}$	-	1.14	-
$\{\Lambda_3\}$	-	-	1.87

node N_1, the bus, and node N_2, respectively. The small boxes depict task executions and message transmissions. The white, grey, and black boxes show the execution of applications Λ_1, Λ_2, and Λ_3, respectively. Each box is labeled with an index that indicates a task or message, and with a number that specifies the job or message instance. For example, the black box labeled $c(1)$ shows that the execution of job $\tau_{3c}^{(1)}$ on node N_2 starts at time 8 and completes at time 20. The grey box labeled $sc(2)$ shows the second message between the sensor and controller task of Λ_2. The message is scheduled for transmission at time 24. Note that the schedule executes periodically every 40 milliseconds.

The outputs of the plants are sampled periodically without jitter (e.g., by some mechanism that stores the sampled data in buffers). The actuations of Λ_1 and Λ_3 finish at times 40 and 28, respectively. Because the schedule is periodic, the delay from sampling to actuation is 40 in each instance of Λ_1 (i.e., the delay is constant). Similarly, application Λ_3 has the constant sampling–actuation delay 28. Two instances are scheduled for application Λ_2. The first actuation finishes at time 20, whereas the second actuation finishes at time 38. By considering the sampling period 20, we note that the sampling–actuation delay of Λ_2 during periodic execution of the schedule is either 20 or 18. With this implementation, we obtained the individual control costs $J_1^{\mathbf{M}_{123}} = 2.98$, $J_2^{\mathbf{M}_{123}} = 1.19$, and $J_3^{\mathbf{M}_{123}} = 2.24$. Table 5.1 includes the individual controller costs of the modes considered in this section.

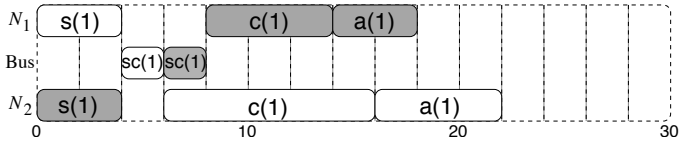

Figure 5.4: Schedule for mode $\{\Lambda_1, \Lambda_2\}$ with periods $h_1 = 30$ and $h_2 = 30$. This design solution gives better control performance for the case in which only plants P_1 and P_2 are controlled, compared to the situation in which the solution in Figure 5.3 is used.

5.2.1 Quality Improvement

Let us now consider mode $\mathbf{M}_{12} = \{\Lambda_1, \Lambda_2\}$ in which Λ_3 is not executing. The system can operate in the new mode by using the schedule and control laws from mode $\mathbf{M}_{123}$. This can be done by not writing to the actuators of Λ_3 or by omitting the executions and communications related to Λ_3. By using the implementation of mode $\mathbf{M}_{123}$, the overall control cost of mode $\mathbf{M}_{12}$ is $J_1^{\mathbf{M}_{123}} + J_2^{\mathbf{M}_{123}} = 2.98 + 1.19 = 4.17$. This cost can be reduced because, compared to mode $\mathbf{M}_{123}$, there is now more computation and communication power available for applications Λ_1 and Λ_2. Thus, it is worth to investigate whether a better implementation can be produced for mode $\mathbf{M}_{12}$—for example, an implementation with reduced periods and delays. By running the synthesis of $\mathbf{M}_{12}$, we obtained an implementation with the periods $h_1^{\mathbf{M}_{12}} = h_2^{\mathbf{M}_{12}} = 30$ and the schedule in Figure 5.4. Note from the new schedule that both the period and delay of Λ_1 have been reduced. The costs of Λ_1 and Λ_2 with the new implementation are $J_1^{\mathbf{M}_{12}} = 1.60$ and $J_2^{\mathbf{M}_{12}} = 1.42$ (Table 5.1). The cost of Λ_1 is reduced significantly as a result of the reduction in its sampling period and delay. The sampling period of Λ_2 is increased, which results in a small increase in the cost of Λ_2. This cost increase is accepted because it makes possible a significant quality improvement of Λ_1, which leads to a cost reduction for mode $\mathbf{M}_{12}$ from 4.17 to 3.02.

To achieve best possible control quality, implementations of all functional modes have to be produced at design time. However, the number of modes is exponential in the number of control loops that run on the platform. Thus, even if some modes are functionally excluded, implementations of all possible functional modes cannot be produced in affordable time (except cases with small number of functional modes or

systems with small number of control loops). The selection of the actual modes to synthesize is thus of critical importance for the overall control quality of most systems. Let us consider that we can only afford the synthesis time of three modes. Modes $\mathbf{M}_{123}$ and $\mathbf{M}_{12}$ have already been discussed in this section. Considering the third mode to be $\mathbf{M}_{13}$, the synthesis resulted in the costs $J_1^{\mathbf{M}_{13}} = 1.60$ and $J_3^{\mathbf{M}_{13}} = 1.95$ (Table 5.1). When using the implementation of $\mathbf{M}_{123}$, the costs of Λ_1 and Λ_3 are 2.98 and 2.24, respectively. The customized implementation of mode $\mathbf{M}_{13}$ gives a significant improvement in control performance, compared to when using the implementation of $\mathbf{M}_{123}$ to operate the system in mode $\mathbf{M}_{13}$.

At runtime, the system has implementations of the modes $\mathcal{M}^{\text{impl}} = \{\mathbf{M}_{123}, \mathbf{M}_{12}, \mathbf{M}_{13}\}$ in memory. For modes that are not implemented, the system chooses an implementation at runtime based on the three available implementations. For example, mode $\mathbf{M}_2 = \{\Lambda_2\}$ does not have an implementation but can run with the implementation of either $\mathbf{M}_{12}$ or $\mathbf{M}_{123}$. The cost of Λ_2 when running with the implementation of $\mathbf{M}_{12}$ is $J_2^{\mathbf{M}_{12}} = 1.42$, whereas it is $J_2^{\mathbf{M}_{123}} = 1.19$ for the implementation of $\mathbf{M}_{123}$. Thus, at runtime, the implementation of $\mathbf{M}_{123}$ is chosen to operate the system in mode $\mathbf{M}_2$.

5.2.2 Memory Space

Let us consider that the memory limitations in the platform imply that we can only store implementations of two modes out of the three modes in $\mathcal{M}^{\text{impl}}$. We cannot use the implementation of $\mathbf{M}_{12}$ or $\mathbf{M}_{13}$ to run the system in mode $\mathbf{M}_{123}$. Thus, the implementation of $\mathbf{M}_{123}$ is mandatory to be stored in memory of the platform. As we discussed in the beginning of this section, the total control cost when running mode $\mathbf{M}_{12}$ with the implementation of $\mathbf{M}_{123}$ is 4.17, compared to the total cost of 3.02 when running with the implementation of $\mathbf{M}_{12}$. The implementation of $\mathbf{M}_{12}$ thus gives a total cost reduction of 1.15. If, on the other hand, $\mathbf{M}_{13}$ runs with the implementation of $\mathbf{M}_{123}$, the total cost is $J_1^{\mathbf{M}_{123}} + J_3^{\mathbf{M}_{123}} = 2.98 + 2.24 = 5.22$. If $\mathbf{M}_{13}$ runs with its produced implementation, the total cost is $J_1^{\mathbf{M}_{13}} + J_3^{\mathbf{M}_{13}} = 1.60 + 1.95 = 3.55$. This gives a cost reduction of 1.67, which is better than the reduction obtained by the implementation of $\mathbf{M}_{12}$. Thus, in the presence of memory limitations, the implementations of $\mathbf{M}_{123}$ and $\mathbf{M}_{13}$ should be stored in memory to

achieve the best control performance.

In this discussion, we have assumed that $\mathbf{M}_{12}$ and $\mathbf{M}_{13}$ are equally important. However, if $\mathbf{M}_{12}$ occurs more frequently than $\mathbf{M}_{13}$, the cost improvement of the implementation of $\mathbf{M}_{12}$ becomes more significant. In this case, the best selection could be to exclude the implementation of $\mathbf{M}_{13}$ and store implementations of $\mathbf{M}_{123}$ and $\mathbf{M}_{12}$ in memory. Importance of modes relative to others are modeled in our framework by weights (Section 5.3).

5.2.3 Usage of Virtual Modes

As the last example, let us consider that $\mathbf{M}_{123}$ is a virtual mode (i.e., it does not occur at runtime). Let $\mathcal{M}^{\text{impl}} = \{\mathbf{M}_{12}, \mathbf{M}_{13}, \mathbf{M}_{23}\}$ be the set of modes with produced implementations. We have run the synthesis of mode $\mathbf{M}_{23} = \{\Lambda_2, \Lambda_3\}$ and obtained a total cost of 3.31. Let us assume that the three produced implementations of the modes in $\mathcal{M}^{\text{impl}}$ cannot be all stored in memory. If the implementation of $\mathbf{M}_{23}$ is removed, for example, there is no valid implementation of the functional mode $\mathbf{M}_{23}$ in memory. Similarly, if any of the other mode implementations are removed, then the system does not have a functionally correct implementation. To solve this problem, we implement the virtual mode $\mathbf{M}_{123}$. Its implementation can be used to run the system in all functional modes—however, with degraded control performance compared to the customized implementations of the modes in $\mathcal{M}^{\text{impl}}$. The available memory allows us to further store the implementation of one of the modes $\mathbf{M}_{12}$, $\mathbf{M}_{13}$, or $\mathbf{M}_{23}$. We choose the mode implementation that gives the largest cost reduction, compared to the implementation with $\mathbf{M}_{123}$. By studying the costs in Table 5.1 and considering the cost reductions in our discussions earlier in this example, we conclude that $\mathbf{M}_{13}$ gives the largest cost reduction among the modes in $\mathcal{M}^{\text{impl}}$. The best control performance under these tight memory constraints is achieved if the virtual mode $\mathbf{M}_{123}$ and functional mode $\mathbf{M}_{13}$ are implemented and stored in memory. This discussion shows that memory limitations can lead to situations in which virtual modes must be implemented to cover a set of functional modes.

5.3 Problem Formulation

In addition to the plants, control applications, and their mapping to an execution platform (Section 5.1), the inputs to the synthesis problem that we address in this chapter are

- a set of functional modes[1] $\mathcal{M}^{\text{func}} \subseteq \mathcal{M}$ that can occur at runtime (the set of virtual modes is $\mathcal{M}^{\text{virt}} = \mathcal{M} \setminus \mathcal{M}^{\text{func}}$);
- a weight[2] $w_{\mathbf{M}} > 0$ for each mode $\mathbf{M} \in \mathcal{M}^{\text{func}}$ ($w_{\mathbf{M}} = 0$ for $\mathbf{M} \in \mathcal{M}^{\text{virt}}$); and
- the available memory in the platform, modeled as a memory limit[3] $\mathsf{mem}_d^{\max} \in \mathbb{N}$ for each node N_d ($d \in \mathcal{I}_{\mathbf{N}}$).

The outputs are implementations of a set of modes $\mathcal{M}^{\text{impl}} \subseteq \mathcal{M} \setminus \{\emptyset\}$. For a design solution to be correct, we require that each functional mode has an implementation or a supermode implementation. Thus, for each functional mode $\mathbf{M} \in \mathcal{M}^{\text{func}} \setminus \{\emptyset\}$, we require that $\mathbf{M} \in \mathcal{M}^{\text{impl}}$ or $\mathcal{M}^{\text{impl}} \cap \overline{\mathcal{M}}(\mathbf{M}) \neq \emptyset$ hold.

If $\mathbf{M} \in \mathcal{M}^{\text{impl}}$, the cost $J^{\mathbf{M}}$ is given from the scheduling and synthesis step that produces the implementation (Equation 5.1). If $\mathbf{M} \notin \mathcal{M}^{\text{impl}}$ and $\mathbf{M} \neq \emptyset$, the cost $J^{\mathbf{M}}$ is given by the available supermode implementations: Given an implemented supermode $\mathbf{M}' \in \mathcal{M}^{\text{impl}} \cap \overline{\mathcal{M}}(\mathbf{M})$ of $\mathbf{M}$, the cost of $\mathbf{M}$ when using the implementation of $\mathbf{M}'$ is

$$J^{\mathbf{M}}(\mathbf{M}') = \sum_{i \in \mathcal{I}_{\mathbf{M}}} J_i^{\mathbf{M}'} = J^{\mathbf{M}'} - \sum_{i \in \mathcal{I}_{\mathbf{M}'} \setminus \mathcal{I}_{\mathbf{M}}} J_i^{\mathbf{M}'}. \tag{5.2}$$

[1] Due to the large number of modes, it can be impractical (or even impossible) to explicitly mark the set of functional modes. In such cases, however, the designer can indicate control loops that cannot be active in parallel due to functionality restrictions. Then, the set of functional modes $\mathcal{M}^{\text{func}}$ includes all modes except those containing two or more mutually exclusive controllers. If no specification of functional modes is made, it is assumed that $\mathcal{M}^{\text{func}} = \mathcal{M}$.

[2] It can be impractical for the designer to assign weights to all functional modes explicitly. An alternative is to assign weights to some particularly important and frequent modes (all other functional modes get a default weight). Another alternative is to correlate the weights to the occurrence frequency of modes (e.g., obtained by simulation).

[3] We model the memory consumption as an integer representing the number of units of physical memory that is occupied.

At runtime, the system uses the supermode implementation that gives the smallest cost. The cost $J^{\mathbf{M}}$ in the case $\mathbf{M} \notin \mathcal{M}^{\text{impl}}$ is thus

$$J^{\mathbf{M}} = \min_{\mathbf{M}' \in \mathcal{M}^{\text{impl}} \cap \overline{\mathcal{M}}(\mathbf{M})} J^{\mathbf{M}}(\mathbf{M}').$$

The cost of the idle mode $\emptyset$ is defined as $J^{\emptyset} = 0$.

The objective is to find a set of modes $\mathcal{M}^{\text{impl}}$ and synthesize them such that their implementations can be stored in the available memory of the platform. The cost to be minimized is

$$J_{\text{tot}} = \sum_{\mathbf{M} \in \mathcal{M}^{\text{func}}} w_{\mathbf{M}} J^{\mathbf{M}}. \tag{5.3}$$

This cost defines the cumulative control cost over the set of functional modes and gives the overall control quality of a multi-mode system with a certain set of implemented functional or virtual modes $\mathcal{M}^{\text{impl}} \subseteq \mathcal{M} \backslash \{\emptyset\}$.

5.4 Synthesis Approach

Our synthesis method consists of two parts. First, we synthesize implementations for a limited set of functional modes (Section 5.4.1). Second, we select the implementations to store under given memory constraints and, if needed, we synthesize implementations of some virtual modes (Section 5.4.2).

5.4.1 Control Quality versus Synthesis Time

The synthesis heuristic is based on a limited depth-first exploration of the Hasse diagram of modes. We use an improvement factor $\lambda \geqslant 0$ to limit the exploration and to find appropriate compromises between control quality and optimization time.

Before we discuss the details of the search heuristic, let us introduce several definitions. Given a set of modes $\mathcal{M}' \subseteq \mathcal{M}$, let us introduce the set

$$\mathcal{M}'_{\uparrow} = \{\mathbf{M} \in \mathcal{M}' : \overline{\mathcal{M}}(\mathbf{M}) \cap \mathcal{M}' = \emptyset\},$$

which contains the modes in $\mathcal{M}'$ that do not have supermodes in the same set $\mathcal{M}'$. Such modes are called *top* modes of $\mathcal{M}'$. For example, mode $\mathbf{\Lambda}$

is the only top mode of $\mathcal{M}$ (i.e., $\mathcal{M}_{\uparrow} = \{\mathbf{\Lambda}\}$). We also introduce the set of *immediate submodes* of $\mathbf{M} \in \mathcal{M}$ as

$$\mathcal{M}^{-}(\mathbf{M}) = \{\mathbf{M}' \in \underline{\mathcal{M}}(\mathbf{M}) : |\mathbf{M}| - 1 = |\mathbf{M}'|\},$$

and the set of *immediate supermodes* as

$$\mathcal{M}^{+}(\mathbf{M}) = \{\mathbf{M}' \in \overline{\mathcal{M}}(\mathbf{M}) : |\mathbf{M}| + 1 = |\mathbf{M}'|\}$$

For example, the set of immediate supermodes of mode $\{\Lambda_2\}$ is

$$\mathcal{M}^{+}(\{\Lambda_2\}) = \{\{\Lambda_1, \Lambda_2\}, \{\Lambda_2, \Lambda_3\}\}.$$

Let us now turn our attention to the synthesis heuristic for multi-mode systems. Figure 5.5 outlines the approach. In the first and second step, the set of modes with produced implementations is initialized to the empty set and an empty list is initialized. In this subsection, we consider only implementations of functional modes. Virtual modes are implemented only as a solution to memory constraints (Section 5.4.2). Note that it is mandatory to implement the top functional modes $\mathcal{M}_{\uparrow}^{\text{func}}$—that is, those functional modes that do not have any functional supermodes. For each such top functional mode $\mathbf{M}$, we perform Steps 3–5 in Figure 5.5. In Step 3, we run the synthesis in Chapter 4 for mode $\mathbf{M}$ to produce an implementation (periods, control laws, and schedule). After the synthesis of a mode $\mathbf{M}$, the set $\mathcal{M}^{\text{impl}}$ is updated by adding mode $\mathbf{M}$. We proceed, in Step 4, by finding the edges from mode $\mathbf{M}$ to its immediate submodes. These edges are added to the beginning of the list `edges`, which contains the edges leading to modes that are chosen for synthesis.

As long as the list `edges` is not empty, we perform Step 5, which is manifold: First, in Step (a), an edge $(\mathbf{M}, \mathbf{M}')$ is removed from the beginning of `edges`. In Step (b), if $\mathbf{M}' \neq \emptyset$ is a virtual mode, we do not consider it in the synthesis and resume the exploration with its immediate submodes. In Step (c), if an implementation of the functional mode $\mathbf{M}' \neq \emptyset$ has not yet been synthesized, we perform four steps (Steps i–iv). We first run the synthesis for mode $\mathbf{M}'$. Based on the obtained cost, we decide whether or not to continue synthesizing modes along the current path (i.e., synthesize immediate submodes of $\mathbf{M}'$). To take this decision, we consider the cost improvement of the implementation of mode $\mathbf{M}'$ relative to the cost when using the implementation of mode $\mathbf{M}$ to operate the system in mode $\mathbf{M}'$. We also consider the weights of the

Initialization:

1. $\mathcal{M}^{\text{impl}} \longleftarrow \emptyset$
2. **list** `edges` $\longleftarrow$ **empty**

For each mode $\mathbf{M} \in \mathcal{M}^{\text{func}}_{\uparrow}$, perform Steps 3–5 below.

3. Synthesize $\mathbf{M}$ and set $\mathcal{M}^{\text{impl}} \longleftarrow \mathcal{M}^{\text{impl}} \cup \{\mathbf{M}\}$
4. For each $\mathbf{M}' \in \mathcal{M}^{-}(\mathbf{M})$, add the edge $(\mathbf{M}, \mathbf{M}')$ to the beginning of the list `edges`
5. **while** `edges` $\neq$ **empty**
   (a) Remove edge $(\mathbf{M}, \mathbf{M}')$ from the beginning of `edges`
   (b) **if** $\mathbf{M}' \in \mathcal{M}^{\text{virt}}$ **and** $\mathbf{M}' \neq \emptyset$, for each $\mathbf{M}'' \in \mathcal{M}^{-}(\mathbf{M}')$, add $(\mathbf{M}, \mathbf{M}'')$ to the beginning of `edges`
   (c) **else if** $\mathbf{M}' \in \mathcal{M}^{\text{func}} \setminus \mathcal{M}^{\text{impl}}$ **and** $\mathbf{M}' \neq \emptyset$
      i. Synthesize $\mathbf{M}'$ and set $\mathcal{M}^{\text{impl}} \longleftarrow \mathcal{M}^{\text{impl}} \cup \{\mathbf{M}'\}$
      ii. $\Delta J^{\mathbf{M}'}(\mathbf{M}) \longleftarrow \left(J^{\mathbf{M}'}(\mathbf{M}) - J^{\mathbf{M}'}\right) \Big/ J^{\mathbf{M}'}(\mathbf{M})$
      iii. Compute the average weight w^{avg} of the immediate functional submodes of $\mathbf{M}'$
      iv. **if** $\Delta J^{\mathbf{M}'}(\mathbf{M}) \geqslant \frac{\lambda}{w^{\text{avg}}}$, for each $\mathbf{M}'' \in \mathcal{M}^{-}(\mathbf{M}')$, add $(\mathbf{M}', \mathbf{M}'')$ to the beginning of `edges`

Figure 5.5: Synthesis with improvement factor λ. The number of modes to synthesize is tuned by the designer through the improvement factor λ. The trade-off in this step is between control performance and synthesis time.

immediate submodes of $\mathbf{M}'$. In Step ii, we compute the relative cost improvement $\Delta J^{\mathbf{M}'}(\mathbf{M})$. The cost $J^{\mathbf{M}'}$ of the synthesized mode $\mathbf{M}'$ is defined in Equation 5.1 on page 68, whereas the cost $J^{\mathbf{M}'}(\mathbf{M})$ of mode $\mathbf{M}'$ when using the implementation of mode $\mathbf{M}$ is given by Equation 5.2 on page 74. In Step iii, we compute the average weight of the immediate functional submodes of $\mathbf{M}'$ as

$$w^{\text{avg}} = \frac{1}{|\mathcal{M}^{-}(\mathbf{M}') \cap \mathcal{M}^{\text{func}}|} \sum_{\mathbf{M}'' \in \mathcal{M}^{-}(\mathbf{M}') \cap \mathcal{M}^{\text{func}}} w_{\mathbf{M}''}.$$

In Step iv, it is decided whether to consider the immediate submodes $\mathcal{M}^{-}(\mathbf{M}')$ in the continuation of the synthesis. This decision is based on the given improvement factor $\lambda \geqslant 0$, the relative improvement $\Delta J^{\mathbf{M}'}(\mathbf{M})$,

and the average mode weight w^{avg}. Note that, in this way, the submodes with larger weights are given higher priority in the synthesis. If $\mathcal{M}^{-}(\mathbf{M}') \cap \mathcal{M}^{\text{func}} = \emptyset$ (i.e., all immediate submodes are virtual), the average weight is set to $w^{\text{avg}} = \infty$, which means that all immediate submodes are added in Step iv.

The parameter $\lambda \geqslant 0$ is used to regulate the exploration of the set of functional modes $\mathcal{M}^{\text{func}}$. For example, a complete synthesis of the functional modes corresponds to $\lambda = 0$. The results of this first part of the synthesis (Figure 5.5) are implementations of a set of functional modes $\mathcal{M}^{\text{impl}} \subseteq \mathcal{M}^{\text{func}}$. Note that all top functional modes are synthesized, which means that the system can operate in any functional mode with this design solution. The next subsection treats the case when all generated implementations cannot be stored in the memory of the underlying platform.

5.4.2 Control Quality versus Memory Consumption

Given are implementations of functional modes $\mathcal{M}^{\text{impl}}$ as a result of the first part presented in Section 5.4.1. We shall now discuss how to select the implementations to store, such that given memory constraints are satisfied and the system can operate in any functional mode with the stored implementations. Note that the set of top functional modes $\mathcal{M}^{\text{func}}_{\uparrow}$ must have implementations in memory. These implementations can be used to operate the system in any of the other functional modes. If the implementations of the top functional modes can be stored with the given available memory space of the platform, we do not need to synthesize implementations of virtual modes. The remaining problem is then to select additional implementations for modes in $\mathcal{M}^{\text{impl}} \setminus \mathcal{M}^{\text{func}}_{\uparrow}$ to store in memory. If, however, the implementations of the top functional modes cannot be stored, we must produce implementations of virtual modes to reduce the required memory space. These implementations should cover a large number of functional modes. An implementation of a virtual supermode can replace several implementations and, therefore, save memory space. These savings, however, are made possible at the cost of degraded control performance in the modes with replaced implementations.

To select the mode implementations to store in memory, we propose the two-step approach outlined in Figure 5.6. Step 1 asserts that there are implementations in memory to operate the system in any of the functional

1. **if** $\sum_{\mathbf{M} \in \mathcal{M}_{\uparrow}^{\text{impl}}} \mathsf{mem}_d^{\mathbf{M}} > \mathsf{mem}_d^{\max}$ for some $d \in \mathcal{I}_{\mathbf{N}}$
   (a) Find $\mathcal{M}' \subseteq \bigcup_{\mathbf{M} \in \mathcal{M}_{\uparrow}^{\text{impl}}} \mathcal{M}^{+}(\mathbf{M})$ with smallest size such that, for each $\mathbf{M} \in \mathcal{M}_{\uparrow}^{\text{impl}}$, $\mathcal{M}' \cap \overline{\mathcal{M}}(\mathbf{M}) \neq \emptyset$
   (b) For each $\mathbf{M}' \in \mathcal{M}'$, synthesize $\mathbf{M}'$ and set $\mathcal{M}^{\text{impl}} \longleftarrow \mathcal{M}^{\text{impl}} \cup \{\mathbf{M}'\}$
   (c) Repeat Step 1
2. Find $b_{\mathbf{M}} \in \{0, 1\}$ for each $\mathbf{M} \in \mathcal{M}^{\text{impl}}$ such that the cost in Equation 5.4 is minimized and the constraint in Equation 5.5 is satisfied for each $d \in \mathcal{I}_{\mathbf{N}}$ and $b_{\mathbf{M}} = 1$ for $\mathbf{M} \in \mathcal{M}_{\uparrow}^{\text{impl}}$

Figure 5.6: Mode-selection approach. If necessary, virtual modes are considered and synthesized if all implementations of top modes cannot be stored in the memory of the platform. The implementations to store in memory are the solutions to an integer linear program.

modes. Potentially, virtual modes must be implemented to guarantee a correct operation of the multi-mode system. We first check whether the implementations of the top functional modes $\mathcal{M}_{\uparrow}^{\text{impl}} = \mathcal{M}_{\uparrow}^{\text{func}}$ can be stored in the available memory on each computation node $N_d \in \mathbf{N}$. If not, we proceed by selecting virtual modes to implement. In Step (a), we find among the immediate supermodes of the top modes $\mathcal{M}_{\uparrow}^{\text{impl}}$ (these supermodes are virtual modes) a subset $\mathcal{M}'$ with minimal size that contains a supermode for each top mode. In Step (b), we produce implementations of the chosen virtual modes $\mathcal{M}'$ and add them to the set of implemented modes $\mathcal{M}^{\text{impl}}$. The set of top modes of $\mathcal{M}^{\text{impl}}$ is now $\mathcal{M}'$. We go back to Step 1 and check whether the implementations of the new top modes can be stored in memory. If not, we repeat Steps (a) and (b) and consider larger virtual modes. This process is repeated until we find a set of virtual modes that cover all functional modes and the implementations of those virtual modes can be stored in the available memory of the platform. After this first step, we know that the implementations of $\mathcal{M}_{\uparrow}^{\text{impl}}$ can be stored in the available memory and that they are sufficient to operate the system in any functional mode.

The second step selects mode implementations to store in the available memory such the control quality is optimized. This selection is done

by solving a linear program with binary variables: Given the set of implemented modes, possibly including virtual modes from Step 1 in Figure 5.6, let us introduce a binary variable $b_{\mathbf{M}} \in \{0,1\}$ for each mode $\mathbf{M} \in \mathcal{M}^{\text{impl}}$. If $b_{\mathbf{M}} = 1$, it means that the implementation of $\mathbf{M}$ is selected to be stored in memory, whereas it is not stored if $b_{\mathbf{M}} = 0$. For a correct implementation of the multi-mode system, the implementation of each top mode in $\mathcal{M}^{\text{impl}}$ must be stored in memory. Thus, $b_{\mathbf{M}} = 1$ for each $\mathbf{M} \in \mathcal{M}^{\text{impl}}_{\uparrow}$. To select the other mode implementations to store, we consider the obtained cost reduction (quality improvement) by implementing a mode. For any $\mathbf{M} \in \mathcal{M}^{\text{impl}} \setminus \mathcal{M}^{\text{impl}}_{\uparrow}$, this cost reduction is defined as

$$\Delta J^{\mathbf{M}} = \min_{\mathbf{M}' \in \mathcal{M}^{\text{impl}} \cap \overline{\mathcal{M}}(\mathbf{M})} \Delta J^{\mathbf{M}}(\mathbf{M}'),$$

where $\Delta J^{\mathbf{M}}(\mathbf{M}')$ is given in Step ii in Figure 5.5 (page 77). By not storing the generated implementation of a mode $\mathbf{M} \in \mathcal{M}^{\text{impl}} \setminus \mathcal{M}^{\text{impl}}_{\uparrow}$ (i.e., by setting $b_{\mathbf{M}} = 0$), we lose this control-quality improvement, which is relative to the control quality that is provided when the system operates in mode $\mathbf{M}$ with the best implementation of the top modes $\mathcal{M}^{\text{impl}}_{\uparrow}$.

When selecting the additional mode implementations to store in memory (i.e., when setting $b_{\mathbf{M}}$ for each $\mathcal{M}^{\text{impl}} \setminus \mathcal{M}^{\text{impl}}_{\uparrow}$), the cost to be minimized is the overall loss of cost reductions for mode implementations that are removed from $\mathcal{M}^{\text{impl}} \setminus \mathcal{M}^{\text{impl}}_{\uparrow}$. This cost is formulated as

$$\sum_{\mathbf{M} \in \mathcal{M}^{\text{impl}} \setminus \mathcal{M}^{\text{impl}}_{\uparrow}} (1 - b_{\mathbf{M}}) w_{\mathbf{M}} \Delta J^{\mathbf{M}}. \tag{5.4}$$

The memory consumption of an implementation of mode $\mathbf{M}$ is $\mathsf{mem}_d^{\mathbf{M}}$ on node $N_d \in \mathbf{N}$. This information is given as an output of the scheduling and control synthesis step. The memory constraint on node $N_d \in \mathbf{N}$ is

$$\sum_{\mathbf{M} \in \mathcal{M}^{\text{impl}}} b_{\mathbf{M}} \mathsf{mem}_d^{\mathbf{M}} \leqslant \mathsf{mem}_d^{\max}, \tag{5.5}$$

considering a given memory limit $\mathsf{mem}_d^{\max}$ on node N_d. Having formulated the linear program (Equations 5.4 and 5.5), Step 2 in Figure 5.6 is performed by optimization tools for integer linear programming; for the experiments presented in the next section, we have used the `eplex` library for mixed integer programming in ECL^{i}PS^e [AW07]. The implementations of the selected modes $\{\mathbf{M} \in \mathcal{M}^{\text{impl}} : b_{\mathbf{M}} = 1\}$ are chosen

to be stored in the memory of the underlying execution platform. This completes the synthesis of multi-mode control systems.

5.5 Experimental Results

We have conducted experiments to evaluate our proposed approach. We created 100 test cases from a set of inverted pendulums, ball and beam processes, DC servos, and harmonic oscillators [ÅW97]. We considered 10 percent of the modes to be virtual for test cases with 6 or more control loops. We generated 3 to 5 tasks for each control application; thus, the number of tasks in our test cases varies from 12 to 60. The tasks were mapped randomly to platforms comprising 2 to 10 nodes. Further, we considered uniform distributions of the execution times of tasks. The best-case and worst-case execution times of the tasks were chosen randomly from 2 to 10 milliseconds and the communication times of messages were chosen from 2 to 4 milliseconds. In this section, we show experimental results for platforms with static cyclic scheduling.

5.5.1 Control Quality versus Synthesis Time

In the first set of experiments, we evaluate the synthesis heuristic in Section 5.4.1. The synthesis was run for each test case and for different values of the improvement factor λ. All experiments were run on a PC with a quad-core CPU at 2.2 GHz, 8 Gb of RAM, and running Linux. For each value of λ and for each test case, we computed the obtained cost after the synthesis. This cost is

$$J_{\text{tot}}^{(\lambda)} = \sum_{\mathbf{M} \in \mathcal{M}^{\text{func}}} w_{\mathbf{M}} J^{\mathbf{M}}$$

(Equation 5.3) and we could compute it in affordable time for test cases with at most 12 control loops. Note that the time-consuming calculation of the cost is only needed for the experimental evaluation and is not part of the heuristic. We used the values $0, 0.1, 0.3$, and 0.5 for the improvement factor λ. Because of long runtime, we could run the synthesis with $\lambda = 0$ (exhaustive synthesis of all functional modes) and $\lambda = 0.1$ for test cases with at most 7 control loops.

As a baseline for the comparison, we used the cost $J_{\text{tot}}^{\uparrow}$ obtained when synthesizing only the top functional modes $\mathcal{M}_{\uparrow}^{\text{func}}$. This cost repre-

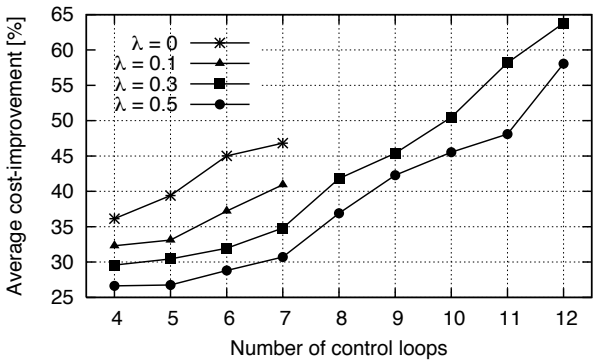

Figure 5.7: Control-performance improvements. The improvements are relative to the case when synthesized solutions are available only for the mandatory top modes. Smaller values of λ yield better design solutions, but at the cost of longer synthesis time.

sents the minimum level of control quality that is achieved if the system only has implementations of the mandatory top functional modes of the test case. For each test case and for each value of λ, we computed the achieved cost improvement as the relative difference

$$\Delta J_{\text{tot}}^{(\lambda)} = \frac{\left(J_{\text{tot}}^{\uparrow} - J_{\text{tot}}^{(\lambda)}\right)}{J_{\text{tot}}^{\uparrow}}.$$

Figure 5.7 shows the obtained cost improvements relative to the baseline solution. The vertical axis shows the average value of the relative cost improvement $\Delta J_{\text{tot}}^{(\lambda)}$ for test cases with number of control loops given on the horizontal axis. The results show that the achieved control performance becomes better with smaller values on λ (i.e., a more thorough exploration of the set of modes). We also observe that the improvement is better for larger number of control loops in the multi-mode system. This demonstrates that, for systems with many control loops, it is important to synthesize additional implementations for other modes than the mandatory top functional modes. The experiments also validate that significant quality improvements can be obtained with larger values of λ, which provide affordable runtimes for large examples.

The runtimes for the synthesis heuristic are shown in Figure 5.8. We

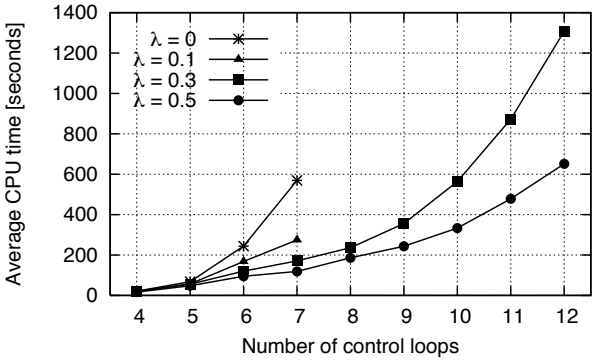

Figure 5.8: Runtimes for mode synthesis. Large values of λ lead to shorter design time, however with compromises in control performance.

show the average runtimes in seconds for the different values of the improvement factor λ and for each dimension of the test cases (number of control loops). Figures 5.7 and 5.8 illustrate how the designer can find trade-offs between control quality and optimization time, depending on the size of the control system. To further illustrate the scaling of synthesis time, we performed experiments with a system consisting of 20 control loops on 12 nodes. A solution was synthesized with $\lambda = 0.5$ in 38 minutes.

5.5.2 Control Quality versus Memory Consumption

In the second set of experiments, we evaluate the mode-selection approach. Towards this, we consider results from the mode-synthesis heuristic (Section 5.4.1) with $\lambda = 0.3$. Let us denote with $\mathsf{mem}_d^{\text{req}}$ the amount of memory needed to store the generated mode implementations on each node N_d. We have run the mode-selection algorithm (Section 5.4.2) with a memory limitation of $0.7\mathsf{mem}_d^{\text{req}}$ for each node. The average runtimes are shown in seconds in Figure 5.9. Note that the mode selection results are optimal if all top functional modes can be stored in memory. The selection of modes to store in memory (i.e., the solution of the integer linear program) is performed only once as a last step of the synthesis. As Figure 5.9 shows, the selection made in negligible runtime compared to the overall runtime of the system synthesis.

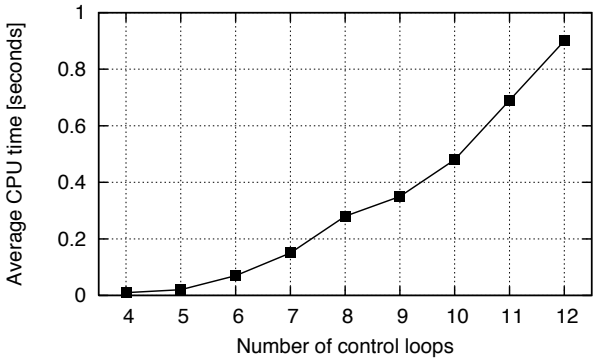

Figure 5.9: Runtimes for mode selection ($\lambda = 0.3$). The solver for the integer linear program finds an optimal solution in less than one second for large systems. This time overhead is insignificant compared to the overall design time of a multi-mode control system.

To study the quality degradation as a result of memory limitations, we used a test case with 12 control loops. We have run the mode synthesis for three scenarios: First, we considered no memory limitation in the platform, which resulted in a cost reduction (quality improvement) of 48.5 percent, relative to the cost obtained with the baseline approach (synthesis of only top functional modes). Second, we considered that only 70 percent of the memory required by the produced implementations can be used. As a result of the mode selection, all top functional modes could be stored, leading to a cost reduction of 41.4 percent (a quality degradation of 7.1 percent compared to the first scenario without memory constraints). For the third and last scenario, we considered memory limitations such that the implementations of the top functional modes cannot be all stored in memory. After the mode-selection approach, including the synthesis of virtual modes, we obtained a solution with a cost reduction of only 30.1 percent (a quality degradation of 18.4 percent compared to the case without memory constraints).

5.6 Summary and Discussion

In this chapter, we addressed control-performance optimization of embedded multi-mode control systems. The main design difficulty is raised

by the potentially very large number of operation modes. In this context, an appropriate selection of the actual modes to be implemented is of critical importance. We presented our synthesis approach that produces optimized schedules and controllers towards an efficient deployment of embedded multi-mode control systems.

We did not address the process of performing a mode change. This is an orthogonal and separate issue when developing systems with multiple operation modes. The mode-change protocols to implement and the synthesis procedure discussed in this chapter will both influence the overall quality of the system. Various mode-change protocols and their analysis methods are available in literature [RC04, PB98, SRLR89, SBB11, PLS11]. The correct protocol to choose depends on the application domain and its requirements. Independent of the mode-change protocol, the synthesis approach presented in this chapter produces schedules and controllers for an efficient deployment of embedded multi-mode control systems.

6
Synthesis of Fault-Tolerant Control Systems

MODERN distributed computing platforms may have components that fail temporarily or permanently during operation. In this chapter, we present a framework for fault-tolerant operation of control applications in the presence of permanent faults.

When a node fails, the *configuration* of the underlying distributed system changes. A configuration of the system is a set of currently operational computation nodes. When the configuration of the system changes (i.e., when one or several nodes fail), the tasks that were running on the failed nodes must now be activated at nodes of this new configuration. To guarantee stability and a minimum level of control quality of the control applications, it is necessary that a *solution* for this new configuration is synthesized at design time and that the system has the ability to adapt to this solution at runtime. A solution comprises a mapping, a schedule, and controllers that are optimized for the computation nodes in a certain configuration. However, such a construction is not trivial due to that the total number of configurations is exponential in the number of nodes of a distributed system. Synthesis of all possible configurations is thus impractical, because of the very large requirements on design time

and memory space of the underlying execution platform. In this chapter, we propose a method to synthesize solutions for a small number of *minimal* configurations and still provide guarantees on fault tolerance and a minimum level of control performance. Additional configurations are considered for synthesis with multiple trade-off considerations related to control performance, synthesis time, migration time, and memory consumption of the platform.

In the next section, we shall present notation and assumptions related to the distributed execution platform and its configurations. Section 6.2 presents a classification of the set of configurations, leading to the identification of *base* configurations in Section 6.3. Section 6.4 presents the extension of our synthesis framework in Chapter 4 for mapping optimization of distributed control systems. In Section 6.5, we consider the synthesis of solutions for base configurations and, if needed, for a set of *minimal* configurations. In Sections 6.6 and 6.7, we motivate and formulate a design-space exploration problem for control-quality optimization, followed by an optimization heuristic in Section 6.8. This chapter ends with experimental results in Section 6.9 and conclusions in Section 6.10.

6.1 Distributed Platform and Configurations

The execution platform comprises a set of computation nodes $\mathbf{N}$ that are connected to a single bus. The set of nodes $\mathbf{N}$ is indexed with $\mathcal{I}_{\mathbf{N}}$. Figure 6.1 shows a set of control loops comprising n plants $\mathbf{P}$ with index set $\mathcal{I}_{\mathbf{P}} = \{1, \ldots, n\}$ and, for each plant P_i, a control application Λ_i with three tasks $\mathbf{T}_i = \{\tau_{is}, \tau_{ic}, \tau_{ia}\}$. The edges indicate the data dependencies between tasks, as well as communication between sensors and actuators (arrows with dashed lines). For the platform in the same figure, we have $\mathbf{N} = \{N_A, N_B, N_C, N_D\}$ and its index set $\mathcal{I}_{\mathbf{N}} = \{A, B, C, D\}$.

We consider that a function $\Pi : \mathbf{T}_{\mathbf{\Lambda}} \longrightarrow 2^{\mathbf{N}}$ is given, as described in Section 3.2.3 on page 34. This determines the set of allowed computation nodes for each task in the system: The set of computation nodes that task $\tau \in \mathbf{T}_{\mathbf{\Lambda}}$ can be mapped to is $\Pi(\tau)$. In Figure 6.1, tasks τ_{1s} and τ_{1a} may be mapped to the nodes indicated by the dashed lines. Thus, we have $\Pi(\tau_{1s}) = \{N_A, N_C\}$ and $\Pi(\tau_{1a}) = \{N_C, N_D\}$. We consider that task τ_{1c} can be mapped to any of the four nodes in the platform—that is, $\Pi(\tau_{1c}) = \mathbf{N}$. We have omitted the many dashed lines for task τ_{1c} to

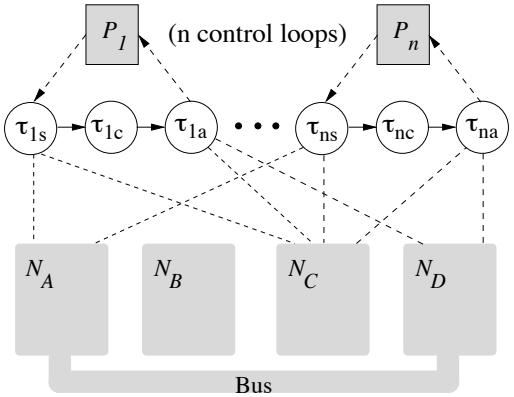

Figure 6.1: A set of feedback-control applications running on a distributed execution platform. Task τ_{1s} reads sensors and may execute on nodes N_A or N_C, whereas the actuator task τ_{1a} may execute on nodes N_C or N_D. Task τ_{1c} can be mapped to any of the four nodes.

obtain a clear illustration. The mapping constraints are the same for the other control applications.

At any moment in time, the system has a set of computation nodes $\mathbf{X} \subseteq \mathbf{N}$ that are operational. The remaining nodes $\mathbf{N} \setminus \mathbf{X}$ have failed and are not available for computation. We shall refer to $\mathbf{X}$ as a *configuration* of the distributed platform. The complete set of configurations is the power set $\mathcal{X} = 2^{\mathbf{N}}$ and is partially ordered by inclusion. The partial order of configurations is shown in Figure 6.2 as a Hasse diagram of configurations for our example with four computation nodes in Figure 6.1. For example, configuration $\{N_A, N_B, N_C\}$ indicates that N_D has failed and only the other three nodes are available for computation. The configuration $\emptyset$ indicates that all nodes have failed.

We consider that the platform implements appropriate mechanisms for fault detection [Kop97, KK07]. The failure of a node must be detected and all remaining operational nodes must know about such failures. In addition, the event that a failed node has been repaired is detected by the operational nodes in the system. This allows each operational node to know about the current configuration. Adaptation due to failed or repaired nodes involves switching schedules and control algorithms that are optimized for the available resources in the new configuration (Sec-

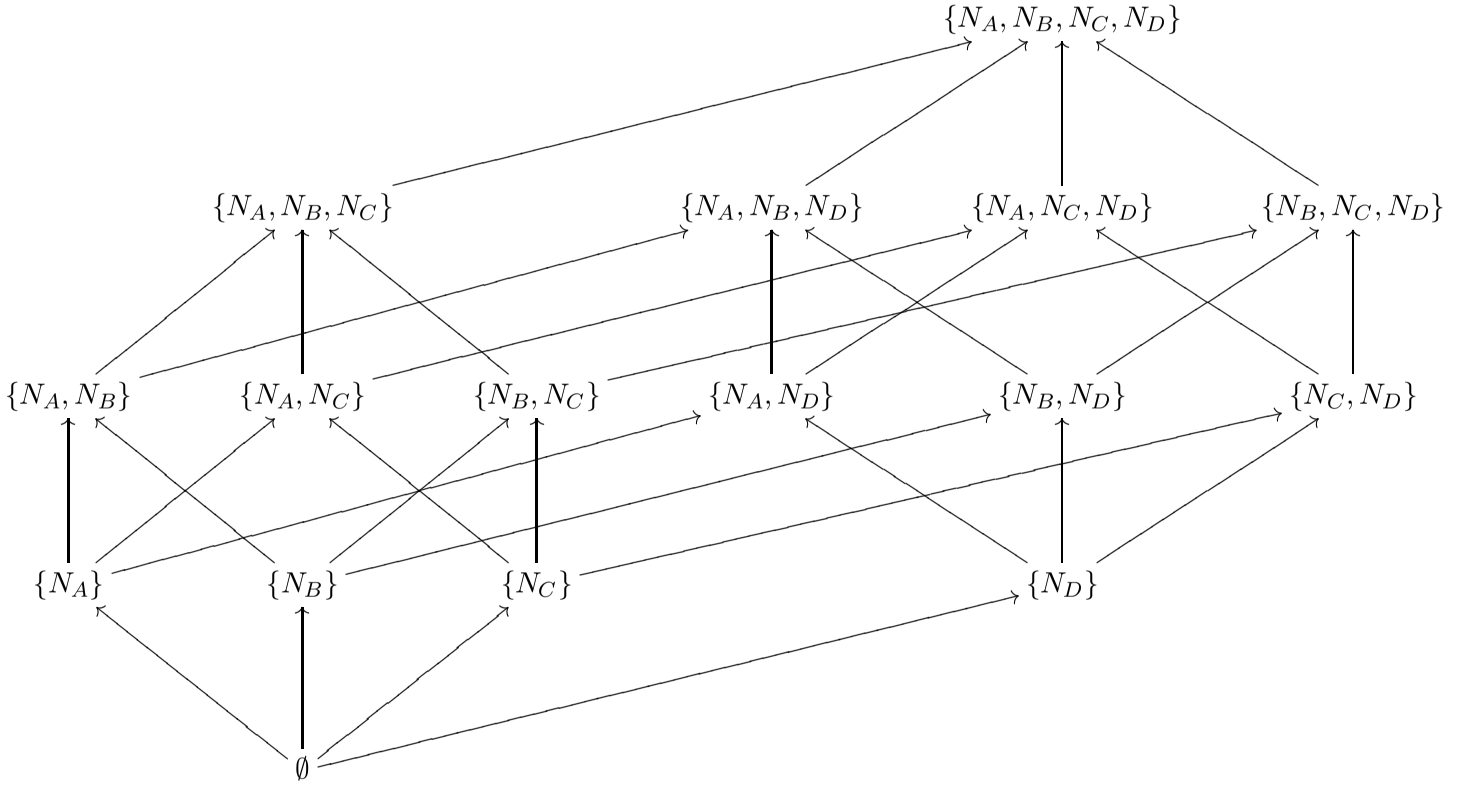

Figure 6.2: Hasse diagram of configurations. Sixteen possible configurations of the platform are shown. The vertices indicate configurations, which are partially ordered by inclusion. The edges connect configurations according to the subset relation. The configuration $\emptyset$ models the situation in which all nodes have failed.

tion 6.8), or, if no such optimizations have been performed at design time, to switch to mandatory backup solutions (Section 6.5). This information is stored in the nodes of the platform [SN05, Kop97]. Another phase during system reconfiguration is task migration [LKP+10] that takes place when tasks that were running on failed nodes are activated at other nodes in the system. We consider that the system has the ability to migrate tasks to other nodes in the platform. Each node stores information regarding those tasks that it must migrate through the bus when the system is adapting to a new configuration. This information is generated at design time (Section 6.8.2). For the communication, we assume that the communication protocol of the system ensures fault-tolerance for messages by different means of redundancy [NSSW05, Kop97].

6.2 Classification of Configurations

In this section, we shall provide a classification of the different configurations that may occur during operation. The first subsection illustrates the idea with the running example in Figure 6.1. The second subsection gives formal definitions of the different types of configurations.

6.2.1 Example of Configurations

Let us consider our example in Figure 6.1. Task τ_{1s} reads sensors and τ_{1a} writes to actuators. Task τ_{1c} does not perform input–output operations and can be executed on any node in the platform. Sensors can be read by nodes N_A and N_C, whereas actuation can be performed by nodes N_C and N_D. The mapping constraints for the tasks are thus given by $\Pi(\tau_{1s}) = \{N_A, N_C\}$, $\Pi(\tau_{1c}) = \mathbf{N}$, and $\Pi(\tau_{1a}) = \{N_C, N_D\}$. The same mapping constraints and discussion hold for the other control applications Λ_i ($i = 2, \ldots, n$). Thus, in the remainder of this example, we shall restrict the discussion to application Λ_1.

First, let us consider the initial scenario in which all computation nodes are operational and are executing one or several tasks each. The system is thus in configuration $\mathbf{N} = \{N_A, N_B, N_C, N_D\}$ (see Figure 6.2) and we assume that the actuator task τ_{1a} executes on node N_C in this configuration. Consider now that N_C fails and the system reaches configuration $\mathbf{X}_{ABD} = \{N_A, N_B, N_D\}$. Task τ_{1a} must now execute on N_D in this new configuration, because actuation can only be performed by

nodes N_C and N_D. According to the mapping constraints given by Π, there exists a possible mapping for each task in configuration $\mathbf{X}_{ABD}$, because $\mathbf{X}_{ABD} \cap \Pi(\tau) \neq \emptyset$ for each task $\tau \in \mathbf{T}_{\mathbf{\Lambda}}$. We refer to such configurations as *feasible* configurations. Thus, for a feasible configuration and any task, there is at least one node in that configuration on which the task can be mapped—without violation of the imposed mapping constraints.

If the system is in configuration $\mathbf{X}_{ABD}$ and node N_A fails, a new configuration $\mathbf{X}_{BD} = \{N_B, N_D\}$ is reached. Because task τ_{1s} cannot be mapped to any node in the new configuration, we say that $\mathbf{X}_{BD}$ is an *infeasible* configuration (i.e., we have $\Pi(\tau_{1s}) \cap \mathbf{X}_{BD} = \emptyset$). If, on the other hand, node N_B fails in configuration $\mathbf{X}_{ABD}$, the system reaches configuration $\mathbf{X}_{AD} = \{N_A, N_D\}$. In this configuration, tasks τ_{1s} and τ_{1a} must execute on N_A and N_D, respectively. Task τ_{1c} may run on either N_A or N_D. Thus, $\mathbf{X}_{AD}$ is a feasible configuration, because it is possible to map each task to a node that is both operational and allowed according to the given mapping restrictions. We observe that if either of the nodes in $\mathbf{X}_{AD}$ fails, the system reaches an infeasible configuration. We shall refer to configurations like $\mathbf{X}_{AD}$ as *base* configurations. Note that any configuration that is a superset of the base configuration $\mathbf{X}_{AD}$ is a feasible configuration. By considering the mapping constraints, we observe that the only other base configuration in this example is $\{N_C\}$ (node N_C may execute any task). The set of base configurations for our example system is thus

$$\mathcal{X}^{\text{base}} = \{\{N_A, N_D\}, \{N_C\}\}.$$

Let us consider that design solutions are generated for the two base configurations in $\mathcal{X}^{\text{base}}$. Considering Figure 6.2 again, we note that the mapping for base configuration $\{N_A, N_D\}$, including the produced schedule, task periods, and control laws, can be used to operate the system in the feasible configurations $\{N_A, N_B, N_C, N_D\}$, $\{N_A, N_B, N_D\}$, and $\{N_A, N_C, N_D\}$. This is done by merely using the two nodes in the base configuration (i.e., N_A and N_D), even though more nodes are operational in the mentioned feasible configurations. Similarly, base configuration $\{N_C\}$ covers another subset of the feasible configurations. Figure 6.3 shows the partial order that remains when infeasible configurations in Figure 6.2 are removed. Specifically, note that the two base configurations cover all feasible configurations together (there is a path to any feasible configuration, starting from a base configuration).

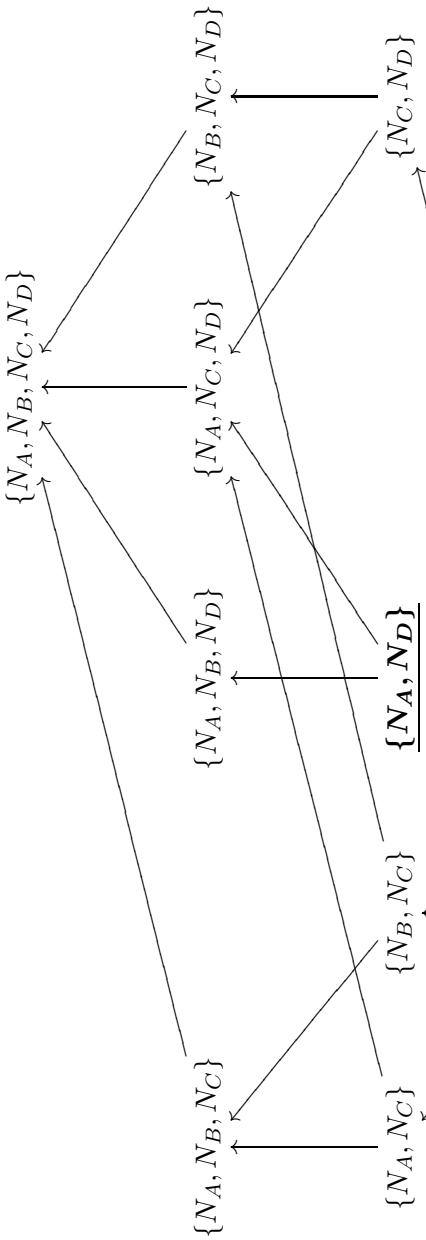

Figure 6.3: Partial Hasse diagram of the set of configurations. Only the feasible configurations are shown. The two base configurations (underlined and typeset in bold) cover all feasible configurations.

By generating a mapping (as well as customized schedules, periods, and control laws as in Chapter 4) for each base configuration, and considering that tasks are stored in the memory of the corresponding computation nodes to realize the base configuration mappings, the system can tolerate any sequence of node failures that lead the system to any feasible configuration. Thus, a necessary and sufficient step in the design phase (in terms of fault tolerance) is to identify the set of base configurations and to generate design solutions for them. It can be the case that the computation capacity is insufficient in some base configurations, because of the small number of operational nodes. We shall discuss this issue in Section 6.5. Although faults leading to any feasible configuration can be tolerated by the fact that execution is supported in base configurations, the control quality of the system can be improved if all computation nodes are utilized to efficiently distribute the executions. Towards this, we shall consider synthesis of additional feasible configurations in Section 6.8.

6.2.2 Formal Definitions

We consider that the mapping constraint $\Pi : \mathbf{T_\Lambda} \longrightarrow 2^{\mathbf{N}}$ is given, meaning that $\Pi(\tau)$ defines the set of computation nodes that task $\tau \in \mathbf{T_\Lambda}$ may execute on. Thus, Π decides directly the set of configurations for which the system must be able to adapt to by using the information that is synthesized at design time and stored in memory. Specifically, a given configuration $\mathbf{X} \in \mathcal{X}$ is defined as a *feasible* configuration if $\mathbf{X} \cap \Pi(\tau) \neq \emptyset$ for each task $\tau \in \mathbf{T_\Lambda}$. The set of feasible configurations is denoted $\mathcal{X}^{\text{feas}}$.

For an infeasible configuration $\mathbf{X} \in \mathcal{X} \setminus \mathcal{X}^{\text{feas}}$, there exists at least one task that due to the given mapping constraints cannot execute on any computation node in $\mathbf{X}$ (i.e., $\mathbf{X} \cap \Pi(\tau) = \emptyset$ for some $\tau \in \mathbf{T_\Lambda}$). A *base configuration* $\mathbf{X}$ is a feasible configuration for which the failure of any computation node $N \in \mathbf{X}$ results in an infeasible configuration $\mathbf{X} \setminus \{N\}$. The set of base configurations is thus defined as

$$\mathcal{X}^{\text{base}} = \{\mathbf{X} \in \mathcal{X}^{\text{feas}} : \mathbf{X} \setminus \{N\} \notin \mathcal{X}^{\text{feas}} \text{ for each } N \in \mathbf{X}\}.$$

The set of configurations $\mathcal{X} = 2^{\mathbf{N}}$ is thus partitioned into disjoint sets of feasible and infeasible configurations. Some of the feasible configurations form a set of base configurations, which represents the boundary between the set of feasible and infeasible configurations.

In the next section, we shall discuss an approach to identify the set of base configurations. In the ideal case, solutions for base configurations

are synthesized, enabling the system to operate in any feasible configuration. If not all base configurations allow for acceptable solutions to be synthesized, we construct solutions for a set of *minimal* configurations in Section 6.5 to cover as many feasible configurations as possible. Such situations may occur, for example, if the computation capacity is too restricted in certain base configurations.

6.3 Identification of Base Configurations

A straightforward approach to find the set of base configurations is to perform a search through the Hasse diagram of configurations. Given the mapping constraint $\Pi : \mathbf{T}_{\mathbf{\Lambda}} \longrightarrow 2^{\mathbf{N}}$, we find the set of base configurations $\mathcal{X}^{\text{base}}$ based on a breadth-first search [LD91] of the Hasse diagram of configurations. The search starts at the full configuration $\mathbf{N}$ with $\mathcal{X}^{\text{base}}$ initialized to $\emptyset$. It is assumed that $\mathbf{N}$ is a feasible configuration. Let us consider an arbitrary visit of a feasible configuration $\mathbf{X}$ during any point of the search. To determine whether or not to add $\mathbf{X}$ to the set of base configurations $\mathcal{X}^{\text{base}}$, we consider each configuration $\mathbf{X}'$ with $|\mathbf{X}'| = |\mathbf{X}| - 1$ (i.e., we consider the failure of any node in $\mathbf{X}$). If each such configuration $\mathbf{X}'$ is infeasible, we add $\mathbf{X}$ to the set of base configurations $\mathcal{X}^{\text{base}}$. Infeasible configurations $\mathbf{X}'$, as well as any configuration $\mathbf{X}'' \subset \mathbf{X}'$, are not visited during the search.

Due to the complexity of the Hasse diagram, a breadth-first search starting from the full configuration $\mathbf{N}$ is practical only for systems with relatively small number of nodes. Let us therefore discuss a practically efficient algorithm that constructs the set of base configurations $\mathcal{X}^{\text{base}}$ from the mapping constraint $\Pi : \mathbf{T}_{\mathbf{\Lambda}} \longrightarrow 2^{\mathbf{N}}$ directly. Without loss of generality, we shall assume that the function Π is injective (i.e., $\Pi(\tau_i) \neq \Pi(\tau_j)$ for $\tau_i \neq \tau_j$). If this is not the case, then, for the purpose of finding the set of base configurations, it is an equivalent problem to study an injective function $\Pi' : \mathbf{T}'_{\mathbf{\Lambda}} \longrightarrow 2^{\mathbf{N}}$ as a mapping constraint, where $\mathbf{T}'_{\mathbf{\Lambda}} \subset \mathbf{T}_{\mathbf{\Lambda}}$. Further in that case, it is required that, for each $\tau \in \mathbf{T}_{\mathbf{\Lambda}} \setminus \mathbf{T}'_{\mathbf{\Lambda}}$, there exists exactly one $\tau' \in \mathbf{T}'_{\mathbf{\Lambda}}$ for which $\Pi(\tau) = \Pi'(\tau')$. Finally, in the following discussion, $\mathbf{T}'_{\mathbf{\Lambda}}$ and Π' replace $\mathbf{T}_{\mathbf{\Lambda}}$ and Π, respectively.

We construct the set of base configurations starting from the tasks that have the most restrictive mapping constraints. Towards this, let us

consider a bijection

$$\sigma : \{1, \ldots, |\mathbf{T_\Lambda}|\} \longrightarrow \mathbf{T_\Lambda},$$

where

$$|\Pi(\sigma(k))| \leqslant |\Pi(\sigma(k+1))|$$

for $1 \leqslant k < |\mathbf{T_\Lambda}|$. This order of the tasks is considered during the construction of the set of base configurations $\mathcal{X}^{\text{base}}$. The construction is based on a function

$$\mathsf{construct} : \{1, \ldots, |\mathbf{T_\Lambda}|\} \longrightarrow 2^{\mathcal{X}},$$

where $\mathsf{construct}(k)$ returns a set of configurations that include the base configurations of the system when considering the mapping constraints for only tasks $\sigma(1), \ldots, \sigma(k)$. We shall give a recursive definition of the function $\mathsf{construct}$. For the base case, we define

$$\mathsf{construct}(1) = \bigcup_{N \in \Pi(\sigma(1))} \{N\}.$$

Before we define $\mathsf{construct}(k)$ for $1 < k \leqslant |\mathbf{T_\Lambda}|$, let us define a function

$$\mathsf{feasible} : \mathcal{X} \times \{1, \ldots, |\mathbf{T_\Lambda}|\} \longrightarrow 2^{\mathcal{X}}$$

as

$$\mathsf{feasible}(\mathbf{X}, k) = \{\mathbf{X}\} \tag{6.1}$$

if $\mathbf{X} \cap \Pi(\sigma(k)) \neq \emptyset$—that is, if configuration $\mathbf{X}$ already includes an allowed computation node for task $\sigma(k)$—and

$$\mathsf{feasible}(\mathbf{X}, k) = \bigcup_{N \in \Pi(\sigma(k))} \mathbf{X} \cup \{N\} \tag{6.2}$$

otherwise. If $\mathbf{X}$ contains a computation node that task $\sigma(k)$ can execute on, then $\mathsf{feasible}(\mathbf{X}, k)$ does not add additional nodes to $\mathbf{X}$ (Equation 6.1). If not, however, then $\mathsf{feasible}(\mathbf{X}, k)$ extends $\mathbf{X}$ in several directions given by the set of nodes $\Pi(\sigma(k))$ that task $\sigma(k)$ may execute on (Equation 6.2). Now, we define recursively

$$\mathsf{construct}(k) = \bigcup_{\mathbf{X} \in \mathsf{construct}(k-1)} \mathsf{feasible}(\mathbf{X}, k)$$

for $1 < k \leqslant |\mathbf{T_\Lambda}|$. The set $\mathsf{construct}(k)$ thus comprises configurations for which it is possible to execute the tasks $\{\sigma(1), \ldots, \sigma(k)\}$ according to the mapping constraints induced by Π.

We know by construction that $\mathcal{X}^{\text{base}} \subseteq \mathsf{construct}(|\mathbf{T_\Lambda}|)$. We also know that $\mathsf{construct}(|\mathbf{T_\Lambda}|)$ does not contain infeasible configurations. A pruning of the set $\mathsf{construct}(|\mathbf{T_\Lambda}|)$ must be performed to identify feasible configurations $\mathsf{construct}(|\mathbf{T_\Lambda}|) \setminus \mathcal{X}^{\text{base}}$ that are not base configurations. This shall end our discussion regarding the identification of the set $\mathcal{X}^{\text{base}}$.

6.4 Task Mapping for Feasible Configurations

Let us define the *mapping* of the task set $\mathbf{T_\Lambda}$ onto a feasible configuration $\mathbf{X} \in \mathcal{X}^{\text{feas}}$ as a function $\mathsf{map}_{\mathbf{X}} : \mathbf{T_\Lambda} \longrightarrow \mathbf{X}$. For each task $\tau \in \mathbf{T_\Lambda}$, $\mathsf{map}_{\mathbf{X}}(\tau)$ is the computation node that executes task τ when the system configuration is $\mathbf{X}$. It is required that the mapping constraints are considered, meaning that $\mathsf{map}_{\mathbf{X}}(\tau) \in \Pi(\tau)$ for each $\tau \in \mathbf{T_\Lambda}$. For a given feasible configuration $\mathbf{X} \in \mathcal{X}^{\text{feas}}$ and mapping $\mathsf{map}_{\mathbf{X}} : \mathbf{T_\Lambda} \longrightarrow \mathbf{X}$, we use our integrated control and scheduling framework for distributed embedded systems in Chapter 4 to obtain a design solution. The solution parameters that are synthesized are the periods and control laws, as well as the execution and communication schedule of the tasks and messages in the system. The objective is to minimize the overall control cost

$$J^{\mathbf{X}} = \sum_{i \in \mathcal{I}_{\mathbf{P}}} J_i, \tag{6.3}$$

which indicates maximization of the total control quality of the system, under the consideration that only the nodes in $\mathbf{X}$ are operational.

We have used a genetic algorithm-based approach—similar to the approach in Section 4.3.2—to optimize task mapping [ASEP11]. The mapping affects the delay characteristics indirectly through task and message scheduling. It is thus of great importance to optimize task mapping, schedules, and control laws to obtain a customized solution with high control quality in a given configuration. Thus, for a given $\mathbf{X} \in \mathcal{X}^{\text{feas}}$, we can find a customized mapping $\mathsf{map}_{\mathbf{X}} : \mathbf{T_\Lambda} \longrightarrow \mathbf{X}$ together with a schedule and controllers (periods and control laws). The mapping is constructed to satisfy the mapping constraints (i.e., $\mathsf{map}_{\mathbf{X}}(\tau) \in \Pi(\tau)$ for each $\tau \in \mathbf{T_\Lambda}$) and with the objective to minimize the total control cost

given by Equation 6.3. We denote with $\mathsf{mem}_d^{\mathbf{X}}$ the amount of memory required on node N_d ($d \in \mathcal{I}_{\mathbf{N}}$) to store information related to the mapping, schedule, periods, and control laws that are customized for configuration $\mathbf{X}$. This memory consumption is given as an output of the synthesis step; we shall consider this memory consumption in the context of memory limitations in Section 6.8.2.

6.5 Minimal Configurations

By definition, it is not possible to operate the system in infeasible configurations, because at least one task cannot be executed in such situations. In this section, we shall discuss the synthesis of mandatory solutions that are required to achieve system operation in feasible configurations. The first approach is to synthesize solutions for each base configuration of the system. It can be the case, however, that no solution can be found for some base configurations; the control cost in Equation 6.3 is infinite in such cases, indicating that at least one control loop is unstable. If a solution cannot be found for a certain configuration, this means that the computation capacity of the platform is insufficient for that configuration. In such cases, we shall progressively synthesize solutions for configurations with additional computation nodes.

We first synthesize a solution for each base configuration $\mathbf{X} \in \mathcal{X}^{\text{base}}$. If a solution could be found—the control cost $J^{\mathbf{X}}$ is finite—then that solution can be used to operate the system in any feasible configuration $\mathbf{X}' \in \mathcal{X}^{\text{feas}}$ for which $\mathbf{X} \subseteq \mathbf{X}'$. If a solution cannot be found for base configuration $\mathbf{X}$, we proceed by synthesizing solutions for configurations with one additional computation node. This process is repeated as long as solutions cannot be found. Let us now outline such an approach.

During the construction of solutions to configurations, we shall maintain two sets $\mathcal{X}^{\text{min}}$ and $\mathcal{X}^*$ with initial values $\mathcal{X}^{\text{min}} = \emptyset$ and $\mathcal{X}^* = \mathcal{X}^{\text{base}}$. The set $\mathcal{X}^{\text{min}}$ shall contain the configurations that have been synthesized successfully: Their design solutions have finite control cost and stability is guaranteed. The set $\mathcal{X}^*$ contains configurations that are yet to be synthesized. The following steps are repeated as long as $\mathcal{X}^* \neq \emptyset$.

1. Select any configuration $\mathbf{X} \in \mathcal{X}^*$.

2. Synthesize a solution for $\mathbf{X}$. This results in the control cost $J^{\mathbf{X}}$.

3. Remove $\mathbf{X}$ from $\mathcal{X}^*$ by the update

$$\mathcal{X}^* \longleftarrow \mathcal{X}^* \setminus \{\mathbf{X}\}.$$

4. If $J^{\mathbf{X}} < \infty$, update $\mathcal{X}^{\min}$ according to

$$\mathcal{X}^{\min} \longleftarrow \mathcal{X}^{\min} \cup \{\mathbf{X}\}, \tag{6.4}$$

otherwise update $\mathcal{X}^*$ as

$$\mathcal{X}^* \longleftarrow \mathcal{X}^* \cup \bigcup_{N \in \mathbf{N} \setminus \mathbf{X}} (\mathbf{X} \cup \{N\}). \tag{6.5}$$

5. If $\mathcal{X}^* \neq \emptyset$, go back to Step 1.

In the first three steps, configurations can be chosen for synthesis in any order from $\mathcal{X}^*$. In Step 4, we observe that the set $\mathcal{X}^*$ becomes smaller as long as solutions can be synthesized with finite control cost (Equation 6.4). If a solution for a certain configuration cannot be synthesized (i.e., the synthesis framework returns an infinite control cost, indicating an unstable control system), we consider configurations with one additional computation node to increase the possibility to find solutions (Equation 6.5).

The configurations for which solutions could be synthesized form a set of *minimal* feasible configurations $\mathcal{X}^{\min}$. The set of minimal configurations $\mathcal{X}^{\min}$ is thus defined by Steps 1–5. A configuration $\mathbf{X} \in \mathcal{X}^{\min}$ is minimal in the sense that it is either a base configuration or it is a feasible configuration with minimal number of nodes that cover a base configuration that could not be synthesized due to insufficient computation capacity of the platform. For each minimal configuration $\mathbf{X} \in \mathcal{X}^{\min}$, we consider that each node $N \in \mathbf{X}$ stores all tasks $\tau \in \mathbf{T}_{\mathbf{\Lambda}}$ for which $\mathsf{map}_{\mathbf{X}}(\tau) = N$; that is, we consider that tasks are stored permanently on nodes to realize mappings for minimal configurations. Further, we consider that all information (e.g., periods, control laws, and schedules) that is needed to switch solutions for minimal configurations at runtime is stored in the memory of computation nodes.

The set of feasible configurations for which the system is operational with our solution is

$$\mathcal{X}^{\text{oper}} = \bigcup_{\mathbf{X} \in \mathcal{X}^{\min}} \{\mathbf{X}' \in \mathcal{X}^{\text{feas}} : \mathbf{X} \subseteq \mathbf{X}'\} \tag{6.6}$$

and it includes the minimal configurations $\mathcal{X}^{\text{min}}$, as well as feasible configurations that are covered by a minimal configuration. The system is not able to operate in the feasible configurations $\mathcal{X}^{\text{feas}} \setminus \mathcal{X}^{\text{oper}}$—this set represents the border between base and minimal configurations—because of insufficient computation capacity of the platform. A direct consequence of the imposed mapping constraints is that the system cannot operate when it is in any infeasible configuration in $\mathcal{X} \setminus \mathcal{X}^{\text{feas}}$. Infeasible configurations, as well as feasible configurations not covered by minimal configurations, are identified by our approach. To tolerate particular fault scenarios that lead the system to configurations in

$$\left(\mathcal{X} \setminus \mathcal{X}^{\text{feas}}\right) \cup \left(\mathcal{X}^{\text{feas}} \setminus \mathcal{X}^{\text{oper}}\right),$$

the problem of insufficient computation capacity has to be solved by considering complementary fault-tolerance techniques (e.g., hardware replication). The system remains operational in all other configurations $\mathcal{X}^{\text{oper}}$ by using the solutions generated for minimal configurations. As a special case, we have $\mathcal{X}^{\text{min}} = \mathcal{X}^{\text{base}}$ if solutions to all base configurations could be synthesized. In that case, we have $\mathcal{X}^{\text{oper}} = \mathcal{X}^{\text{feas}}$, meaning that the system is operational in all feasible configurations.

6.6 Motivational Example for Optimization

The synthesis of a set of minimal configurations $\mathcal{X}^{\text{min}}$ in the previous section results in a solution that covers all fault scenarios that lead the system to a configuration in $\mathcal{X}^{\text{oper}}$ (Equation 6.6). The synthesis of minimal configurations provides not only fault tolerance for the configurations $\mathcal{X}^{\text{oper}}$ but also a minimum level of control quality. Considering that all solutions for minimal configurations are realized by storing information in the memory of the platform, we shall in this section motivate and formulate an optimization problem for control-quality improvements, relative to the minimum quality provided by minimal configurations.

Let us resume our example in Section 6.2.1 by considering synthesis of additional configurations than the minimal configurations. We have considered three control applications for three inverted pendulums (i.e., $n = 3$ Figure 6.1). We shall find that such optimizations can lead to better control quality than a system that only uses the mandatory design solutions for minimal configurations.

Table 6.1: Control costs for several configurations. The first two entries indicate a minimum level of control quality given by the two minimal configurations. Control quality is improved (cost is reduced) for configurations with additional operational nodes.

Configuration $\mathbf{X}$	Control cost $J^{\mathbf{X}}$
$\{N_A, N_D\}$	5.2
$\{N_C\}$	7.4
$\{N_A, N_B, N_C, N_D\}$	3.1
$\{N_A, N_B, N_C\}$	4.3

6.6.1 Improved Solutions for Feasible Configurations

Let us consider the set of base configurations

$$\mathcal{X}^{\text{base}} = \{\{N_A, N_D\}, \{N_C\}\}.$$

Considering that solutions for the two base configurations have been synthesized, and that these solutions have finite control costs, we note that the set of minimal configurations is $\mathcal{X}^{\text{min}} = \mathcal{X}^{\text{base}}$. We thus have $\mathcal{X}^{\text{oper}} = \mathcal{X}^{\text{feas}}$, meaning that the system can operate in any feasible configuration with the solutions for minimal configurations. Let us also consider that a customized solution (mapping, schedule, and controllers) has been synthesized for the configuration in which all nodes are operational. This solution exploits the full computation capacity of the platform to achieve as high control quality as possible. Note that all feasible configurations can be handled with solutions for the two base configurations (Figure 6.3).

We shall now improve control quality by additional synthesis of configurations. Towards this, we have synthesized solutions for the two minimal configurations, as well as configuration $\{N_A, N_B, N_C\}$. Table 6.1 shows the obtained control costs defined by Equation 6.3. Considering that a solution for $\{N_A, N_B, N_C\}$ would not have been generated, then in that configuration the system can only run with the solution for the minimal configuration $\{N_C\}$ with a cost of 7.4. By generating a customized solution, however, we can achieve a better control quality in that configuration according to the obtained cost 4.3—a cost improvement of 3.1. By synthesizing additional feasible configurations, we can obtain additional control-quality improvements—however, at the expense

Table 6.2: Task mapping for two configurations and three control applications. Each row shows tasks that run on a certain node in a given configuration.

$\mathbf{X}$	N_A	N_B	N_C	N_D
$\{N_A, N_B, N_C\}$	$\tau_{1s}, \tau_{2s}, \tau_{3s}$	$\tau_{1c}, \tau_{2c}, \tau_{3c}$	$\tau_{1a}, \tau_{2a}, \tau_{3a}$	–
$\{N_C\}$	–	–	$\tau_{1s}, \tau_{1c}, \tau_{1a}, \tau_{2s}, \tau_{2c}, \tau_{2a}, \tau_{3s}, \tau_{3c}, \tau_{3a}$	–

of the total synthesis time of all solutions. The particular selection of additional configurations to synthesize solutions for is based on the allowed synthesis time, the failure probabilities of the nodes in the system, and the potential improvement in control quality relative to the minimum level provided by the minimal configurations. We shall elaborate on this selection in more detail in Section 6.8.

6.6.2 Mapping Realization

Once a solution for a configuration—not a minimal configuration—has been synthesized, it must be verified whether it is possible for the system to adapt to this solution at runtime. Thus, for the additional mapping of configuration $\{N_A, N_B, N_C\}$ in our example, we must check whether the mapping can be realized if the system is in configuration $\{N_A, N_B, N_C, N_D\}$ and node N_D fails. In Table 6.2, we show the mapping for this configuration, as well as the mapping of its corresponding minimal configuration $\{N_C\}$. For the minimal configurations, we consider that the tasks are stored on the corresponding computation nodes. For example, the tasks in the column for N_C, corresponding to the minimal configuration, are stored on node N_C. Let us consider the mapping of the tasks to the configuration $\{N_A, N_B, N_C\}$. We note that all tasks that are needed to realize the mapping for node N_C are already stored on that node. Nodes N_A and N_B, however, do not store the tasks that are needed to realize the mapping for configuration $\{N_A, N_B, N_C\}$. When switching to the solution for this configuration—from configuration $\{N_A, N_B, N_C, N_D\}$—the tasks for nodes N_A and N_B need to be migrated from node N_C. Note that it is always possible to migrate tasks

from nodes in a minimal configuration: Because any feasible configuration in $\mathcal{X}^{\text{oper}}$ is covered by a minimal configuration, which realizes its mapping by storing tasks in memory of the operational nodes, there is always at least one operational node that stores a certain task for a given feasible configuration.

During task migration, the program state does not need to be transferred (because of the feedback mechanism of control applications, the state is automatically restored when task migration has completed). The migration time cannot exceed specified bounds, in order to guarantee stability. Hence, if the migration time for tasks τ_{1s}, τ_{2s}, τ_{3s}, τ_{1c}, τ_{2c}, and τ_{3c} satisfies the specified bound, the system can realize the solution for configuration $\{N_A, N_B, N_C\}$ at runtime.

If the time required to migrate the required tasks at runtime exceeds the given bounds, then the solution for the minimal configuration $\{N_C\}$ is used at runtime with control cost 7.4. In that case, the operational nodes N_A and N_B are not utilized. Alternatively, more memory can be used to store additional tasks on nodes N_A and N_B, in order to realize the mapping at runtime without or with reduced task migration. In this way, we avoid the excessive amount of migration time and we can realize the mapping, although at the cost of larger required memory space to achieve the better control cost of 4.3 in configuration $\{N_A, N_B, N_C\}$. In the following section, we present a formal statement of the design-space exploration problem for control-quality optimization. Thereafter, in Section 6.8, we present an optimization approach that synthesizes selected configurations and considers the trade-off between control quality, memory cost, and synthesis time.

6.7 Problem Formulation

Given is a distributed platform with computation nodes $\mathbf{N}$, a set of plants $\mathbf{P}$, and their control applications $\mathbf{\Lambda}$. We consider that a task mapping $\mathsf{map}_{\mathbf{X}} : \mathbf{T}_{\mathbf{\Lambda}} \longrightarrow \mathbf{X}$, as well as corresponding schedules and controllers, have been generated for each minimal configuration $\mathbf{X} \in \mathcal{X}^{\text{min}}$ as discussed in Section 6.5. We consider that tasks are stored permanently on appropriate computation nodes to realize the task mappings for the minimal configurations (i.e., no task migration is needed at runtime to adapt to solutions for minimal configurations). Thus, to realize the mappings

for minimal configurations, each task $\tau \in \mathbf{T_\Lambda}$ is stored on nodes

$$\bigcup_{\mathbf{X} \in \mathcal{X}^{\min}} \{\mathsf{map}_{\mathbf{X}}(\tau)\}.$$

The set of tasks that are stored on node $N_d \in \mathbf{N}$ is

$$\mathbf{T}^{(d)} = \bigcup_{\mathbf{X} \in \mathcal{X}^{\min}} \{\tau \in \mathbf{T_\Lambda} : \mathsf{map}_{\mathbf{X}}(\tau) = N_d\}. \tag{6.7}$$

In addition, the inputs specific to the optimization step discussed in this section are

- the time $\mu(\tau)$ required to migrate task τ from a node to any other node in the platform;
- the maximum amount of migration time $\mu_i^{\max}$ for plant P_i (this constraint is based on the maximum amount of time that a plant P_i can stay in open loop without leading to instability [Tab07] or degradation of control quality below a specified threshold, as well as the actual time to detect faults [Kop97, KK07]);
- the memory space $\mathsf{mem}_d(\tau)$ required to store task $\tau \in \mathbf{T_\Lambda}$ on node N_d ($d \in \mathcal{I}_{\mathbf{N}}$);
- the additional available memory $\mathsf{mem}_d^{\max}$ of each node N_d in the platform (note that this does not include the memory consumed for the minimal configurations, as these are mandatory to implement and sufficient dedicated memory is assumed to be provided); and
- the failure probability $p(N)$ per time unit for each node $N \in \mathbf{N}$.

The failure probability $p(N)$ depends on the mean time to failure (MTFF) of the computation node. The MTFF is decided by the technology of the production process, the ambient temperature of the components, and voltage or physical shocks that the components may suffer in the operational environment of the system [KK07].

The decision variables of the optimization problem are a subset of configurations $\mathcal{X}^{\text{impl}} \subseteq \mathcal{X}^{\text{oper}} \setminus \mathcal{X}^{\min}$ and a mapping $\mathsf{map}_{\mathbf{X}}$, schedule, and controllers for each $\mathbf{X} \in \mathcal{X}^{\text{impl}}$. Thus, in addition to the minimal

configurations, we generate mappings for the other feasible configurations $\mathcal{X}^{\text{impl}}$. We require that $\mathbf{N} \in \mathcal{X}^{\text{impl}}$, which means that it is mandatory to generate solutions for the case when all nodes in the system are operational.

Let us now define the cost that characterizes the overall control quality of the system in any feasible configuration based on the solutions (mappings, schedules, and controllers) for the selected set of configurations. We shall associate a cost $J^{\mathbf{X}}$ for each feasible configuration $\mathbf{X} \in \mathcal{X}^{\text{oper}}$. If $\mathbf{X} \in \mathcal{X}^{\text{min}} \cup \mathcal{X}^{\text{impl}}$, a customized mapping for that configuration has been generated with a cost $J^{\mathbf{X}}$ given by Equation 6.3. If $\mathbf{X} \notin \mathcal{X}^{\text{min}} \cup \mathcal{X}^{\text{impl}}$ and $\mathbf{X} \in \mathcal{X}^{\text{oper}}$, then at runtime the system uses the mapping of a configuration $\mathbf{X}'$ for which $\mathbf{X}' \in \mathcal{X}^{\text{min}} \cup \mathcal{X}^{\text{impl}}$ and $\mathbf{X}' \subset \mathbf{X}$. It is guaranteed that such a configuration $\mathbf{X}'$ can be found in the set of minimal configurations $\mathcal{X}^{\text{min}}$ (Equation 6.6). If such a configuration is also included in $\mathcal{X}^{\text{impl}}$, then the control quality is better than in the corresponding minimal configuration because of better utilization of the operational computation nodes. Thus, for the case $\mathbf{X} \in \mathcal{X}^{\text{oper}} \setminus (\mathcal{X}^{\text{min}} \cup \mathcal{X}^{\text{impl}})$, the cost of the feasible configuration $\mathbf{X}$ is

$$J^{\mathbf{X}} = \min_{\substack{\mathbf{X}' \in \mathcal{X}^{\text{min}} \cup \mathcal{X}^{\text{impl}} \\ \mathbf{X}' \subset \mathbf{X}}} J^{\mathbf{X}'}, \tag{6.8}$$

which means that the best functionally correct solution—in terms of control quality—is used to operate the system in configuration $\mathbf{X}$. The cost to minimize when selecting the set of additional feasible configurations $\mathcal{X}^{\text{impl}} \subseteq \mathcal{X}^{\text{oper}} \setminus \mathcal{X}^{\text{min}} \setminus \{\mathbf{N}\}$ to synthesize is defined as

$$J = \sum_{\mathbf{X} \in \mathcal{X}^{\text{oper}} \setminus \mathcal{X}^{\text{min}} \setminus \{\mathbf{N}\}} p^{\mathbf{X}} J^{\mathbf{X}}, \tag{6.9}$$

where $p^{\mathbf{X}}$ is the probability of node failures that lead the system to configuration $\mathbf{X}$ (we shall discuss the computation of this probability in Equation 6.10 on page 107). Towards this, we shall consider the given failure probability $p(N)$ of each computation node $N \in \mathbf{N}$.

The cost in Equation 6.9 characterizes the control quality of the system as a function of the additional feasible configurations for which solutions have been synthesized. If solutions are available only for the set of minimal configurations, the system tolerates all node failures that lead

the system to a configuration in $\mathcal{X}^{\text{oper}}$—however, at a large cost J in Equation 6.9. This is because other feasible configurations operate at runtime with solutions of minimal configurations. In those situations, not all operational computation nodes are utilized, at the cost of reduced overall control quality. By synthesizing solutions for additional feasible configurations in $\mathcal{X}^{\text{oper}} \setminus \mathcal{X}^{\text{min}} \setminus \{\mathbf{N}\}$, the cost in Equation 6.9 is reduced (i.e., the overall control quality is improved) due to the cost reduction in the terms related to the selected set of configurations.

6.8 Optimization Approach

Figure 6.4 shows an overview of our proposed design approach. The first component, which we discussed in Sections 6.3–6.5, is the identification of base configurations and synthesis of minimal configurations (labeled as "fault-tolerant design" in the figure). The second component (labeled as "optimization") comprises the exploration and synthesis of additional configurations, as well as the mapping-realization step that considers the constraints related to task migration and memory space. This second component is the topic of this section and is our proposed solution to the problem formulation in Section 6.7. The selection and synthesis of additional feasible configurations is described in Section 6.8.1. For each synthesized feasible configuration, it must be checked whether the solution can be realized with regard to the memory consumption in the platform and the amount of task migration required at runtime. Memory and migration trade-offs, as well as memory-space and migration-time constraints, are presented in Section 6.8.2.

6.8.1 Exploration of the Set of Configurations

Our optimization heuristic aims to minimize the cost in Equation 6.9 and is based on a priority-based search of the Hasse diagram of configurations. The priorities are computed iteratively as a step of the optimization process based on probabilities for the system to reach the different configurations. The heuristic belongs to the class of anytime algorithms, meaning that it can be stopped at any point in time and return a feasible solution. This is due to that minimal configurations already have been synthesized and fault tolerance is achieved. The overall quality of the system is improved as more optimization time is invested.

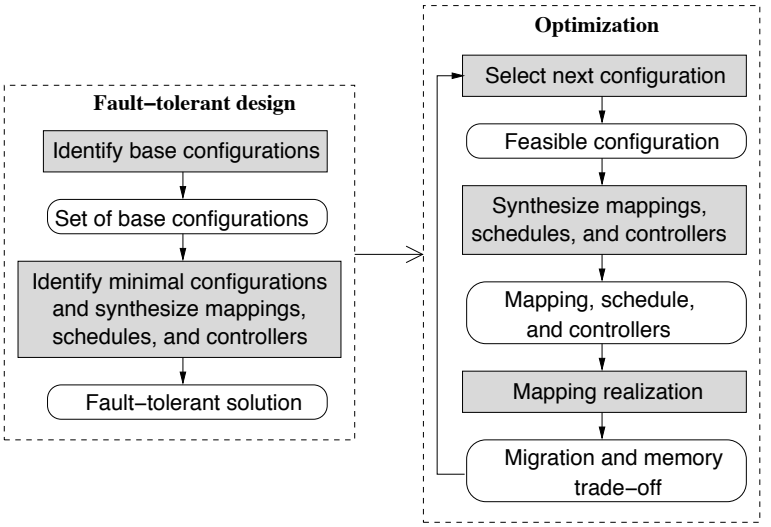

Figure 6.4: Overview of the design framework. The first step is to construct solutions for a set of minimal configurations, which is based on the identification of base configurations, to achieve fault-tolerance and a minimum control-quality level. In the second step, the system is further optimized for additional configurations.

Initially, as a mandatory step, we synthesize a mapping for the configuration $\mathbf{N}$, in order to support the execution of the control system for the case when all computation nodes are operational. During the exploration process, a priority queue with configurations is maintained. Whenever a mapping $\mathsf{map}_{\mathbf{X}} : \mathbf{T}_{\mathbf{\Lambda}} \longrightarrow \mathbf{X}$ has been synthesized for a certain feasible configuration $\mathbf{X} \in \mathcal{X}^{\text{oper}}$ (note that $\mathbf{N}$ is the first synthesized configuration), each feasible configuration $\mathbf{X}' \subset \mathbf{X}$ with $|\mathbf{X}'| = |\mathbf{X}| - 1$ is added to the priority queue with priority equal to the probability

$$p^{\mathbf{X}'} = p^{\mathbf{X}} p(N), \tag{6.10}$$

where $\{N\} = \mathbf{X} \setminus \mathbf{X}'$. For the initial configuration $\mathbf{N}$, we consider $p^{\mathbf{N}} = 1$.

Subsequently, for configuration $\mathbf{X}$, we check whether it is possible to realize the generated mapping $\mathsf{map}_{\mathbf{X}} : \mathbf{T}_{\mathbf{\Lambda}} \longrightarrow \mathbf{X}$ at runtime with task migration and the available additional memory to store tasks. This step is described in detail in the next subsection (Section 6.8.2). If this

step succeeds, it means that the mapping can be realized at runtime and we thus add $\mathbf{X}$ to the set $\mathcal{X}^{\text{impl}}$ (this set is initially empty). Further in that case, for each node N_d, the set of tasks $\mathbf{T}^{(d)}$ stored on N_d and the amount of additional consumed memory mem_d is updated. The set of tasks $\mathbf{T}^{(d)}$ that are stored on node N_d is initialized according to Equation 6.7. If the mapping realization does not succeed, the generated solution for configuration $\mathbf{X}$ is excluded. This means that a solution for a minimal configuration must be used at runtime to operate the system in the feasible configuration $\mathbf{X}$. Independently of whether the mapping realization of $\mathbf{X}$ succeeds, the exploration continues by generating a solution for the next configuration in the maintained priority queue of configurations. The exploration terminates when the additional memory space on all computation nodes has been consumed, or when a specified design time has passed (e.g., the designer stops the exploration process). Let us now discuss the mapping-realization step that deals with the memory and migration-time constraints for a given solution of a configuration.

6.8.2 Mapping Realization

In the previous subsection (Section 6.8.1), we proposed a search order to explore and synthesize solutions for other feasible configurations than the minimal configurations. For each configuration $\mathbf{X} \in \mathcal{X}^{\text{oper}} \setminus \mathcal{X}^{\text{min}}$ that is considered in the exploration process, a mapping $\mathsf{map}_{\mathbf{X}} : \mathbf{T}_{\mathbf{\Lambda}} \longrightarrow \mathbf{X}$ is constructed (along with customized schedules and controllers). We shall in the remainder of this section focus on whether and how this mapping can be realized at runtime in case the system reaches configuration $\mathbf{X}$. We first check whether there is sufficient memory to store information related to the solution (mapping, schedules, and controllers) for the configuration. The required memory for this information is denoted $\mathsf{mem}_d^{\mathbf{X}}$ and is an output of the mapping and synthesis step for configuration $\mathbf{X}$ (Section 6.4). Let us denote with mem_d the amount of additional memory that is already consumed on N_d for other configurations in $\mathcal{X}^{\text{impl}} \subset \mathcal{X}^{\text{oper}} \setminus \mathcal{X}^{\text{min}}$. If

$$\mathsf{mem}_d + \mathsf{mem}_d^{\mathbf{X}} > \mathsf{mem}_d^{\text{max}}$$

for some $d \in \mathcal{I}_{\mathbf{N}}$, it means that the information related to the mapping, schedules, and controllers for configuration $\mathbf{X}$ cannot be stored on the computation platform. For such cases, we declare that the mapping

$\mathsf{map}_{\mathbf{X}}$ cannot be realized (we remind that solutions for minimal configurations, however, can be used to operate the system in configuration $\mathbf{X}$).

If the solution for $\mathbf{X}$ can be stored within the given memory limit, we check whether migration of tasks that are needed to realize the mapping can be done within the maximum allowed migration time

$$\mu^{\max} = \min_{i \in \mathcal{I}_{\mathbf{P}}} \mu_i^{\max}.$$

If the migration-time constraint cannot be met, we reduce the migration time below the threshold $\mu^{\max}$ by storing tasks in the memory of computation nodes (this memory consumption is separate from the memory space needed to store tasks for the realization of minimal configurations). The main idea is to store as few tasks as possible to satisfy the migration-time constraint. Towards this, let us consider the set of tasks

$$\mathbf{\Psi}_d(\mathbf{X}) = \left\{ \tau \in \mathbf{T}_{\mathbf{\Lambda}} \setminus \mathbf{T}^{(d)} : \mathsf{map}_{\mathbf{X}}(\tau) = N_d \right\}$$

that need to be migrated to node N_d at runtime in order to realize the mapping $\mathsf{map}_{\mathbf{X}}$, given that $\mathbf{T}^{(d)}$ is the set of tasks that are already stored on node N_d. The objective is to find a set of tasks $\mathbf{S}_d \subseteq \mathbf{\Psi}_d(\mathbf{X})$ to store on each node $N_d \in \mathbf{N}$ such that the memory consumption is minimized and the maximum allowed migration time is considered. We formulate this problem as an integer linear program (ILP) by introducing a binary variable b_d^τ for each node $N_d \in \mathbf{N}$ and each task $\tau \in \mathbf{\Psi}_d(\mathbf{X})$. Task $\tau \in \mathbf{\Psi}_d(\mathbf{X})$ is stored on N_d if $b_d^\tau = 1$, and migrated if $b_d^\tau = 0$. The memory constraint is thus formulated as

$$\mathsf{mem}_d + \mathsf{mem}_d^{\mathbf{X}} + \sum_{\tau \in \mathbf{\Psi}_d(\mathbf{X})} b_d^\tau \mathsf{mem}_d(\tau) \leqslant \mathsf{mem}_d^{\max}, \tag{6.11}$$

which models that the memory consumption $\mathsf{mem}_d^{\mathbf{X}}$ of the solution together with the memory needed to store the selected tasks do not exceed the memory limitations. The migration-time constraint is formulated similarly as

$$\sum_{d \in \mathcal{I}_{\mathbf{N}}} \left(\sum_{\tau \in \mathbf{\Psi}_d(\mathbf{X})} (1 - b_d^\tau) \mu(\tau) \right) \leqslant \mu^{\max}. \tag{6.12}$$

The memory cost to minimize in the selection of $\mathbf{S}_d \subseteq \mathbf{\Psi}_d(\mathbf{X})$ is given by

$$\sum_{d \in \mathcal{I}_\mathbf{N}} \left(\sum_{\tau \in \mathbf{\Psi}_d(\mathbf{X})} b_d^\tau \mathsf{mem}_d(\tau) \right). \tag{6.13}$$

If a solution to the ILP formulation cannot be found, then the mapping cannot be realized. If a solution is found, we have

$$\mathbf{S}_d = \{\tau \in \mathbf{\Psi}_d(\mathbf{X}) : b_d^\tau = 1\}$$

and we update the set $\mathbf{T}^{(d)}$ and the memory consumption mem_d, respectively, according to

$$\mathbf{T}^{(d)} \longleftarrow \mathbf{T}^{(d)} \cup \mathbf{S}_d$$

and

$$\mathsf{mem}_d \longleftarrow \mathsf{mem}_d + \mathsf{mem}_d^\mathbf{X} + \sum_{\tau \in \mathbf{S}_d} \mathsf{mem}_d(\tau).$$

Even for large systems, the ILP given by Equations 6.13, 6.11, and 6.12 can be solved optimally and efficiently with modern solvers. We have used the `eplex` library for ILP in ECL^{i}PS^e [AW07], and it incurred negligible time overhead—less than one second—in our experiments.

6.9 Experimental Results

We have conducted experiments to evaluate our proposed design framework. We constructed a set of test cases with inverted pendulums, ball and beam processes, DC servos, and harmonic oscillators [ÅW97]. The test cases vary in size between 5 and 9 computation nodes with 4 to 6 control applications. All experiments were performed on a PC with a quad-core CPU at 2.2 GHz, 8 GB of RAM, and running Linux.

As a baseline of comparison, we considered a straightforward design approach for which we synthesize solutions for all minimal configurations and the initial configuration $\mathbf{N}$. This constitutes the mandatory set of solutions to achieve fault tolerance in any feasible configuration, as well as an optimized solution for the case when all nodes are operational. We computed a cost $J^{\min}$ according to Equation 6.9, considering that solutions have been synthesized for minimal configurations and the initial configuration, and that all other feasible configurations run with the

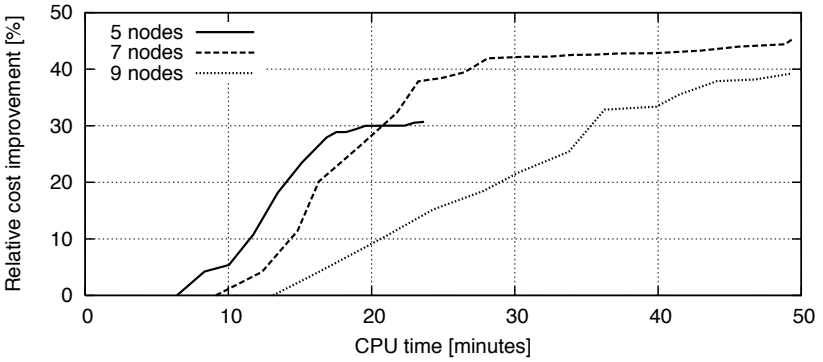

Figure 6.5: Relative cost improvements and runtimes of the proposed design approach. The synthesis time related to zero improvement corresponds to the construction of solutions for the mandatory minimal configurations and the configuration in which all nodes are operational. Additional design time for other feasible configurations leads to improved control quality.

corresponding minimal configuration with the minimum level of control quality given by Equation 6.8. The cost $J^{\min}$ indicates the overall control quality of the fault-tolerant control system with only the mandatory solutions synthesized.

Subsequently, we made experiments with our optimization heuristic to select additional configurations for synthesis. For each feasible configuration that is synthesized, individual cost terms in Equation 6.9 are decreased (control quality is improved compared to what is provided by minimal configurations). The optimization phase was conducted for varying amounts of design time. For each additional configuration that was synthesized, the total cost in Equation 6.9 was updated. Reminding that a small control cost indicates high control quality, and vice versa, we are interested in the control-cost improvement

$$\frac{J^{\min} - J}{J^{\min}}$$

relative to the control cost $J^{\min}$ that is obtained when only considering the mandatory configurations.

Figure 6.5 shows the design time on the horizontal axis and the corresponding relative improvement on the vertical axis. The design time

corresponding to the case of zero improvement refers to the mandatory design phase of identification and synthesis of minimal configurations. The mandatory design phase for minimal configurations is around only 10 minutes, which is sufficient to cover all fault scenarios and provide a minimum level of control quality in any feasible configuration. Any additional design time that is invested leads to improved control quality compared to the already synthesized fault-tolerant solution. For example, we can achieve an improvement of around 30 percent already after 20 minutes for systems with 5 and 7 computation nodes. We did not run the heuristic for the case of 5 nodes for more than 23 minutes, because at that time it has already synthesized all feasible configurations. For the other cases, the problem size was too large to afford an exhaustive exploration of all configurations. It should be noted that the quality improvement is smaller at large design times. At large design times, the heuristic typically evaluates and optimizes control quality for configurations with many failed nodes. However, these quality improvements do not contribute significantly to the overall quality (Equation 6.9), because the probability of many nodes failing is very small (Equation 6.10). We conclude that the designer can stop the optimization process when the improvement at each step is no longer considered significant.

6.10 Summary and Discussion

We proposed a design framework for distributed embedded control applications with support for execution even if some computation nodes in the system fail. We presented an algorithm to identify base configurations and construct mappings for minimal configurations of the distributed system to achieve fault-tolerant operation. To improve the overall control quality relative to the minimum level of quality provided by the minimal configurations, we construct additional design solutions efficiently.

The system can adapt to situations in which nodes have failed by switching to an appropriate solution that has been synthesized at design time. Task replication and migration are mechanisms that are used to implement remapping of control tasks. In this way, the system adapts to different configurations as a response to failed components. These mechanisms and the solutions prepared by our framework are sufficient to operate the system in case computation nodes fail. The alternative to

this software-based approach is hardware replication, which can be very costly in some application domains; for example, the design of many applications in the automotive domain are highly cost constrained.

We note that our framework is not restricted to control applications, but can be adapted to other application domains for which distributed platforms and fault-tolerance requirements are inherent. In particular, our idea of base and minimal configurations is general and may be applied to any application area. The information regarding base and minimal configurations also serves as an indication to the designer regarding those computation nodes that are of particular importance. Hardware replication of nodes in minimal configurations reduces the probability of reaching infeasible configurations, or reaching configurations that are not covered by minimal configurations, whereas all other fault scenarios are handled with the less costly software-based approach that we presented in this chapter. The design optimization problem is relevant for other application domains for which performance metrics exist and depend on the available computing and communication resources.

7

Scheduling of Self-Triggered Control Systems

SCHEDULING of multiple self-triggered control tasks on uniprocessor platforms is the topic of this chapter. Self-triggered control has been proposed recently as a resource-efficient alternative to periodic control. The motivation is the need to reduce the number of controller executions while still maintaining a certain level of control performance. This requirement may be due to constraints and costs related to the allowed amount of sensing, actuation, as well as computation and communication bandwidth. Energy consumption and resource bandwidth are two examples of such costs. To exploit the advantages of self-triggered control approaches, in the context of multiple control loops on a shared computation infrastructure, we present a software-based scheduling component for optimization of control performance and CPU usage. Stability of the control system is guaranteed by a design-time verification step and by construction of the scheduling heuristic.

This chapter is organized as follows. We present the system and plant model in Section 7.1. In Section 7.2, we discuss temporal properties of self-triggered control. Section 7.3 shows an example of the execution of multiple self-triggered tasks on a uniprocessor platform. The exam-

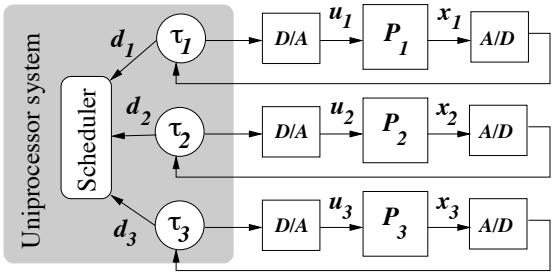

Figure 7.1: Control-system architecture. The three feedback-control loops include three control tasks on a uniprocessor computation platform. Deadlines are computed at runtime and given to the scheduler.

ple further highlights the scheduling and optimization objectives of this chapter. The scheduling problem is defined in Section 7.4 and is followed by the scheduling heuristic in Sections 7.5 and 7.6. Experimental results with comparisons to periodic control are presented in Section 7.7. We summarize the contribution of this chapter in Section 7.8.

7.1 System Model

Let us in this section introduce the system model and components that we shall consider throughout this chapter. Figure 7.1 shows an example of a control system with a CPU hosting three control tasks τ_1, τ_2, and τ_3—depicted with white circles—that implement controllers for the three plants P_1, P_2, and P_3, respectively. The outputs of a plant are connected to A/D converters and sampled by the corresponding control task. The produced control signals are written to the actuators through D/A converters and are held constant until the next execution of the task. The tasks are scheduled on the CPU according to some scheduling policy, priorities, and deadlines. The scheduler component is the main contribution of this chapter.

The set of self-triggered control tasks and its index set are denoted with $\mathbf{T}$ and $\mathcal{I}_{\mathbf{T}}$, respectively. Each task $\tau_i \in \mathbf{T}$ ($i \in \mathcal{I}_{\mathbf{T}}$) implements a given feedback controller of a plant. The dynamical properties of this plant are given by a linear, continuous-time state-space model

$$\dot{\boldsymbol{x}}_i(t) = A_i \boldsymbol{x}_i(t) + B_i \boldsymbol{u}_i(t) \tag{7.1}$$

in which the vectors $\boldsymbol{x}_i$ and $\boldsymbol{u}_i$ are the plant state and controlled input, respectively. The plant state is measured and sampled by the control task τ_i. The controlled input $\boldsymbol{u}_i$ is updated at time-varying sampling intervals according to the control law

$$\boldsymbol{u}_i = K_i \boldsymbol{x}_i \tag{7.2}$$

and is held constant between executions of the control task. We thus consider that it is possible to measure all components of the state vector $\boldsymbol{x}_i$. The actual time instants of these updates are determined by the scheduler at runtime, based on the deadlines and trade-offs between control quality and CPU usage. The control gain K_i is given and is computed by control design for continuous-time controllers. The design of K_i typically addresses some costs related to the plant state $\boldsymbol{x}_i$ and controlled input $\boldsymbol{u}_i$. Like in Equation 3.1 on page 27, we may add disturbances to Equation 7.1. However, unlike the stochastic model for disturbances assumed in the previous chapters, the magnitudes of these disturbances are considered to be bounded and can be taken into account by a self-triggered control task when computing deadlines for its future execution [MT09]. The worst-case execution time of task τ_i is denoted c_i and is obtained at design time with tools for worst-case execution time analysis [WEE+08].

7.2 Timing of Self-Triggered Control

A self-triggered control task [VFM03, VMB08, AT08a, AT09, WL09, MAT10] uses the sampled plant states not only to compute control signals, but also to compute a deadline for the next task execution. Stability of the control system is guaranteed if this deadline is met at runtime. A self-triggered control task comprises two execution segments. The first execution segment consists of three sequential parts:

1. Sampling of the plant state $\boldsymbol{x}$ (possibly followed by some data processing).
2. Computation of the control signal $\boldsymbol{u}$.
3. Actuation of the control signal.

This first execution segment is similar to what is performed by a traditional periodic control task.

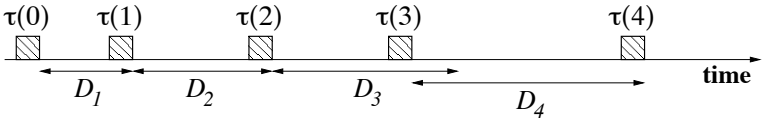

Figure 7.2: Execution of a self-triggered control task. Each job of the task computes a completion deadline for the next job. The deadline is time varying and state dependent.

The second execution segment is characteristic to self-triggered control and computes temporal deadlines on task executions. As shown by Anta and Tabuada [AT08a, AT09], the computation of the deadlines is based on the sampled state, the control law, and the plant dynamics in Equation 7.1. The computed deadlines are valid if there is no preemption between sampling and actuation (a constant delay between sampling and actuation can be taken into consideration). The deadline of a task execution is taken into account by the scheduler and must be met to guarantee stability of the control system. Thus, in addition to the first execution segment, which comprises sampling and actuation, a self-triggered control task computes—in the second execution segment—a completion deadline D on the next task execution. This deadline is relative to the completion time of the task execution. The exact time instant of the next task execution, however, is decided by the scheduler based on optimizations of control performance and CPU usage.

Figure 7.2 shows the execution of several jobs $\tau(q)$ of a control task τ. After the first execution of τ (i.e., after the completion of job $\tau(0)$), we have a relative deadline D_1 for the completion of the second execution of τ. The deadline of $\tau(1)$ is denoted D_1 and is relative to the completion time of job $\tau(0)$. Note that the deadline between two consecutive job executions is varying; this shows that the control-task execution is regulated by the dynamically changing plant state, rather than by a fixed period. Note that the fourth execution of τ, job $\tau(3)$, starts and completes before the imposed deadline D_3. The reason why this execution is placed earlier than its deadline can be due to control-quality optimizations or conflicts with the execution of other control tasks. The deadline D_4 of the successive execution is relative to the completion time and not relative to the previous deadline.

For a control task $\tau_i \in \mathbf{T}$, it is possible to compute a lower and upper bound $D_i^{\min}$ and $D_i^{\max}$, respectively, for the deadline of a task ex-

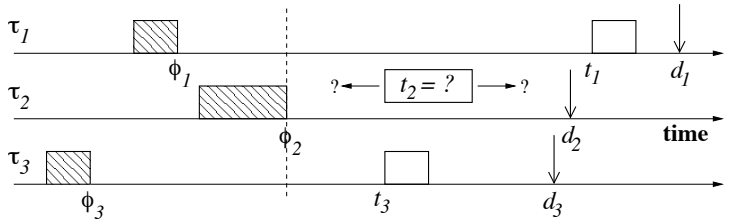

Figure 7.3: Scheduling example. Tasks τ_1 and τ_3 have completed their execution before ϕ_2 and their next execution instants are t_1 and t_3, respectively. Task τ_2 completes execution at time ϕ_2 and its next execution instant t_2 has to be decided by the scheduler. The imposed deadlines must be met to guarantee stability.

ecution relative to the completion of its previous execution. We thus have $D_i \in [D_i^{\min}, D_i^{\max}]$. The minimum relative deadline $D_i^{\min}$ bounds the CPU requirement of the control task and is computed at design time based on the plant dynamics and control law [AT08a, Tab07]. The maximum relative deadline is used to ensure that the control task executes with a certain minimum rate (e.g., to achieve some level of robustness to disturbances or a minimum amount of control quality).

7.3 Motivational Example

Figure 7.3 shows the execution of three self-triggered control tasks τ_1, τ_2, and τ_3. The time axes show the scheduled executions of the three tasks. Considering that the current time is ϕ_2, a dashed rectangle indicates a completed task execution, where the length of the rectangle represents the execution time of the task. The white rectangles show jobs that are scheduled for execution after time moment ϕ_2. The scenario is that task τ_2 has finished its execution at time ϕ_2, including the computation of its next deadline d_2. The scheduler is activated at time ϕ_2 to schedule the next execution of τ_2, considering the existing scheduled executions of τ_1 and τ_3 (the white rectangles) and the deadlines d_1, d_2, and d_3. Prior to time ϕ_2, task τ_3 finished its execution at ϕ_3 and its next execution was placed at time t_3 by the scheduler. In a similar way, the start time t_1 of task τ_1 was decided at its most recent completion time ϕ_1. Other application tasks may execute in the time intervals in which no control

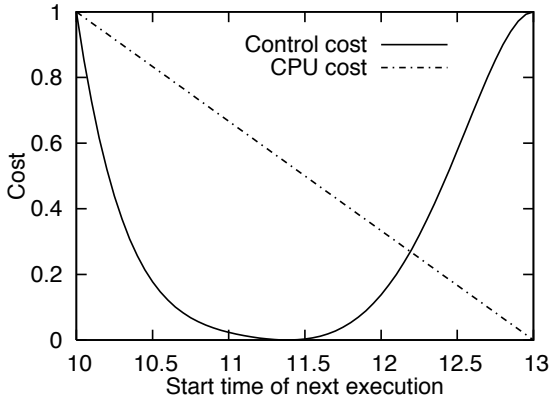

Figure 7.4: Control and CPU costs. The two costs depend on the next execution instant of the control task and are in general conflicting objectives.

task is scheduled for execution.

The objective of the scheduler at time ϕ_2 in Figure 7.3 is to schedule the next execution of task τ_2 (i.e., to decide the start time t_2) such that the task completes execution before or at the deadline d_2. This decision is in part based on trade-offs between control quality and CPU usage. Figure 7.4 shows an example of the control and CPU costs—as functions of t_2—of task τ_2 with solid and dashed lines, respectively. Note that a low control cost in the figure indicates high control quality, and vice versa. For this example, we have $\phi_2 = 10$ and $d_2 - c_2 = 13$ that bound the start time t_2 of the next execution of τ_2. By only considering the control cost, we observe that the optimal start time is 11.4. The intuition is that it is not good to schedule a task immediately after its previous execution (early start times), because the plant state has not changed much by that time. It is also not good to execute the task very late, because this leads to a longer time in which the plant is in open loop between actuations.

By only considering the CPU cost, the optimal start time is 13, which means that the execution will complete exactly at the imposed deadline—if the task experiences its worst-case execution time. As we have discussed, the objective is to consider both the control cost and CPU cost during scheduling. The two costs can be combined together with weights that are based on the required level of trade-off between control perfor-

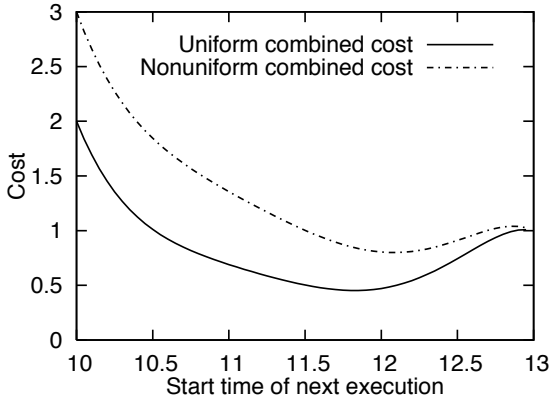

Figure 7.5: Combined control and CPU costs. Two different combinations are shown with different weights between control and CPU costs.

mance and resource usage, as well as the characteristics and temporal requirements of other applications that execute on the uniprocessor platform (e.g., best-effort applications). The solid line in Figure 7.5 shows the sum of the control and CPU cost, indicating equal importance of achieving high control quality and low CPU usage. The dashed line indicates the sum of the two costs in which the CPU cost is included twice in the summation. By considering the cost shown by the dashed line during scheduling, the start time is chosen more in favor of low CPU usage than high control performance. For the solid line, we can see that the optimal start time is 11.8, whereas it is 12.1 for the dashed line. For systems with multiple control tasks, the best start time in each case might be in conflict with an already scheduled execution of another control task (e.g., with task τ_3 in Figure 7.3). In such cases, the scheduler can decide to move an already scheduled execution, if this degradation of control performance and resource usage for that execution is considered affordable.

7.4 Problem Formulation

We shall in this section present the specification and objective of the runtime-scheduler component in Figure 7.1. The two following subsections present the scheduling constraints that are present at runtime, as

well as the optimization objectives of the scheduler.

7.4.1 Scheduling Constraints

Let us first define non-preemptive scheduling of a task set $\mathbf{T}$ with index set $\mathcal{I}_{\mathbf{T}}$. We shall consider that each task $\tau_i \in \mathbf{T}$ ($i \in \mathcal{I}_{\mathbf{T}}$) has a worst-case execution time c_i and an absolute deadline $d_i = \phi_i + D_i$, where D_i is computed by the second execution segment of the control task and is the deadline relative to the completion time ϕ_i of the task (Section 7.2). A schedule of the task set $\mathbf{T}$ under consideration is an assignment of the start time t_i of the execution of each task $\tau_i \in \mathbf{T}$ such that there exists a bijection $\sigma : \{1, \ldots, |\mathbf{T}|\} \longrightarrow \mathcal{I}_{\mathbf{T}}$ that satisfies the following properties:

$$t_{\sigma(k)} + c_{\sigma(k)} \leqslant d_{\sigma(k)} \qquad \text{for } k \in \{1, \ldots, |\mathbf{T}|\} \tag{7.3}$$

$$t_{\sigma(k)} + c_{\sigma(k)} \leqslant t_{\sigma(k+1)} \qquad \text{for } k \in \{1, \ldots, |\mathbf{T}| - 1\} \tag{7.4}$$

The bijection σ gives the order of execution of the task set $\mathbf{T}$: The tasks are executed in the order $\tau_{\sigma(1)}, \ldots, \tau_{\sigma(|\mathbf{T}|)}$. Thus, task τ_i starts its execution at time t_i and is preceded by executions of $\sigma^{-1}(i) - 1$ tasks. Equation 7.3 models that the start times are chosen such that each task execution meets its imposed deadline. Equation 7.4 models that the scheduled task executions do not overlap in time—that is, the CPU can execute at most one task at any time instant.

Having introduced the scheduling constraints, let us proceed with the problem definition. The initial schedule (the schedule at time zero) of the set of control tasks $\mathbf{T}$ is given and determined offline. At runtime, when a task completes its execution, the scheduler is activated to schedule the next execution of that task by considering its deadline and the trade-off between control quality and resource usage. Thus, when a task $\tau_i \in \mathbf{T}$ completes at time ϕ_i, we have at that time a schedule for the task set $\mathbf{T}' = \mathbf{T} \setminus \{\tau_i\}$ with index set $\mathcal{I}_{\mathbf{T}'} = \mathcal{I}_{\mathbf{T}} \setminus \{i\}$. This means that we have a bijection $\sigma' : \{1, \ldots, |\mathbf{T}'|\} \longrightarrow \mathcal{I}_{\mathbf{T}'}$ and an assignment of the start times

$$\{t_j\}_{j \in \mathcal{I}_{\mathbf{T}'}}$$

such that

$$\phi_i \leqslant t_{\sigma'(1)}$$

and that Equations 7.3 and 7.4 hold, with $\mathbf{T}$ replaced by $\mathbf{T}'$. At time ϕ_i, task τ_i has a new deadline d_i and the scheduler must decide the start

time t_i of the next execution of τ_i to obtain a schedule for the entire set of control tasks $\mathbf{T}$. The scheduler is allowed to change the current order and start times of the already scheduled tasks $\mathbf{T}'$. Thus, after scheduling, each task $\tau_j \in \mathbf{T}$ has a start time $t_j \geqslant \phi_i$ such that all start times constitute a schedule for $\mathbf{T}$, according to Equations 7.3 and 7.4. The next subsection presents the optimization objectives that are taken into consideration when determining the start time of a task.

7.4.2 Optimization Objective

The optimization objective of the scheduler at runtime is twofold: to minimize the control cost (a small cost indicates high control quality) and to minimize the CPU cost (the CPU cost indicates the CPU usage of the control tasks). Let us recall that ϕ_i is the completion time of task τ_i and d_i is the deadline of the next execution of τ_i. Because execution is non-preemptive and the task must complete before its deadline, the start time t_i is allowed to be at most $d_i - c_i$. Let us therefore define the control and CPU costs for start times of task τ_i in the time interval $[\phi_i, d_i - c_i]$. Thereafter, we shall define the overall cost to be minimized by the scheduler.

State Cost The *state cost* in the considered time interval $[\phi_i, d_i - c_i]$ is defined as

$$J_i^{\mathrm{x}}(t_i) = \int_{\phi_i}^{d_i} \boldsymbol{x}_i^{\mathsf{T}}(t) Q_i \boldsymbol{x}_i(t) dt, \tag{7.5}$$

where $t_i \in [\phi_i, d_i - c_i]$ is the start time of the next execution of τ_i. The weight matrix Q_i is used to assign weights to the individual state components in $\boldsymbol{x}_i$. It can also be used to transform the cost to a common baseline or to specify importance relative to other control loops. Note that a small cost indicates high control performance, and vice versa. Unlike the stationary control cost defined in Equation 3.3 on page 28, the state cost in Equation 7.5 is defined in terms of a finite time interval considering only the next execution of task τ_i. The scheduler thus takes decisions based on the state cost defined in terms of the next execution of each task. Future deadlines and resource requirements are not known and depend on future states. These are considered in future scheduling points of each task.

The dependence of the state cost on the start time t_i is implicit in Equation 7.5: The start time decides the time when the control signal

is updated and thus affects the dynamics of the plant state $\boldsymbol{x}_i$ according to Equation 7.1. In some control problems (e.g., when computing the actual state-feedback law in Equation 7.2), the cost in Equation 7.5 also includes a term penalizing the controlled input $\boldsymbol{u}_i$. We do not include this term, because the input is determined uniquely by the state through the given control law $\boldsymbol{u}_i = K_i \boldsymbol{x}_i$. The design of the actual control law, however, typically addresses both the state and the input costs.

Control Cost Let us denote the minimum and maximum value of the state cost J_i^{x} in the time interval $[\phi_i, d_i - c_i]$ with $J_i^{\mathrm{x,min}}$ and $J_i^{\mathrm{x,max}}$, respectively. We define the finite-horizon *control cost*

$$J_i^{\mathrm{c}} : [\phi_i, d_i - c_i] \longrightarrow [0, 1]$$

as

$$J_i^{\mathrm{c}}(t_i) = \frac{J_i^{\mathrm{x}}(t_i) - J_i^{\mathrm{x,min}}}{J_i^{\mathrm{x,max}} - J_i^{\mathrm{x,min}}}. \tag{7.6}$$

Note that this is a function from $[\phi_i, d_i - c_i]$ to $[0, 1]$, where 0 and 1, respectively, indicate the best and worst possible control performance in the considered time interval.

CPU Cost The CPU cost $J_i^{\mathrm{r}} : [\phi_i, d_i - c_i] \longrightarrow [0, 1]$ for task τ_i is defined as the linear cost

$$J_i^{\mathrm{r}}(t_i) = \frac{d_i - c_i - t_i}{d_i - c_i - \phi_i}, \tag{7.7}$$

which models a linear decrease between a CPU cost of 1 at $t_i = \phi_i$ and a cost of 0 at the latest possible start time $t_i = d_i - c_i$. The intuition is that the CPU load can be decreased by postponing the next task execution. An example of the control and CPU costs is shown in Figure 7.4, which we discussed in the example in Section 7.3.

Overall Trade-Off There are many different possibilities to achieve a trade-off between control performance and CPU usage of the control tasks. Specifically, we define the cost $J_i(t_i)$ of the task τ_i under scheduling as a linear combination of the control and CPU costs according to

$$J_i(t_i) = J_i^{\mathrm{c}}(t_i) + \rho J_i^{\mathrm{r}}(t_i), \tag{7.8}$$

where $\rho \geqslant 0$ is a design parameter that is chosen offline to specify the required trade-off between achieving a low control cost versus reducing the CPU usage. For example, by studying Figure 7.5 again, we observe that the solid line shows the sum of the control and CPU costs in Figure 7.4 with $\rho = 1$. The dashed line shows the case for $\rho = 2$.

At each scheduling point (e.g., at time ϕ_i when task τ_i has completed its execution), the optimization goal is to minimize the overall cost of all control tasks; note that already scheduled executions may be moved in time. The cost to be minimized is defined as

$$J = \sum_{j \in \mathcal{I}_{\mathbf{T}}} J_j(t_j), \tag{7.9}$$

which models the cumulative control and CPU cost of the task set $\mathbf{T}$ at a given scheduling point.

7.5 Design Activities

To reduce the time complexity of the scheduling component, two main activities are performed at design time. The first aims to reduce the complexity of computing the state cost in Equation 7.5 at runtime. This is addressed by constructing approximate cost functions, which can be evaluated efficiently at runtime during optimization. The second activity is to verify that the platform has sufficient computation capacity to achieve stability of the control loops in all possible execution scenarios. Given these two steps, we shall discuss the scheduler component in Section 7.6.

7.5.1 Cost-Function Approximation

We consider that a task τ_i has completed its execution at time ϕ_i at which its next execution is to be scheduled and completed before its imposed deadline d_i. Thus, the start time t_i must be chosen in the time interval $[\phi_i, d_i - c_i]$. The most recent known state is $\boldsymbol{x}_{i,0} = \boldsymbol{x}_i(t'_i)$, where t'_i is the start time of the just completed execution of τ_i. The control signal has been updated by the task according to the control law $\boldsymbol{u}_i = K_i \boldsymbol{x}_i$ (Section 7.1). By solving the differential equation in Equation 7.1 with the theory presented by Åström and Wittenmark [ÅW97], we can describe the cost in Equation 7.5 as

$$J_i^{\mathrm{x}}(\phi_i, t_i) = \boldsymbol{x}_{i,0}^{\mathsf{T}} M_i(\phi_i, t_i) \boldsymbol{x}_{i,0}.$$

The matrix M_i includes matrix exponentials and integrals and is decided by the plant, controller, and cost parameters. It further depends on the difference $d_i - \phi_i$, which is bounded by $D_i^{\min}$ and $D_i^{\max}$ (Section 7.2). Each element in $M_i(\phi_i, t_i)$ is a function of the completion time ϕ_i of task τ_i and the start time $t_i \in [\phi_i, d_i - c_i]$ of the next execution of τ_i. An important characteristic of M_i is that it depends only on the difference $t_i - \phi_i$.

Due to time complexity, the computation of the matrix $M_i(\phi_i, t_i)$ is impractical to perform at runtime. To cope with this complexity, our approach is to use an approximation $\widehat{M}_i(\phi_i, t_i)$ of $M_i(\phi_i, t_i)$. The scheduler presented in Section 7.6 shall thus consider the approximate state cost

$$\widehat{J}_i^{\mathrm{x}}(t_i) = \boldsymbol{x}_{i,0}^{\mathsf{T}} \widehat{M}_i(\phi_i, t_i) \boldsymbol{x}_{i,0} \tag{7.10}$$

in the optimization process. The approximation of $M_i(\phi_i, t_i)$ is done at design time by computing M_i for a number of values of the difference $d_i - \phi_i$. The matrix $M_i(\phi_i, t_i)$, which depends only on the difference $t_i - \phi_i$, is computed for equidistant values of $t_i - \phi_i$ between 0 and $d_i - c_i$. The precalculated points are all stored in memory and are used at runtime to compute $\widehat{M}_i(\phi_i, t_i)$. The granularity of the approximation is a design parameter and is decided based on the required accuracy and the memory space needed to store the information required to compute $\widehat{M}_i$.

7.5.2 Verification of Computation Capacity

Before the control system is deployed, it must be made certain that stability of all control loops is guaranteed. This verification is twofold:

1. We must make sure that there is sufficient computation capacity to achieve stability.

2. We must make sure that the scheduler—in any execution scenario—finds a schedule that guarantees stability by meeting the imposed deadlines.

The first step is to verify at design time that the condition

$$\sum_{j \in \mathcal{I}_{\mathbf{T}}} c_j \leqslant \min_{j \in \mathcal{I}_{\mathbf{T}}} D_j^{\min} \tag{7.11}$$

holds. The second step, which is guaranteed by construction of the scheduler, is described in Section 7.6.3. To understand Equation 7.11, let us

consider that a task $\tau_i \in \mathbf{T}$ has finished its execution at time ϕ_i and its next execution is to be scheduled. The other tasks $\mathbf{T} \setminus \{\tau_i\}$ are already scheduled before their respective deadlines. The worst-case execution scenario is characterized by two properties:

1. The next execution of τ_i is due within its minimum deadline $D_i^{\min}$, relative to time ϕ_i. This means that $d_i = \phi_i + D_i^{\min}$.

2. Each scheduled task has its deadline within the minimum deadline of τ_i. That is, for each scheduled task $\tau_j \in \mathbf{T} \setminus \{\tau_i\}$, we have $d_j \leqslant d_i = \phi_i + D_i^{\min}$.

In this execution scenario, each task must execute exactly once within a time period of $D_i^{\min}$ (i.e., in the time interval $[\phi_i, \phi_i + D_i^{\min}]$). Equation 7.11 follows by considering that τ_i is the control task with the smallest possible relative deadline. In Section 7.6.3, we describe how the schedule is constructed to guarantee stability, provided that the property in Equation 7.11 holds.

The time overhead of the scheduler described in the next section can be bounded by computing its worst-case execution overhead at design time (this is performed with tools for worst-case execution time analysis [WEE+08]). For simplicity of presentation in Equation 7.11, we consider this overhead to be included in the worst-case execution time c_j of task τ_j. Independent of the scheduling heuristic, the test guarantees not only that all stability-related deadlines can be met at runtime but also that a minimum level of control performance is achieved. The scheduling heuristic presented in the following section improves on these minimum control-performance guarantees.

7.6 Runtime Scheduler

This section describes the scheduling heuristic that is to be implemented by the scheduler component. We shall in this section consider that task $\tau_i \in \mathbf{T}$ has completed its execution at time ϕ_i and that its next execution is to be scheduled before the computed deadline d_i. Each task $\tau_j \in \mathbf{T} \setminus \{\tau_i\}$ has already been scheduled for execution at start time $t_j \geqslant \phi_i$. These start times constitute a schedule for the task set $\mathbf{T} \setminus \{\tau_i\}$, according to the definition of a schedule in Section 7.4.1 and Equations 7.3 and 7.4. The scheduler must decide the start time t_i of the next execution of τ_i

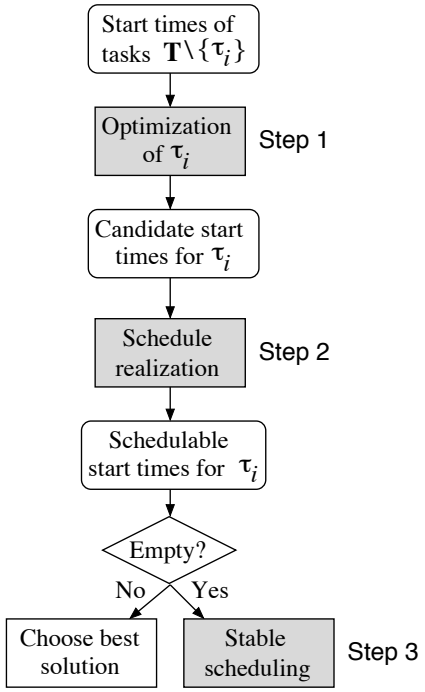

Figure 7.6: Flowchart of the scheduling heuristic. The first step finds candidate start times that, in the second step, are evaluated with regard to scheduling. If needed, the third step is executed to guarantee stability.

such that $\phi_i \leqslant t_i \leqslant d_i - c_i$, possibly changing the start times of the other task $\mathbf{T} \setminus \{\tau_i\}$. The condition is that the resulting start times $\{t_j\}_{j\in\mathcal{I}_\mathbf{T}}$ constitute a schedule for the task set $\mathbf{T}$.

Figure 7.6 shows a flowchart of our proposed scheduler. The first step is to optimize the start time t_i of the next execution of τ_i (Section 7.6.1). In this step, we do not consider the existing start times of the other tasks $\mathbf{T}\backslash\{\tau_i\}$ but merely focus on the cost J_i in Equation 7.8. Thus, in this first step, we do not consider possible execution conflicts when optimizing the schedule for task τ_i. The optimization is based on a search heuristic that results in a set of *candidate start times*

$$\mathbf{\Xi}_i = \left\{t_i^{(1)}, \ldots, t_i^{(n)}\right\} \subset [\phi_i, d_i - c_i].$$

After this step, the cost $J_i(t_i)$ has been computed for each $t_i \in \mathbf{\Xi}_i$. In

the second step (Section 7.6.2), for each $t_i \in \Xi_i$, we determine whether it is possible to schedule the execution of task τ_i at time t_i, considering the existing start time t_j of each task $\tau_j \in \mathbf{T} \setminus \{\tau_i\}$. If necessary, this may include a modification of the starting times of the already scheduled tasks to accommodate the execution of τ_i at the candidate start time t_i. If the start times cannot be modified such that all imposed deadlines are met, then task τ_i is not schedulable at the candidate start time t_i. The result of the second step (schedule realization) is a subset $\Xi'_i \subseteq \Xi_i$ of the candidate start times.

For each $t_i \in \Xi'_i$, the execution of τ_i can be accommodated at that time, possibly with a modification of the start times

$$\{t_j\}_{j \in \mathcal{I}_{\mathbf{T}} \setminus \{i\}}$$

such that the scheduling constraints in Equations 7.3 and 7.4 are satisfied for the whole task set $\mathbf{T}$. For each $t_i \in \Xi'_i$, the scheduler computes the total control and CPU cost of all control tasks (Equation 7.9). The scheduler chooses the start time $t_i \in \Xi'_i$ that gives the best overall cost. If $\Xi'_i = \emptyset$—this means that task τ_i is not schedulable at any of the candidate start times in Ξ_i—the scheduler resorts to the third step (stable scheduling), which guarantees to find a solution that meets all imposed stability-related completion deadlines (Section 7.6.3). Let us, in the following three subsections, discuss the three steps in Figure 7.6 in more detail.

7.6.1 Optimization of Start Time

As we have mentioned, in this step, we consider the minimization of the cost $J_i(t_i)$ in Equation 7.8, which is the combined control and CPU cost of task τ_i. Let us first, however, consider the approximation $\widehat{J}_i^{\mathrm{x}}(t_i)$ (Equation 7.10) of the state cost $J_i^{\mathrm{x}}(t_i)$ in Equation 7.5. The scheduler performs a minimization of this approximate cost by a golden-section search [PTVF07, GW04]. The search is iterative and maintains, in each iteration, three points

$$\omega_1, \omega_2, \omega_3 \in [\phi_i, d_i - c_i]$$

for which the cost $\widehat{J}_i^{\mathrm{x}}$ has been evaluated at previous iterations of the search. For the first iteration, the initial values of the end points are

$\omega_1 = \phi_i$ and $\omega_3 = d_i - c_i$. The middle point ω_2 is initially chosen according to the golden ratio as

$$\frac{\omega_3 - \omega_2}{\omega_2 - \omega_1} = \frac{1 + \sqrt{5}}{2}.$$

The next step is to evaluate the function value for a point ω_4 in the largest of the two intervals $[\omega_1, \omega_2]$ and $[\omega_2, \omega_3]$. This point ω_4 is chosen such that $\omega_4 - \omega_1 = \omega_3 - \omega_2$. If $\widehat{J}_i^{\mathrm{x}}(\omega_4) < \widehat{J}_i^{\mathrm{x}}(\omega_2)$, we update the three points ω_1, ω_2, and ω_3 according to

$$\begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{bmatrix} \longleftarrow \begin{bmatrix} \omega_2 \\ \omega_4 \\ \omega_3 \end{bmatrix}$$

and then repeat the golden-section search. If $\widehat{J}_i^{\mathrm{x}}(\omega_4) > \widehat{J}_i^{\mathrm{x}}(\omega_2)$, we perform the update

$$\begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{bmatrix} \longleftarrow \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_4 \end{bmatrix}$$

and proceed with the next iteration. The cost $\widehat{J}_i^{\mathrm{x}}$ is computed efficiently for each point based on the latest sampled state and the precalculated values of M_i, which are stored in memory before runtime (Section 7.5.1). The search ends after a given number of iterations. We shall consider this design parameter in the experimental evaluation.

The result of the search is a set of visited points

$$\mathbf{\Omega}_i = \left\{ t_i^{(1)}, \ldots, t_i^{(n)} \right\}$$

for which we have

$$\{\phi_i, d_i - c_i\} \subset \mathbf{\Omega}_i \subset [\phi_i, d_i - c_i].$$

The number of iterations of the search is $n - 3$. The search has evaluated $\widehat{J}_i^{\mathrm{x}}(t_i)$ for each $t_i \in \mathbf{\Omega}_i$. Let us introduce the minimum and maximum approximate state costs $\widehat{J}_i^{\mathrm{x,min}}$ and $\widehat{J}_i^{\mathrm{x,max}}$, respectively, as

$$\widehat{J}_i^{\mathrm{x,min}} = \min_{t_i \in \mathbf{\Omega}_i} \widehat{J}_i^{\mathrm{x}}(t_i) \text{ and } \widehat{J}_i^{\mathrm{x,max}} = \max_{t_i \in \mathbf{\Omega}_i} \widehat{J}_i^{\mathrm{x}}(t_i).$$

We define the approximate control cost $\widehat{J}_i^{\mathrm{c}}(t_i)$—compare to Equation 7.6 on page 124—for each $t_i \in \mathbf{\Omega}_i$ as

$$\widehat{J}_i^{\mathrm{c}}(t_i) = \frac{\widehat{J}_i^{\mathrm{x}}(t_i) - \widehat{J}_i^{\mathrm{x,min}}}{\widehat{J}_i^{\mathrm{x,max}} - \widehat{J}_i^{\mathrm{x,min}}}. \tag{7.12}$$

Let us now extend

$$\left\{ \widehat{J}_i^{\mathrm{c}}(t_i^{(1)}), \ldots, \widehat{J}_i^{\mathrm{c}}(t_i^{(n)}) \right\}$$

to define $\widehat{J}_i^{\mathrm{c}}(t_i)$ for an arbitrary $t_i \in [\phi_i, d_i - c_i]$. Without loss of generality, we assume that

$$\phi_i = t_i^{(1)} < t_i^{(2)} < \cdots < t_i^{(n)} = d_i - c_i.$$

For any $q \in \{1, \ldots, n-1\}$, we use linear interpolation in the open time interval $\left(t_i^{(q)}, t_i^{(q+1)}\right)$, resulting in

$$\widehat{J}_i^{\mathrm{c}}(t_i) = \left(1 - \frac{t_i - t_i^{(q)}}{t_i^{(q+1)} - t_i^{(q)}}\right) \widehat{J}_i^{\mathrm{c}}(t_i^{(q)}) + \frac{t_i - t_i^{(q)}}{t_i^{(q+1)} - t_i^{(q)}} \widehat{J}_i^{\mathrm{c}}(t_i^{(q+1)}) \tag{7.13}$$

for $t_i^{(q)} < t_i < t_i^{(q+1)}$. Equations 7.12 and 7.13 define, for the complete time interval $[\phi_i, d_i - c_i]$, the approximation $\widehat{J}_i^{\mathrm{c}}$ of the control cost in Equation 7.6.

We can now define the approximation $\widehat{J}_i$ of the overall cost J_i in Equation 7.8 as

$$\widehat{J}_i(t_i) = \widehat{J}_i^{\mathrm{c}}(t_i) + \rho J_i^{\mathrm{r}}(t_i).$$

To consider the twofold optimization objective of control quality and CPU usage, we perform the golden-section search in the time interval $[\phi_i, d_i - c_i]$ for the function $\widehat{J}_i(t_i)$. The cost evaluations are in this step merely based on Equations 7.12, 7.13, and 7.7, which do not involve any computations based on the sampled state or the precalculated values of M_i. This last search results in a finite set of candidate start times $\mathbf{\Xi}_i \subset [\phi_i, d_i - c_i]$ to be considered in the next step.

7.6.2 Schedule Realization

We shall consider the given start time t_j for each task $\tau_j \in \mathbf{T} \setminus \{\tau_i\}$. These start times have been chosen under the consideration of the scheduling constraints in Equations 7.3 and 7.4. We thus have a bijection

$$\sigma : \{1, \ldots, |\mathbf{T}| - 1\} \longrightarrow \mathcal{I}_{\mathbf{T}} \setminus \{i\}$$

that gives the order of execution of the task set $\mathbf{T} \setminus \{\tau_i\}$ (Section 7.4.1). We shall now describe the scheduling procedure to be performed for each candidate start time $t_i \in \mathbf{\Xi}_i$ of task τ_i obtained in the previous step. The scheduler first checks whether the execution of τ_i at the candidate start time t_i overlaps with any existing scheduled task execution. If there is an overlap, the second step is to move the existing overlapping executions forward in time. If this modification leads to a satisfaction of the timing constraints (Equation 7.3), or if no overlapping execution was found at all, this start time is declared schedulable. We shall in the remainder of this section provide detailed descriptions of the steps to be performed for each candidate start time.

Let us consider a candidate start time $t_i \in \mathbf{\Xi}_i$ and discuss how to identify and move overlapping executions of $\mathbf{T} \setminus \{\tau_i\}$. The idea is to identify the first overlapping execution, considering that τ_i starts its execution at t_i. If such an overlap exists, the overlapping execution and its successive executions are pushed forward in time by the minimum amount of time that is required to schedule τ_i at time t_i and to satisfy the scheduling constraint in Equation 7.4 for the entire task set $\mathbf{T}$. To find the first overlapping execution, the scheduler searches for the smallest $k \in \{1, \ldots, |\mathbf{T}| - 1\}$ for which

$$\left[t_{\sigma(k)}, t_{\sigma(k)} + c_{\sigma(k)}\right] \cap [t_i, t_i + c_i] \neq \emptyset. \tag{7.14}$$

If no overlapping execution is found, meaning that Equation 7.14 is not satisfied for any $k \in \{1, \ldots, |\mathbf{T}| - 1\}$, the candidate start time t_i is declared *schedulable*. In this case, the execution of τ_i can be scheduled at time t_i without any modification of the schedule of $\mathbf{T} \setminus \{\tau_i\}$. If, on the other hand, an overlap is found, we have the smallest k that satisfies Equation 7.14. The schedule is modified in that case as follows; note that the schedule of the task set $\{\tau_{\sigma(1)}, \ldots, \tau_{\sigma(k-1)}\}$ is not modified. First, the scheduler computes the minimum amount of time

$$\Delta = t_i + c_i - t_{\sigma(k)} \tag{7.15}$$

to shift the execution of $\tau_{\sigma(k)}$ forward. The new start time of $\tau_{\sigma(k)}$ is thus

$$t'_{\sigma(k)} = t_{\sigma(k)} + \Delta. \tag{7.16}$$

This modification can introduce new overlapping executions or change the order of the schedule. To avoid this situation, we consider the successive executions $\tau_{\sigma(k+1)}, \ldots, \tau_{\sigma(|\mathbf{T}|-1)}$ by iteratively computing a new

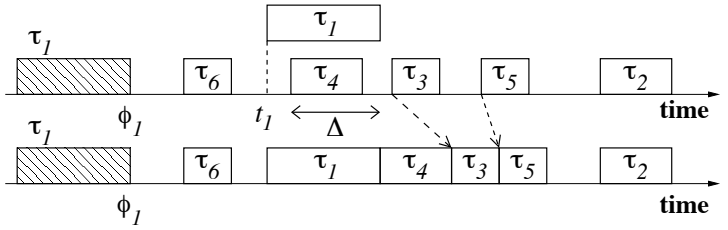

Figure 7.7: Schedule realization. The upper schedule shows a candidate start time t_1 for τ_1 that is in conflict with the existing schedule. The conflict is solved by pushing the current schedule forward in time by an amount Δ, resulting in the schedule shown in the lower part.

start time $t'_{\sigma(q)}$ for task $\tau_{\sigma(q)}$ according to

$$t'_{\sigma(q)} = \max\left(t_{\sigma(q)}, t'_{\sigma(q-1)} + c_{\sigma(q-1)}\right), \tag{7.17}$$

where q ranges from $k+1$ to $|\mathbf{T}|-1$ in increasing order. The maximum value is taken over the current start time $t_{\sigma(q)}$ of task $\tau_{\sigma(q)}$ and the completion time that task $\tau_{\sigma(q-1)}$ will have if it executes at the modified starting time $t'_{\sigma(q-1)}$. Note that the iteration can be stopped at the first q for which $t_{\sigma(q)} = t'_{\sigma(q)}$. The reason is that the schedule for the task set $\{\tau_{\sigma(q)}, \ldots, \tau_{\sigma(|\mathbf{T}|-1)}\}$ is unchanged by the modification process given by Equation 7.17. The candidate start time t_i is declared schedulable if, after the updates in Equations 7.16 and 7.17, we have

$$t'_{\sigma(q)} + c_{\sigma(q)} \leqslant d_{\sigma(q)}$$

for each $q \in \{k, \ldots, |\mathbf{T}|-1\}$. We denote the set of *schedulable candidate start times* with $\mathbf{\Xi}'_i$.

Let us study Figure 7.7 and discuss how Equations 7.16 and 7.17 are used to schedule a task τ_1 for a given candidate start time t_1. The scheduling is done at time ϕ_1 at which the execution of the tasks $\tau_2, \ldots, \tau_6$ are already scheduled. The upper chart in the figure shows that the candidate start time t_1 is in conflict with the scheduled execution of τ_4. In the lower chart, it is shown that the scheduler has used Equation 7.16 to move τ_4 forward by Δ. This is indicated in the figure and computed with Equation 7.15 to $\Delta = t_4 + c_4 - t_1$. Tasks τ_3 and τ_5 are moved iteratively according to Equation 7.17 by an amount less than or equal to Δ. Task τ_2

is not affected because the change in execution of τ_4 does not introduce an execution overlap with τ_2.

Let us consider that $\Xi_i' \neq \emptyset$, which means that at least one candidate start time in Ξ_i is schedulable. For each schedulable candidate start time $t_i \in \Xi_i'$, we shall associate a cost $\Psi_i(t_i)$ that represents the overall cost (Equation 7.9) of scheduling τ_i at time t_i and possibly moving other tasks according to Equations 7.16 and 7.17. This cost is defined as

$$\Psi_i(t_i) = \sum_{q=1}^{k-1} \widehat{J}_{\sigma(q)}(t_{\sigma(q)}) + \widehat{J}_i(t_i) + \sum_{q=k}^{|\mathbf{T}|-1} \widehat{J}_{\sigma(q)}(t'_{\sigma(q)}),$$

where the notation and new start times $t'_{\sigma(q)}$ are the same as our discussion around Equations 7.16 and 7.17. The costs in the first summation have already been computed at previous scheduling points, because the start time of each task $\tau_{\sigma(q)}$ is not modified ($1 \leqslant q < k$). Further, the cost $\widehat{J}_i(t_i)$ has been computed at the current scheduling point (Section 7.6.1). Finally, each cost $\widehat{J}_{\sigma(q)}(t'_{\sigma(q)})$ in the last summation can be computed efficiently, because the scheduler has—at a previous scheduling point—already performed the optimizations in Section 7.6.1 for each task $\tau_{\sigma(q)} \in \mathbf{T} \setminus \{\tau_i\}$.

The final solution chosen by the scheduler is the best schedulable candidate start time in terms of the cost $\Psi_i(t_i)$. The scheduler thus assigns the start time t_i of task τ_i to any start time in

$$\arg\min_{t \in \Xi_i'} \Psi_i(t).$$

If an overlapping execution exists, its start time and the start times of its subsequent executions are updated according to Equations 7.16 and 7.17. In that case, the update

$$t_{\sigma(q)} \longleftarrow t'_{\sigma(q)}$$

is made iteratively from $q = k$ to $q = |\mathbf{T}| - 1$ in increasing order, where $\tau_{\sigma(k)}$ is the first overlapping execution according to Equation 7.14 and $t'_{\sigma(q)}$ is given by Equations 7.16 and 7.17. We have now completed our discussion to choose a start time t_i for task τ_i in the set of schedulable candidate start times Ξ_i'. However, at runtime, it can be the case that none of the candidate start times Ξ_i from the first step (Section 7.6.1) are schedulable. Thus, if $\Xi_i' = \emptyset$, the scheduler must find an assignment

of start times such that all tasks complete their execution before or at the deadlines, which are imposed by the self-triggered control tasks based on requirements on stability and a minimum level of control performance. Such a procedure is described in the next subsection.

7.6.3 Stability Guarantee

The scheduling and optimization step in Sections 7.6.1 and 7.6.2 can fail to find a valid schedule for the task set $\mathbf{T}$. To ensure stability of the control system in such cases, the scheduler must find a schedule that meets the imposed deadlines, without necessarily considering any optimization of control performance and resource usage. Thus, the scheduler is allowed in such critical situations to use the full computation capacity in order to meet the timing constraints imposed by the self-triggered control tasks. Let us describe how to construct such a schedule at an arbitrary scheduling point.

At a given time instant ϕ_i, task τ_i has completed its execution and its next execution must be scheduled to complete before the deadline d_i. A schedule for the other tasks $\mathbf{T} \setminus \{\tau_i\}$ exists at time ϕ_i. We thus have a bijection

$$\sigma : \{1, \ldots, |\mathbf{T}| - 1\} \longrightarrow \mathcal{I}_{\mathbf{T}} \setminus \{i\}$$

and a start time $t_{\sigma(q)}$ for the next execution of task $\tau_{\sigma(q)} \in \mathbf{T} \setminus \{\tau_i\}$ ($1 \leqslant q < |\mathbf{T}|$). Because the start time of a task cannot be smaller than the completion time of its preceding task in the schedule (Equation 7.4), we have

$$t_{\sigma(k)} \geqslant \phi_i + \sum_{q=1}^{k-1} c_{\sigma(q)} \tag{7.18}$$

for $1 < k < |\mathbf{T}|$ and $t_{\sigma(1)} \geqslant \phi_i$ for the first task $\tau_{\sigma(1)}$ in the schedule. In Equation 7.18, the sum models the cumulative worst-case execution time of the $k-1$ executions that precede task $\tau_{\sigma(k)}$. Note that the deadline constraints (Equation 7.3 on page 122) for the task set $\mathbf{T} \setminus \{\tau_i\}$ are satisfied, considering the given start times. Important also to highlight is that the deadline of a task $\tau_j \in \mathbf{T} \setminus \{\tau_i\}$ is not violated by scheduling its execution earlier than the assigned start time t_j. To accommodate the execution of the task under scheduling τ_i, we shall thus modify the existing start times for the task set $\mathbf{T} \setminus \{\tau_i\}$ to achieve equality in Equation 7.18.

For the first task, the scheduler performs the assignment

$$t_{\sigma(1)} \longleftarrow \phi_i, \tag{7.19}$$

and for each task $\tau_{\sigma(k)}$ $(1 < k < |\mathbf{T}|)$, it assigns

$$t_{\sigma(k)} \longleftarrow \phi_i + \sum_{q=1}^{k-1} c_{\sigma(q)}. \tag{7.20}$$

As we have discussed, these modifications do not violate the timing constraints for the task set $\mathbf{T} \setminus \{\tau_i\}$. The start time t_i of task τ_i is assigned as

$$t_i \longleftarrow \phi_i + \sum_{q=1}^{|\mathbf{T}|-1} c_{\sigma(q)}. \tag{7.21}$$

This completes the schedule for $\mathbf{T}$. With this assignment of start times, the worst-case completion time of τ_i is

$$t_i + c_i = \phi_i + \sum_{q=1}^{|\mathbf{T}|-1} c_{\sigma(q)} + c_i = \phi_i + \sum_{j \in \mathcal{I}_{\mathbf{T}}} c_j,$$

which, if Equation 7.11 on page 126 holds, is smaller than or equal to any possible deadline d_i for τ_i, because

$$t_i + c_i = \phi_i + \sum_{j \in \mathcal{I}_{\mathbf{T}}} c_j \leqslant \phi_i + D_i^{\min} \leqslant d_i.$$

With Equations 7.19–7.21, provided that Equation 7.11 holds (to be verified at design time), the scheduler can meet all deadlines in any execution scenario.

Let us consider Figure 7.8 to illustrate the scheduling policy given by Equations 7.19–7.21. In the schedule on the left side, task τ_2 completes its execution at time ϕ_2 and the scheduler must find a placement of the next execution of τ_2 such that it completes before its imposed deadline d_2. Tasks τ_1 and τ_3 are already scheduled to execute at times t_1 and t_3, respectively, such that the deadlines d_1 and d_3 are met. In the schedule on the right side, it is shown that the executions of τ_1 and τ_3 are moved towards earlier start times (Equations 7.19 and 7.20) to accommodate the execution of τ_2. Since the deadlines already have been met by the start

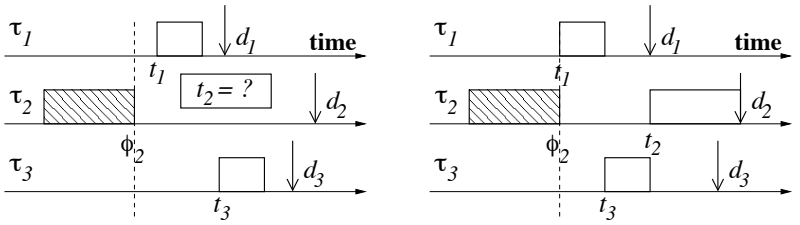

Figure 7.8: Stable scheduling. The left schedule shows a scenario in which, at time ϕ_2, the scheduler must accommodate CPU time to the next execution of τ_2. In the right schedule, CPU time for this execution is accommodated by moving the scheduled executions of τ_1 and τ_3 to earlier start times.

times in the left schedule, this change in start times t_1 and t_3 does not violate the imposed deadlines of τ_1 and τ_3, since the order of the two tasks is preserved. Task τ_2 is then scheduled immediately after τ_3 (Equation 7.21) and its deadline is met, provided that Equation 7.11 holds.

7.7 Experimental Results

We have evaluated the proposed scheduling heuristic with simulations of 50 systems comprising 2 to 5 control tasks that control plants with given initial conditions of the state equations in Equation 7.1. We have run experiments for several values of the design constant ρ in Equation 7.8 (the trade-off between control quality and CPU usage) in order to obtain simulations with different amounts of CPU usage. For each simulation, we computed the total control cost of the entire task set $\mathbf{T}$ as

$$J^{\text{c,sim}} = \sum_{j \in \mathcal{I}_{\mathbf{T}}} \int_0^{t^{\text{sim}}} \boldsymbol{x}_j^{\mathsf{T}}(t) Q_j \boldsymbol{x}_j(t) dt, \tag{7.22}$$

where t^{sim} is the amount of simulated time. This cost represents the control performance during the whole simulated time interval; a low cost $J^{\text{c,sim}}$ indicates high control performance. For each experiment, we recorded the amount of CPU usage of all control tasks, including the time overhead of the scheduling heuristic; the time overhead was estimated based on its runtime on the host machine of the experiments. The baseline of comparison is a periodic implementation for which the pe-

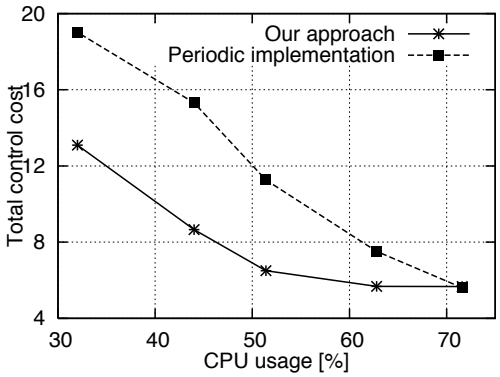

Figure 7.9: Scaling of the control cost. Our approach is compared to a periodic implementation for different CPU-usage levels. Periodic control uses more CPU bandwidth to achieve the same level of control performance as our approach with reduced CPU usage.

riods are chosen to achieve the measured CPU usage. For this periodic implementation, we computed the corresponding total control cost $J_{\text{per}}^{\text{c,sim}}$ in Equation 7.22.

Figure 7.9 shows on the vertical axis the total control costs $J^{\text{c,sim}}$ and $J_{\text{per}}^{\text{c,sim}}$ for our scheduling approach and a periodic implementation, respectively. On the horizontal axis, we show the corresponding CPU usage. The main message conveyed by the results in Figure 7.9 is that the self-triggered implementation with our proposed scheduling approach results in a smaller total control cost (i.e., better control performance) compared to a periodic implementation that uses the same amount of CPU time. The designer can regulate the CPU usage of the control tasks within a wide range (30 to 60 percent of CPU usage) to obtain solutions with better control performance, compared to solutions based on periodic control. For example, for a CPU usage of 44 percent, the total control costs of our approach and a periodic implementation are 8.7 and 15.3, respectively (in this case, our approach improves the control performance by 43 percent, relative to the periodic implementation). The average cost reduction

$$\frac{J_{\text{per}}^{\text{c,sim}} - J^{\text{c,sim}}}{J_{\text{per}}^{\text{c,sim}}}$$

of our approach, relative to the periodic implementation, is 41 percent for the experiments with 30 to 60 percent of CPU usage. Note that for high levels of CPU usage in Figure 7.9, a periodic implementation samples and actuates the controlled plants very often, which in turns leads to similar control performance as a self-triggered implementation.

The time overhead has been included in the simulations by scaling the measured execution time of the scheduling heuristic relative to the execution times of the control tasks. The main parameter that decides the time overhead of the scheduler is the number of iterations to be implemented by the golden-section search in Section 7.6.1. Based on experiments, we have found that a relatively small number of iterations are sufficient to achieve good results in terms of our two optimization objectives; the experiments have been conducted with four iterations of the golden-section search. The results presented in this section show that the proposed solution outperforms a periodic solution in terms of control performance and CPU usage.

7.8 Summary and Discussion

We presented a framework for dynamic scheduling of multiple control tasks on uniprocessor platforms. The self-triggered control tasks compute their CPU needs at runtime and are scheduled with the objective of finding trade-offs between control quality and CPU usage. The solution in this chapter is based on runtime heuristics aided by design-time approximation and verification. Our results show that high control performance can be achieved with reduced CPU usage.

The weight between control cost and CPU cost has been considered as constant. However, this weight is a parameter that—instead of treating it as fixed—can be adjusted at runtime to achieve high control performance at a certain level of CPU usage. A typical scenario and motivation for nonperiodic control approaches is in the context of resource sharing among multiple control applications and a best-effort partition comprising several applications with varying resource demand. In this context, feedback control-based approaches for resource management in real-time systems [LSST02, ASL+03, HDPT04, AHSG07, BBE+11] can, for example, be used to adjust the weight between control and CPU cost based on the desired level of CPU usage of the best-effort applications that are

running on the same platform as the self-triggered control tasks.

8 Conclusions

SYNTHESIS and control-quality optimization can be performed efficiently for embedded control systems with multiple operation modes, computation components with susceptibility to permanent faults, and multiple control applications with state-triggered execution in resource-constrained environments. The complex temporal properties of the execution and communication in modern embedded control systems are acknowledged as major factors to consider during control synthesis, computer-systems design, and control-quality optimization in an integrated manner. In Chapter 4, we presented such a synthesis and optimization framework for multiple control applications on distributed embedded platforms. The framework supports several scheduling policies (static cyclic scheduling and priority-based scheduling) and communication protocols (TTP, CAN, and FlexRay) of the underlying execution platform.

Embedded control systems must be able to react and adapt to various changes in the operational environment at runtime. Such requirements lead to scaling problems because of the very large number of situations that may occur at runtime. In Chapters 5–7, we have approached the synthesis problem for adaptive embedded systems by optimizations and approximations to be performed at design time, as well as efficient solu-

tions for decisions to be taken at runtime as a response to several sources of runtime variability. We have taken several approaches to the design of adaptive embedded systems for control applications, motivated by the presence of multiple operation modes, probabilities of component failures, and varying requirements on the level of resource usage at runtime.

Chapter 5 presents a problem formulation and solution for the synthesis of embedded control systems with multiple operation modes. The main design and optimization difficulty is raised by the very large number of modes that the system may switch to at runtime. We have developed heuristics for trade-offs between several competing objectives: control quality, optimization time, and the platform memory needed to store design solutions for multiple operation modes. In Chapter 6, we considered a similar problem as in Chapter 5 for systems in which computation nodes may fail at runtime. A similar design difficulty, although one that must be approached differently than in Chapter 5, is due to the large number of combinations of operational computation components at runtime. The main objective is to find optimized solutions for fault-tolerant execution of the control applications for a set of fault scenarios. The approach taken in Chapter 6 is based on synthesis of solutions for a limited number of situations that may occur at runtime due to permanent faults, as well as trade-offs between task migration and memory space.

Periodic control is the underlying execution model of control applications considered in Chapters 4–6. In Chapter 7, we considered self-triggered control as an alternative execution model. The main advantage of self-triggered control is its ability to reduce the number of controller executions compared to periodic implementations. This is motivated in any environment in which tight constraints are imposed on the amount of sensor measurements, actuations, and resource usage related to computation and communication. For example, this is the case in systems where control applications coexist with other application tasks with varying resource demand, or when the requirements on energy consumption are changing at runtime. We proposed a scheduling component for runtime-management of computation resources shared by multiple self-triggered control applications. The solution is based on design-time approximations and a runtime-scheduling heuristic.

In the remainder of this chapter, we shall summarize the research contributions and conclusions separately for the problems and solutions presented in this thesis. The chapter organization follows the order in

which the contributions are presented throughout the thesis.

8.1 Integrated Control and Scheduling

In Chapter 4, we proposed a system-level design approach for integrated period assignment, controller synthesis, and scheduling of tasks and messages on distributed embedded systems. The construction of design solutions is an optimization process that is driven by control quality. We considered several decision variables that affect the performance of the multiple control loops in the system. These variables are the execution period of each application and its control law. Further, the periods and the scheduling of tasks and messages on the individual computation and communication components in the system influence the control quality through the delay characteristics. This is the key issue that has been considered in the integrated design framework.

Our solution to the problem of integrated control and scheduling for distributed embedded systems is based on the integration of several optimization methods. We used genetic algorithms for period and priority assignment to the tasks and messages constituting the control applications. For the case of static cyclic scheduling of tasks and messages, we used a framework for constraint logic programming and a cost function that combines average sensor–actuator delays and their variance. Controller synthesis with delay compensation is integrated with the solutions for period assignment and scheduling. Experimental results have demonstrated significant quality improvements when using the synthesis framework in Chapter 4, instead of a traditional approach for which control design and computer-systems optimization are separated.

8.2 Multi-Mode Control Systems

Multiple operation modes of distributed embedded control systems is the topic of Chapter 5. We considered that the set of controlled plants may change during operation. To achieve optimal usage of the computation and communication resources of the underlying execution platform, customized solutions (schedules and controllers) must be synthesized for each possible operation mode at design time. The large and complex design space, which is an inherent property for design problems of adap-

tive embedded systems, leads to long design times and requirements on large amount of platform memory, if optimal solutions are required to be implemented.

We proposed techniques to find appropriate trade-offs among several competing parameters: control quality, design time, and the amount of memory required to store the design solutions generated by our framework. The techniques are based on the synthesis of a set of mandatory solutions for a proper operation of the system in all possible functional modes, as well as an optimization process that produces additional, incremental improvements of the overall control quality of the system. As a last step of the optimization process, for the case of limited memory space of the execution platform, we formulated an integer linear program to be solved at design time. Experimental results have validated that high-quality solutions for multi-mode control systems can be synthesized efficiently. If the synthesis problem for multi-mode systems has very tight memory constraints, we have demonstrated that virtual modes, which do not occur at runtime, can be considered during synthesis to generate design solutions that cover several functional modes in a memory-efficient manner.

8.3 System Synthesis and Permanent Faults

The work in Chapter 6 is motivated by the need to tolerate and adapt to component faults at runtime. We considered situations that occur at runtime as a consequence of failed computation nodes in the underlying distributed execution platform. The design space is complex because of the many different sets of failed components at runtime. To give guarantees on fault tolerance, we considered the mandatory synthesis of a set of minimal configurations of the system. By supporting execution in such configurations, the system is able to operate in a large set of additional configurations that arise when computation nodes fail. Solutions for minimal configurations enable the system to be resilient to many scenarios with failed computation nodes by operating with a minimum level of control quality.

To improve control quality relative to the minimum level of quality given by the solutions for minimal configurations, we proposed an optimization approach that considers additional configurations in an in-

cremental manner. The search order is based on probabilities that are associated to each configuration. We have also elaborated on the trade-off between migration time and memory usage for the implementation of design solutions on the execution platform. This trade-off between task migration and task replication is formulated as an integer linear program and is solved efficiently with negligible time overhead. Supported by the presented experimental results, we conclude that we can construct design solutions efficiently to guarantee fault-tolerant operation at runtime. We have also demonstrated that significant quality improvements can be achieved with relatively little additional design time. In addition to this design time, a limiting factor for quality improvement is the memory space that is available for task replication and storage of design solutions.

8.4 Self-Triggered Control Applications

Self-triggered control is an execution paradigm that has been developed with the main motivation to decrease the number of controller executions compared to traditional periodic control. Nevertheless, there has been little work on the deployment of multiple such control applications on a shared computation platform. In Chapter 7, we discussed scheduling of the execution of multiple self-triggered control tasks on a uniprocessor platform. Self-triggered control tasks compute deadlines for their future executions based on the sampled plant states. Therefore, scheduling decisions for task execution need to be done at runtime. Further, the need for optimization and trade-off between control quality and CPU usage raises additional difficulties when developing efficient scheduling policies.

The scheduling approach presented in Chapter 7 is based on design-time approximations to avoid time-consuming computations at runtime. At runtime, the scheduling policy is based on a fast search procedure and a heuristic to solve conflicts in task executions. The scheduling decisions are guided by control quality and CPU usage. We have also discussed a complementary scheduling policy to guarantee that stability and a minimum level of control performance is maintained during system operation. Experimental results have demonstrated that control quality and resource usage can be optimized for multiple self-triggered control applications with low time overhead.

9 Future Work

THERE are certainly several directions for future studies and research in the area of integrated control and computing. This chapter outlines possible research directions based on the material presented in this thesis. We shall discuss two main categories of future work: First, we shall outline several directions related to integrated communication synthesis and control in Section 9.1. Second, in Section 9.2, we shall motivate and discuss the integration of embedded real-time computing with event-based and self-triggered control.

9.1 Communication Synthesis and Control

We have in this thesis considered schedule tables and priorities as decision variables in the integrated optimization frameworks for distributed control systems. This is applicable to systems with TTP, CAN, and FlexRay. We have not, however, considered any further details regarding bus-access configuration for these particular communication protocols. FlexRay is a hybrid communication protocol that comprises a static and a dynamic communication phase, each with its own configuration parameters. There are thus several parameters that affect temporal behavior and control performance. The optimization of these, integrated

with system-level scheduling and control synthesis, would be an important research contribution. Its practical relevance is exemplified by the increased use of FlexRay in the automotive systems domain. A first attempt, considering only frame identifiers during optimization, has been published recently [SEPC11]. Several other parameters must be considered before efficient solutions can be implemented. Such parameters are the lengths of the static and dynamic phases, slot sizes, the minislot length, and the mapping of frame identifiers to messages and computation nodes. This research direction could lead to better exploitation and utilization of the underlying communication infrastructure, as well as a tighter integration of real-time communication and control for improved control quality. Last, multiple control-performance metrics could be considered during design-space exploration for embedded control systems. Voit et al. [VSG+10] have taken a step in this direction by considering performance metrics related to transient and steady-state control performance, as well as the delay in the control loop. This can be important to consider in the context of joint synthesis and optimization of multiple types of embedded feedback-control applications.

Another research direction related to communication and control may be taken for systems where the communication infrastructure leads to occasional corruption or loss of transmitted data (e.g., due to transient faults in communication links or the inherent characteristics of wireless control). Goswami et al. [GSC11a] presented communication synthesis for FlexRay-based embedded control systems with guarantees on stability based on the ratio between successful and lost samples. Fundamental theory of control over lossy networks [SSF+07], as well as recent research results on wireless networked control in the context of data-corrupting channels [SCV+11] and packet dropouts [LH11], have raised interesting control-quality optimization problems for control applications closed over communication networks.

9.2 Nonperiodic Control and Computing

Nonperiodic control approaches like event-based and self-triggered control generally reduce the number of controller executions compared to periodic execution. Such control paradigms are important for systems with tight constraints on the amount of sensor measurements and actua-

tions, as well as on the computation and communication bandwidth that is occupied by the control application. Event-based and self-triggered control can thus be important in the context of energy-constrained and battery-powered computing systems. Other examples are when the use of sensing and actuation devices incurs high cost—for example, energy consumption, wear out, or the lifetime of the devices. In the automotive systems domain, we see a huge increase in software on platforms with limited computation and communication resources. In such contexts, it is important to find solutions that not only provide high quality of service but also are efficient in terms of their resource utilization. Event-based and self-triggered control are promising solutions in highly resource-constrained environments.

To implement nonperiodic control applications on embedded computing platforms, we need methods for integration of multiple event-based or self-triggered control applications on uniprocessor and multiprocessor systems. Several problems related to task mapping, scheduling, and communication synthesis are interesting to study in more detail. Further, the effect of delay and jitter may potentially be more dramatic on control performance as compared to delay and jitter in systems with periodic execution. The relation between delay sensitivity and the event-triggering condition is an interesting research question; for periodic control, the important relation is the one between the delay characteristics and the chosen control period for sampling and actuation.

The nature of event-based and self-triggered control is that the execution is triggered based on the runtime state of the controlled processes. The execution of multiple such control applications on a shared computation and communication infrastructure needs to be supported by adaptive resource management policies. In addition to finding high-quality solutions, these policies have to be executed on the platform with low time overhead and without excessive resource usage. These are interesting and challenging research questions that may lead to efficient utilization of computation and communication resources, as well as better control performance than periodic control. Research in this direction can thus lead to important results of both theoretical and practical relevance. Steps towards this direction have been made for event-based controllers on a shared communication link [CH08, HC10] and for self-triggered control applications communicating on CAN [AT09] or wireless networks [TFJD10, AAM+11]. However, there is much research that

remains to be done in order to develop a complete design and runtime framework for nonperiodic control applications on modern computing platforms.

Bibliography

[AAM+11] J. Araújo, A. Anta, M. Mazo Jr., J. Faria, A. Hernandez, P. Tabuada, and K. H. Johansson. Self-triggered control over wireless sensor and actuator networks. In *Proceedings of the 7th IEEE International Conference on Distributed Computing in Sensor Systems*, 2011.

[ABR+93] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[ACSF10] P. Albertos, A. Crespo, J. E. Simó, and A. Fernández. Control co-design: Algorithms and their implementation. In *Proceedings of the 15th Ada-Europe International Conference on Reliable Software Technologies*, volume 6106 of *Lecture Notes in Computer Science*, pages 19–40. Springer, 2010.

[AHSG07] M. Amirijoo, J. Hansson, S. H. Son, and S. Gunnarsson. Experimental evaluation of linear time-invariant models for feedback performance control in real-time systems. *Real-Time Systems*, 35(3):209–238, 2007.

[ASEP11] A. Aminifar, S. Samii, P. Eles, and Z. Peng. Control-quality driven task mapping for distributed embedded control systems. In *Proceedings of the 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2011.

[ASL+03] T. F. Abdelzaher, J. A. Stankovic, C. Lu, R. Zhang, and Y. Lu. Feedback performance control in software services. *IEEE Control Systems Magazine*, 23(3):74–90, 2003.

[ASP10] J. Almeida, C. Silvestre, and A. M. Pascoal. Self-triggered state feedback control of linear plants under bounded disturbances. In *Proceedings of the* 49^{th} *IEEE Conference on Decision and Control*, pages 7588–7593, 2010.

[AT08a] A. Anta and P. Tabuada. Self-triggered stabilization of homogeneous control systems. In *Proceedings of the American Control Conference*, pages 4129–4134, 2008.

[AT08b] A. Anta and P. Tabuada. Space-time scaling laws for self-triggered control. In *Proceedings of the* 47^{th} *IEEE Conference on Decision and Control*, pages 4420–4425, 2008.

[AT09] A. Anta and P. Tabuada. On the benefits of relaxing the periodicity assumption for networked control systems over CAN. In *Proceedings of the* 30^{th} *IEEE Real-Time Systems Symposium*, pages 3–12, 2009.

[AT10] A. Anta and P. Tabuada. To sample or not to sample: Self-triggered control for nonlinear systems. *IEEE Transactions on Automatic Control*, 55(9):2030–2042, 2010.

[AW07] K. R. Apt and M. G. Wallace. *Constraint Logic Programming using ECL^{i}PS^e*. Cambridge University Press, 2007.

[BBE+11] E. Bini, G. Buttazzo, J. Eker, S. Schorr, R. Guerra, G. Fohler, K.-E. Årzén, V. R. Segovia, and C. Scordino. Resource management on multicore systems: The ACTORS approach. *IEEE Micro*, 31(3):72–81, 2011.

[BC07] G. Buttazzo and A. Cervin. Comparative assessment and evaluation of jitter control methods. In *Proceedings of the* 15^{th} *International Conference on Real-Time and Network Systems*, pages 137–144, 2007.

[BC08] E. Bini and A. Cervin. Delay-aware period assignment in control systems. In *Proceedings of the* 29^{th} *IEEE Real-Time Systems Symposium*, pages 291–300, 2008.

[BC11] S. Borkar and A. A. Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.

[BCH06] M. E.-M. Ben Gaid, A. S. Cela, and Y. Hamam. Optimal integrated control and scheduling of networked control systems with communication constraints: Application to a car suspension system. *IEEE Transactions on Control Systems Technology*, 14(4):776–787, 2006.

[BCH09] M. E.-M. Ben Gaid, A. S. Cela, and Y. Hamam. Optimal real-time scheduling of control tasks with state feedback resource allocation. *IEEE Transactions on Control Systems Technology*, 2(17):309–326, 2009.

[BD05] E. Bini and M. Di Natale. Optimal task rate selection in fixed priority systems. In *Proceedings of the* 26^{th} *IEEE Real-Time Systems Symposium*, pages 399–409, 2005.

[Bor05] S. Y. Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, 2005.

[Bos91] R. Bosch GmbH. *CAN Specification Version 2.0*. 1991.

[BRVC04] P. Balbastre, I. Ripoll, J. Vidal, and A. Crespo. A task model to reduce control delays. *Real-Time Systems*, 27(3):215–236, 2004.

[But97] G. C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer Academic, 1997.

[BVM07] G. Buttazzo, M. Velasco, and P. Martí. Quality-of-control management in overloaded real-time systems. *IEEE Transactions on Computers*, 56(2):253–266, 2007.

[CBS00] M. Caccamo, G. Buttazzo, and L. Sha. Elastic feedback control. In *Proceedings of the* 12^{th} *Euromicro Conference on Real-Time Systems*, pages 121–128, 2000.

[CBS02] M. Caccamo, G. Buttazzo, and L. Sha. Handling execution overruns in hard real-time control systems. *IEEE Transactions on Computers*, 51(7):835–849, 2002.

[CE05] A. Cervin and J. Eker. Control-scheduling codesign of real-time systems: The control server approach. *Journal of Embedded Computing*, 1(2):209–224, 2005.

[CEBÅ02] A. Cervin, J. Eker, B. Bernhardsson, and K.-E. Årzén. Feedback–feedforward scheduling of control tasks. *Real-Time Systems*, 23(1–2):25–53, 2002.

[Cer99] A. Cervin. Improved scheduling of control tasks. In *Proceedings of the* 11^{th} *Euromicro Conference on Real-Time Systems*, pages 4–10, 1999.

[CH08] A. Cervin and T. Henningsson. Scheduling of event-triggered controllers on a shared network. In *Proceedings of the* 47^{th} *IEEE Conference on Decision and Control*, pages 3601–3606, 2008.

[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, 2003.

[CJ08] A. Cervin and E. Johannesson. Sporadic control of scalar systems with delay, jitter and measurement noise. In *Proceedings of the* 17^{th} *IFAC World Congress*, volume 17, 2008.

[CKT03a] S. Chakraborty, S. Künzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *Proceedings of the Design, Automation and Test in Europe Conference*, pages 190–195, 2003.

[CKT+03b] S. Chakraborty, S. Künzli, L. Thiele, A. Herkersdorf, and P. Sagmeister. Performance evaluation of network processor architectures: Combining simulation with analytical estimation. *Computer Networks*, 41(5):641–665, 2003.

[CL10] A. Cervin and B. Lincoln. Jitterbug 1.23 reference manual, July 2010.

[CLE+04] A. Cervin, B. Lincoln, J. Eker, K.-E. Årzén, and G. Buttazzo. The jitter margin and its application in the design of real-time control systems. In *Proceedings of the* 10^{th} *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2004.

[CMV+06] R. Castañé, P. Martí, M. Velasco, A. Cervin, and D. Henriksson. Resource management for control tasks based on the transient dynamics of closed-loop systems. In *Proceedings of the* 18^{th} *Euromicro Conference on Real-Time Systems*, pages 171–182, 2006.

[Cog09] R. Cogill. Event-based control using quadratic approximate value functions. In *Proceedings of the* 48^{th} *IEEE Conference on Decision and Control*, pages 5883–5888, 2009.

[CVMC10] A. Cervin, M. Velasco, P. Martí, and A. Camacho. Optimal online sampling period assignment: Theory and experiments. *IEEE Transactions on Control Systems Technology*, 2010.

[DBBL07] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien. Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, 2007.

[DS00] M. Di Natale and J. A. Stankovic. Scheduling distributed real-time tasks with minimum jitter. *IEEE Transactions on Computers*, 49(4):303–316, 2000.

[DZD+07] A. Davare, Q. Zhu, M. Di Natale, C. Pinello, S. Kanajan, and A. Sangiovanni-Vincentelli. Period optimization for hard real-time distributed automotive systems. In *Proceedings of the* 44^{th} *Design Automation Conference*, pages 278–283, 2007.

[EDPP00] P. Eles, A. Doboli, P. Pop, and Z. Peng. Scheduling with bus access optimization for distributed embedded systems. *IEEE Transactions on Very Large Scale Integrated Systems*, 8(5):472–491, 2000.

[EHÅ00] J. Eker, P. Hagander, and K.-E. Årzén. A feedback scheduler for real-time controller tasks. *Control Engineering Practice*, 8(12):1369–1378, 2000.

[Fid98] C. J. Fidge. Real-time schedulability tests for preemptive multitasking. *Real-Time Systems*, 14(1):61–93, 1998.

[Fle05] FlexRay Consortium. *FlexRay Communications System. Protocol Specification Version 2.1*. 2005.

[GG95] J. J. Gutierrez Garcia and M. Gonzalez Harbour. Optimized priority assignment for tasks and messages in distributed real-time systems. In *Proceedings of the* 3^{rd} *Workshop on Parallel and Distributed Real-Time Systems*, pages 124–132, 1995.

[Gri04] R. P. Grimaldi. *Discrete and Combinatorial Mathematics: An Applied Introduction*. Pearson, fifth edition, 2004.

[GSC11a] D. Goswami, R. Schneider, and S. Chakraborty. Co-design of cyber-physical systems via controllers with flexible delay constraints. In *Proceedings of the* 16^{th} *Asia and South Pacific Design Automation Conference*, pages 225–230, 2011.

[GSC11b] D. Goswami, R. Schneider, and S. Chakraborty. Re-engineering cyber-physical control applications for hybrid communication protocols. In *Proceedings of the Design, Automation and Test in Europe Conference*, pages 1–6, 2011.

[GW04] C. F. Gerald and P. O. Wheatley. *Applied Numerical Analysis*. Addison-Wesley, seventh edition, 2004.

[HBC+07] A. Hagiescu, U. D. Bordoloi, S. Chakraborty, P. Sampath, P. V. V. Ganesan, and S. Ramesh. Performance analysis of FlexRay-based ECU networks. In *Proceedings of the* 44^{th} *Design Automation Conference*, pages 284–289, 2007.

[HC05] D. Henriksson and A. Cervin. Optimal on-line sampling period assignment for real-time control tasks based on plant state information. In *Proceedings of the* 44^{th} *IEEE Conference on Decision and Control*, pages 4469–4474, 2005.

[HC10] T. Henningsson and A. Cervin. A simple model for the interference between event-based control loops using a shared medium. In *Proceedings of the* 49^{th} *IEEE Conference on Decision and Control*, pages 3240–3245, 2010.

[HCÅÅ02a] D. Henriksson, A. Cervin, J. Åkesson, and K.-E. Årzén. Feedback scheduling of model predictive controllers. In *Proceedings of the* 8^{th} *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 207–216, 2002.

[HCÅÅ02b] D. Henriksson, A. Cervin, J. Åkesson, and K.-E. Årzén. On dynamic real-time scheduling of model predictive controllers. In *Proceedings of the* 41^{st} *IEEE Conference on Decision and Control*, pages 1325–1330, 2002.

[HDPT04] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. Wiley, 2004.

[HHJ+05] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis – the SymTA/S approach. *IEE Proceedings of Computers and Digital Techniques*, 152(2):148–166, 2005.

[HJC08] T. Henningsson, E. Johannesson, and A. Cervin. Sporadic event-based control of first-order linear stochastic systems. *Automatica*, 44(11):2890–2895, 2008.

[HNX07] J. P. Hespanha, P. Naghshtabrizi, and Y. Xu. A survey of recent results in networked control systems. *Proceedings of the IEEE*, 95(1):138–162, 2007.

[Hol75] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[HSv08] W. P. M. H. Heemels, J. H. Sandee, and P. P. J. van den Bosch. Analysis of event-driven controllers for linear systems. *International Journal of Control*, 81(4):571–590, 2008.

[JP86] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[KK07] I. Koren and C. M. Krishna. *Fault-Tolerant Systems*. Morgan Kaufmann, 2007.

[KL04] C.-Y. Kao and B. Lincoln. Simple stability criteria for systems with time-varying delays. *Automatica*, 40(8):1429–1434, 2004.

[KMN+00] K. Keutzer, S. Malik, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: Orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1523–1543, 2000.

[KN05] U. Kiencke and L. Nielsen. *Automotive Control Systems for Engine, Driveline, and Vehicle*. Springer, second edition, 2005.

[Kop97] H. Kopetz. *Real-Time Systems—Design Principles for Distributed Embedded Applications*. Kluwer Academic, 1997.

[KS97] C. M. Krishna and K. G. Shin. *Real-Time Systems*. McGraw-Hill, 1997.

[Kuc03] K. Kuchcinski. Constraints-driven scheduling and resource assignment. *ACM Transactions on Design Automation of Electronic Systems*, 8(3):355–383, 2003.

[LC02] B. Lincoln and A. Cervin. Jitterbug: A tool for analysis of real-time control performance. In *Proceedings of the* 41^{st} *IEEE Conference on Decision and Control*, pages 1319–1324, 2002.

[LD91] H. R. Lewis and L. Denenberg. *Data Structures and Their Algorithms*. Addison Wesley, 1991.

[Lee09] E. A. Lee. Computing needs time. *Communications of the ACM*, 52(5):70–79, 2009.

[LH02] G. Leen and D. Heffernan. TTCAN: A new time-triggered controller area network. *Microprocessors and Microsystems*, 26(2):77–94, 2002.

[LH11] M. Lemmon and X. S. Hu. Almost sure stability of networked control systems under exponentially bounded bursts of dropouts. In *Proceedings of the* 14^{th} *International Conference on Hybrid Systems: Computation and Control*, pages 301–310, 2011.

[Lin02] B. Lincoln. Jitter compensation in digital control systems. In *Proceedings of the American Control Conference*, pages 2985–2990, 2002.

[LKP+10] C. Lee, H. Kim, H. Park, S. Kim, H. Oh, and S. Ha. A task remapping technique for reliable multi-core embedded systems. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, pages 307–316, 2010.

[LL73] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):47–61, 1973.

[LL10] J. Lunze and D. Lehmann. A state-feedback approach to event-based control. *Automatica*, 46(1):211–215, 2010.

[LSST02] C. Lu, J. A. Stankovic, S. H. Son, and G. Tao. Feedback-control real-time scheduling: Framework, modeling, and algorithms. *Real-Time Systems*, 23(1–2):85–126, 2002.

[MAT10] M. Mazo Jr., A. Anta, and P. Tabuada. An ISS self-triggered implementation of linear controllers. *Automatica*, 46(8):1310–1314, 2010.

[MFFR01] P. Martí, J. M. Fuertes, G. Fohler, and K. Ramamritham. Jitter compensation for real-time control systems. In *Proceedings of the* 22^{nd} *IEEE Real-Time Systems Symposium*, pages 39–48, 2001.

[Mic96] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer, third edition, 1996.

[MLB+04] P. Martí, C. Lin, S. A. Brandt, M. Velasco, and J. M. Fuertes. Optimal state feedback based resource allocation

for resource-constrained control tasks. In *Proceedings of the* 25^{th} *IEEE Real-Time Systems Symposium*, pages 161–172, 2004.

[MLB+09] P. Martí, C. Lin, S. A. Brandt, M. Velasco, and J. M. Fuertes. Draco: Efficient resource management for resource-constrained control tasks. *IEEE Transactions on Computers*, 58(1):90–105, 2009.

[MRT11] R. Majumdar, E. Render, and P. Tabuada. Robust discrete synthesis against unspecified disturbances. In *Proceedings of the* 14^{th} *International Conference on Hybrid Systems: Computation and Control*, pages 211–220, 2011.

[MT09] M. Mazo Jr. and P. Tabuada. Input-to-state stability of self-triggered control systems. In *Proceedings of the* 48^{th} *IEEE Conference on Decision and Control*, pages 928–933, 2009.

[MYV+04] P. Martí, J. Yépez, M. Velasco, R. Villà, and J. M. Fuertes. Managing quality-of-control in network-based control systems by controller and message scheduling co-design. *IEEE Transactions on Industrial Electronics*, 51(6):1159–1167, 2004.

[NBW98] J. Nilsson, B. Bernhardsson, and B. Wittenmark. Stochastic analysis and control of real-time systems with random time delays. *Automatica*, 34(1):57–64, 1998.

[NSSW05] N. Navet, Y. Song, F. Simonot-Lion, and C. Wilwert. Trends in automotive communication systems. *Proceedings of the IEEE*, 93(6):1204–1223, 2005.

[PB98] P. Pedro and A. Burns. Schedulability analysis for mode changes in flexible real-time systems. In *Proceedings of the* 10^{th} *Euromicro Conference on Real-Time Systems*, pages 172–179, 1998.

[PCS08] C. Pinello, L. P. Carloni, and A. L. Sangiovanni-Vincentelli. Fault-tolerant distributed deployment of embedded control software. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(5):906–919, 2008.

[PEP03] T. Pop, P. Eles, and Z. Peng. Schedulability analysis for distributed heterogeneous time/event-triggered real-time systems. In *Proceedings of the* 15^{th} *Euromicro Conference on Real-Time Systems*, pages 257–266, 2003.

[PEP04] P. Pop, P. Eles, and Z. Peng. *Analysis and Synthesis of Distributed Real-Time Embedded Systems*. Kluwer Academic, 2004.

[PEPP06] P. Pop, P. Eles, Z. Peng, and T. Pop. Analysis and optimization of distributed real-time embedded systems. *ACM Transactions on Design Automation of Electronic Systems*, 11(3):593–625, 2006.

[PG98] J. C. Palencia Gutiérrez and M. González Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proceedings of the* 19^{th} *IEEE Real-Time Systems Symposium*, pages 26–37, 1998.

[PLS11] L. T. X. Phan, I. Lee, and O. Sokolsky. A semantic framework for mode change protocols. In *Proceedings of the* 17^{th} *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 91–100, 2011.

[PPE+08] T. Pop, P. Pop, P. Eles, Z. Peng, and A. Andrei. Timing analysis of the FlexRay communication protocol. *Real-Time Systems*, 39(1–3):205–235, 2008.

[PPEP07] T. Pop, P. Pop, P. Eles, and Z. Peng. Bus access optimisation for FlexRay-based distributed embedded systems. In *Proceedings of the Design, Automation and Test in Europe Conference*, pages 51–62, 2007.

[PPEP08] T. Pop, P. Pop, P. Eles, and Z. Peng. Analysis and optimisation of hierarchically scheduled multiprocessor embedded systems. *International Journal of Parallel Programming*, 36(1):37–67, 2008.

[PPS+02] L. Palopoli, C. Pinello, A. Sangiovanni-Vincentelli, L. Elghaoui, and A. Bicchi. Synthesis of robust control systems

under resource constraints. In *Proceedings of the* 5^{th} *International Workshop on Hybrid Systems: Computation and Control*, pages 337–350, 2002.

[PSA97] D. T. Peng, K. G. Shin, and T. F. Abdelzaher. Assignment and scheduling communicating tasks in distributed real-time systems. *IEEE Transactions on Software Engineering*, 23(12):745–759, 1997.

[PTVF07] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, third edition, 2007.

[Ram95] K. Ramamritham. Allocation and scheduling of precedence-related periodic tasks. *IEEE Transactions on Parallel and Distributed Systems*, 6(4):412–420, 1995.

[RC04] J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Systems*, 26(2):161–197, 2004.

[Ree93] C. R. Reeves, editor. *Modern Heuristic Techniques for Combinatorial Problems*. Blackwell Scientific Publications, 1993.

[RJE03] K. Richter, M. Jersak, and R. Ernst. A formal approach to MpSoC performance verification. *IEEE Computer*, 36(4):60–67, 2003.

[RS00] H. Rehbinder and M. Sanfridson. Integration of off-line scheduling and optimal control. In *Proceedings of the* 12^{th} *Euromicro Conference on Real-Time Systems*, pages 137–143, 2000.

[SAÅ+04] L. Sha, T. Abdelzaher, K.-E. Årzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. Real-time scheduling theory: A historical perspective. *Real-Time Systems*, 28(2–3):101–155, 2004.

[SBB11] L. Santinelli, G. Buttazzo, and E. Bini. Multi-moded resource reservations. In *Proceedings of the* 17^{th} *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 37–46, 2011.

[SBEP11] S. Samii, U. D. Bordoloi, P. Eles, and Z. Peng. Control-quality optimization of distributed embedded control systems with adaptive fault tolerance. In *Proceedings of the Workshop on Adaptive and Reconfigurable Embedded Systems*, 2011.

[SBK10a] D. Skarin, R. Barbosa, and J. Karlsson. Comparing and validating measurements of dependability attributes. In *Proceedings of the* 8^{th} *European Dependable Computing Conference*, pages 3–12, 2010.

[SBK10b] D. Skarin, R. Barbosa, and J. Karlsson. GOOFI-2: A tool for experimental dependability assessment. In *Proceedings of the* 40^{th} *International Conference on Dependable Systems and Networks*, pages 557–562, 2010.

[SCEP09] S. Samii, A. Cervin, P. Eles, and Z. Peng. Integrated scheduling and synthesis of control applications on distributed embedded systems. In *Proceedings of the Design, Automation and Test in Europe Conference*, pages 57–62, 2009.

[SCV+11] S. Sundaram, J. Chang, K. K. Venkatasubramanian, C. Enyioha, I. Lee, and G. J. Pappas. Reputation-based networked control with data-corrupting channels. In *Proceedings of the* 14^{th} *International Conference on Hybrid Systems: Computation and Control*, pages 291–300, 2011.

[SEP+10] S. Samii, P. Eles, Z. Peng, P. Tabuada, and A. Cervin. Dynamic scheduling and control-quality optimization of self-triggered control applications. In *Proceedings of the* 31^{st} *IEEE Real-Time Systems Symposium*, pages 95–104, 2010.

[SEPC09] S. Samii, P. Eles, Z. Peng, and A. Cervin. Quality-driven synthesis of embedded multi-mode control systems. In *Pro-*

ceedings of the 46^{th} *Design Automation Conference*, pages 864–869, 2009.

[SEPC10] S. Samii, P. Eles, Z. Peng, and A. Cervin. Runtime trade-offs between control performance and resource usage in embedded self-triggered control systems. In *Proceedings of the Workshop on Adaptive Resource Management*, 2010.

[SEPC11] S. Samii, P. Eles, Z. Peng, and A. Cervin. Design optimization and synthesis of FlexRay parameters for embedded control applications. In *Proceedings of the* 6^{th} *IEEE International Symposium on Electronic Design, Test and Applications*, pages 66–71, 2011.

[SLCB00] L. Sha, X. Liu, M. Caccamo, and G. Buttazzo. Online control optimization using load driven scheduling. In *Proceedings of the* 39^{th} *IEEE Conference on Decision and Control*, pages 4877–4882, 2000.

[SLSS96] D. Seto, J. P. Lehoczky, L. Sha, and K. G. Shin. On task schedulability in real-time control systems. In *Proceedings of the* 17^{th} *IEEE Real-Time Systems Symposium*, pages 13–21, 1996.

[SLSS01] D. Seto, J. P. Lehoczky, L. Sha, and K. G. Shin. Trade-off analysis of real-time control performance and schedulability. *Real-Time Systems*, 21(3):199–217, 2001.

[SM99] K. G. Shin and C. L. Meissner. Adaptation and graceful degradation of control system performance by task reallocation and period adjustment. In *Proceedings of the* 11^{th} *Euromicro Conference on Real-Time Systems*, pages 29–36, 1999.

[SN05] D. Srivastava and P. Narasimhan. Architectural support for mode-driven fault tolerance in distributed applications. In *Proceedings of the Workshop on Architecting Dependable Systems*, 2005.

[SREP08] S. Samii, S. Rafiliu, P. Eles, and Z. Peng. A simulation methodology for worst-case response time estimation of

distributed real-time systems. In *Proceedings of the Design, Automation and Test in Europe Conference*, pages 556–561, 2008.

[SRLR89] L. Sha, R. Rajkumar, J. P. Lehoczky, and K. Ramamritham. Mode change protocols for priority-driven preemptive scheduling. *Real-Time Systems*, 1(3):243–264, 1989.

[SSDB95] J. A. Stankovic, M. Spuri, M. Di Natale, and G. C. Buttazzo. Implications of classical scheduling results for real-time systems. *IEEE Computer*, 26(6):16–25, 1995.

[SSF+07] L. Schenato, B. Sinopoli, M. Franceschetti, K. Poolla, and S. S. Sastry. Foundations of control and estimation over lossy networks. *Proceedings of the IEEE*, 95(1):163–187, 2007.

[SYP+09] S. Samii, Y. Yin, Z. Peng, P. Eles, and Y. Zhang. Immune genetic algorithms for optimization of task priorities and FlexRay frame identifiers. In *Proceedings of the* 15^{th} *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 486–493, 2009.

[Tab07] P. Tabuada. Event-triggered real-time scheduling of stabilizing control tasks. *IEEE Transactions on Automatic Control*, 52(9):1680–1685, 2007.

[TC94] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Journal on Microprocessing and Microprogramming*, 40(2–3):117–134, 1994.

[TFJD10] U. Tiberi, C. Fischione, K. H. Johansson, and M. D. Di Benedetto. Adaptive self-triggered control over IEEE 802.15.4 networks. In *Proceedings of the* 49^{th} *IEEE Conference on Decision and Control*, pages 2099–2104, 2010.

[Tör98] M. Törngren. Fundamentals of implementing real-time control applications in distributed computer systems. *Real-Time Systems*, 14(3):219–250, 1998.

[VFM03] M. Velasco, J. M. Fuertes, and P. Martí. The self triggered task model for real-time control systems. In *Proceedings of the 24th IEEE Real-Time Systems Symposium, Work-in-Progress Track*, 2003.

[VMB08] M. Velasco, P. Martí, and E. Bini. Control-driven tasks: Modeling and analysis. In *Proceedings of the 29th IEEE Real-Time Systems Symposium*, pages 280–290, 2008.

[VMB09] M. Velasco, P. Martí, and E. Bini. On Lyapunov sampling for event-driven controllers. In *Proceedings of the 48th IEEE Conference on Decision and Control*, pages 6238–6243, 2009.

[VMC+06] M. Velasco, P. Martí, R. Castañé, J. Guardia, and J. M. Fuertes. A CAN application profile for control optimization in networked embedded systems. In *Proceedings of the 32nd IEEE Annual Conference on Industrial Electronics*, pages 4638–4643, 2006.

[VSG+10] H. Voit, R. Schneider, D. Goswami, A. Annaswamy, and S. Chakraborty. Optimizing hierarchical schedules for improved control performance. In *Proceedings of the International Symposium on Industrial Embedded Systems*, pages 9–16, 2010.

[WBBC10] Y. Wu, G. Buttazzo, E. Bini, and A. Cervin. Parameter selection for real-time controllers in resource-constrained systems. *IEEE Transactions on Industrial Informatics*, 6(4):610–620, 2010.

[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*, 7(3):36:1–36:53, 2008.

[WL09] X. Wang and M. Lemmon. Self-triggered feedback control systems with finite-gain L2 stability. *IEEE Transactions on Automatic Control*, 45(3):452–467, 2009.

[WMF05] W. Wang, A. K. Mok, and G. Fohler. Pre-scheduling. *Real-Time Systems*, 30(1–2):83–103, 2005.

[WNT95] B. Wittenmark, J. Nilsson, and M. Törngren. Timing problems in real-time control systems. In *Proceedings of the American Control Conference*, pages 2000–2004, 1995.

[Wol09] W. Wolf. Cyber-physical systems. *IEEE Computer*, 42(9):88–89, 2009.

[WÅÅ02] B. Wittenmark, K. J. Åström, and K.-E. Årzén. Computer control: An overview, 2002. IFAC Professional Brief.

[XP90] J. Xu and D. Parnas. Scheduling processes with release times, deadlines, precedence and exclusion relations. *IEEE Transactions on Software Engineering*, 16(3):360–369, 1990.

[ZBP01] W. Zhang, M. S. Branicky, and S. M. Phillips. Stability of networked control systems. *IEEE Control Systems Magazine*, 21(1):84–99, 2001.

[ZSWM08] F. Zhang, K. Szwaykowska, W. Wolf, and V. Mooney. Task scheduling for control oriented requirements for cyber-physical systems. In *Proceedings of the* 29^{th} *IEEE Real-Time Systems Symposium*, pages 47–56, 2008.

[ZZDS07] W. Zheng, Q. Zhu, M. Di Natale, and A. Sangiovanni-Vincentelli. Definition of task allocation and priority assignment in hard real-time distributed systems. In *Proceedings of the* 28^{th} *IEEE Real-Time Systems Symposium*, pages 161–170, 2007.

[ÅB99] K. J. Åström and B. Bernhardsson. Comparison of periodic and event based sampling for first-order stochastic systems. In *Proceedings of the* 14^{th} *IFAC World Congress*, volume J, pages 301–306, 1999.

[ÅC05] K.-E. Årzén and A. Cervin. Control and embedded computing: Survey of research directions. In *Proceedings of the* 16^{th} *IFAC World Congress*, volume 16, 2005.

[ÅCES00] K.-E. Årzén, A. Cervin, J. Eker, and L. Sha. An introduction to control and scheduling co-design. In *Proceedings of the* 39^{th} *IEEE Conference on Decision and Control*, pages 4865–4870, 2000.

[ÅCH05] K.-E. Årzén, A. Cervin, and D. Henriksson. Implementation-aware embedded control systems. In *Handbook of Networked and Embedded Control Systems*. Birkhäuser, 2005.

[Årz99] K.-E. Årzén. A simple event-based PID controller. In *Proceedings of the* 14^{th} *IFAC World Congress*, 1999.

[Åst07] K. J. Åström. Event based control. In *Analysis and Design of Nonlinear Control Systems: In Honor of Alberto Isidori*. Springer, 2007.

[ÅW97] K. J. Åström and B. Wittenmark. *Computer-Controlled Systems*. Prentice Hall, third edition, 1997.

A
Notation

THIS appendix lists symbols that appear often throughout this thesis. The symbols are grouped together according to relevance and topic. For each symbol, we present a short description and a reference to the page where that symbol is introduced and defined for the first time. For the general mathematical notation used throughout this thesis, no page references are given.

Mathematics

$\mathbb{N}$	Set of natural numbers, $0, 1, 2, \ldots$
$\mathbb{R}$	Set of real numbers
$\mathbb{R}^n$	Set of vectors of n real numbers, $n > 0$
$[a, b]$	Closed interval between the real numbers a and b
$[a, b)$	Half-open interval between the real numbers a and b
$\lfloor \cdot \rfloor$	Floor of a real number
$\lceil \cdot \rceil$	Ceiling of a real number
A^T	The transpose of the matrix (or vector) A
$\mathrm{E}\{\cdot\}$	Expected value of a stochastic variable
$\mathrm{D}\{\cdot\}$	Standard deviation of a stochastic variable
$\longleftarrow$	Assignment or initialization

$f : X \longrightarrow Y$ Function from the set X to the set Y
$f^{-1} : Y \longrightarrow X$ Inverse of the function f
$\times$ Cartesian product of sets
$|\cdot|$ Cardinality of a finite set
$\cup$ Union of sets
$\cap$ Intersection of sets
$\setminus$ Set difference
$\subset$ Subset relation
$\subseteq$ Subset-or-equality relation
$\emptyset$ The empty set
2^X Power set of the set X
$\{a_i\}_{i\in\mathcal{I}}$ Set of items indexed by $\mathcal{I}$

Control Applications

$\mathbf{P}$ Set of plants, page 27
$\mathcal{I}_{\mathbf{P}}$ Index set of $\mathbf{P}$, page 27
P_i Plant ($i \in \mathcal{I}_{\mathbf{P}}$), page 27
$\boldsymbol{x}_i$ Plant state, page 27
$\boldsymbol{u}_i$ Plant input (actuators), page 27
$\boldsymbol{y}_i$ Plant output (sensors), page 27
$\boldsymbol{\Lambda}$ Set of control applications, page 32
Λ_i Control application ($i \in \mathcal{I}_{\mathbf{P}}$), page 32
$\mathbf{T}_i$ Set of tasks in control application Λ_i, page 32
$\mathcal{I}_i$ Index set of $\mathbf{T}_i$, page 32
$\boldsymbol{\Gamma}_i$ Set of messages of control application Λ_i, page 32
$\mathbf{T}_{\boldsymbol{\Lambda}}$ Set of all tasks $\bigcup_{i\in\mathcal{I}_{\mathbf{P}}} \mathbf{T}_i$, page 33
τ Task, page 33
τ_{ij} Arbitrary task of application Λ_i ($i \in \mathcal{I}_{\mathbf{P}}$, $j \in \mathcal{I}_i$), page 33
$\tau_{ij}^{(q)}$ Job q of task τ_{ij}, page 34
γ_{ijk} Message between tasks τ_{ij} and τ_{ik} in application Λ_i, page 33
$\gamma_{ijk}^{(q)}$ Instance q of message γ_{ijk}, page 34
J_i Control cost for plant P_i (quality metric), page 28

Platform

$\mathbf{N}$	Set of computation nodes, page 32
$\mathcal{I}_{\mathbf{N}}$	Index set of $\mathbf{N}$, page 32
map	Task mapping to computation nodes, page 34
Π	Specification of allowed nodes for each task, page 34
$\mathsf{mem}_d^{\max}$	Memory on node N_d ($d \in \mathcal{I}_{\mathbf{N}}$) for design solutions, page 74

Control Timing

h_i	Period of control application Λ_i, page 33
$h_{\mathbf{\Lambda}}$	Hyper period of the system, page 34
$\mathbf{H}_i$	Set of allowed periods of application Λ_i, page 43
$\boldsymbol{h}$	Period assignment of the application set $\mathbf{\Lambda}$, page 44
c_{ij}^{bc}	Best-case execution time of τ_{ij}, page 35
c_{ij}^{wc}	Worst-case execution time of τ_{ij}, page 35
$\xi_{c_{ij}}$	Probability function of the execution time of τ_{ij}, page 35
c_{ijk}	Communication time of γ_{ijk} on the bus, page 35
δ_i^{sa}	Sensor–actuator delay, page 30
Δ_i^{sa}	Stochastic variable representing sensor–actuator delay, page 31
$\xi_{\Delta_i}^{\text{sa}}$	Probability function of Δ_i^{sa}, page 31

Multi-Mode Systems

$\mathcal{M}$	Set of modes, page 67
$\mathcal{M}^{\text{func}}$	Set of functional modes, page 68
$\mathcal{M}^{\text{virt}}$	Set of virtual modes, page 68
$\mathbf{M}$	Mode, page 67
$\mathcal{I}_{\mathbf{M}}$	Index set of mode $\mathbf{M}$, page 67
$\overline{\mathcal{M}}(\mathbf{M})$	Supermodes of $\mathbf{M}$, page 67
$\underline{\mathcal{M}}(\mathbf{M})$	Submodes of $\mathbf{M}$, page 67
$\mathcal{M}'_{\uparrow}$	Top modes of the set of modes $\mathcal{M}' \subseteq \mathcal{M}$, page 75
$\mathcal{M}^{\text{impl}}$	Set of implemented modes, page 68
$J^{\mathbf{M}}$	Total control cost in mode $\mathbf{M}$, page 68
$J_i^{\mathbf{M}}$	Control cost of Λ_i in mode $\mathbf{M}$, page 68

$J^{\mathbf{M}}(\mathbf{M}')$ Control cost of $\mathbf{M}$ with the solution of $\mathbf{M}'$, page 74
$\mathsf{mem}_d^{\mathbf{M}}$ Memory space on N_d for the solution of $\mathbf{M}$, page 68

Fault-Tolerant Systems

$\mathcal{X}$ Set of configurations, page 89
$\mathbf{X}$ Configuration, page 89
$\mathcal{X}^{\text{feas}}$ Set of feasible configurations, page 94
$\mathcal{X}^{\text{base}}$ Set of base configurations, page 94
$\mathcal{X}^{\text{min}}$ Set of minimal configurations, page 99
$\mathcal{X}^{\text{oper}}$ Set of configurations with resilience to faults, page 99
$\mathsf{map}_{\mathbf{X}}$ Task mapping to configuration $\mathbf{X}$, page 97
$\mu(\tau)$ Migration time of task τ, page 104
μ^{max} Maximum allowed migration time, page 109
$\mathsf{mem}_d^{\mathbf{X}}$ Memory space on N_d for the solution of $\mathbf{X}$, page 98
$\mathsf{mem}_d(\tau)$ Memory space of τ on N_d, page 104
$\mathcal{X}^{\text{impl}}$ Set of synthesized optional configurations, page 104
$J^{\mathbf{X}}$ Overall control cost in configuration $\mathbf{X}$, page 97
$p(N)$ Failure probability of node $N \in \mathbf{N}$, page 104
$p^{\mathbf{X}}$ Probability of reaching configuration $\mathbf{X}$, page 105
$\mathbf{T}^{(d)}$ Set of tasks that are stored on node N_d ($d \in \mathcal{I}_{\mathbf{N}}$), page 104

Self-Triggered Control

$\mathbf{T}$ Set of self-triggered control tasks, page 116
$\mathcal{I}_{\mathbf{T}}$ Index set of $\mathbf{T}$, page 116
d_i Absolute deadline of the next execution of τ_i, page 122
D_i Relative deadline of the next execution of τ_i, page 122
D_i^{min} Minimum relative deadline of τ_i, page 118
D_i^{max} Maximum relative deadline of τ_i, page 118
ϕ_i Completion time of the latest execution of τ_i, page 122
t_i Start time of the next execution of τ_i, page 122
J_i^{x} State cost of τ_i in the time interval $[\phi_i, d_i]$, page 123
J_i^{c} Control cost of τ_i in the time interval $[\phi_i, d_i]$, page 124
J_i^{r} CPU cost of τ_i in the time interval $[\phi_i, d_i]$, page 124

J_i	Combined control and CPU cost of τ_i, page 124
ρ	Trade-off between control and CPU cost, page 125
$\widehat{J}_i^{\mathrm{x}}$	Approximate state cost of τ_i, page 126
$\widehat{J}_i^{\mathrm{c}}$	Approximate control cost of τ_i, page 131
$\widehat{J}_i$	Approximate combined control and CPU cost of τ_i, page 131
$\boldsymbol{\Xi}_i$	Set of candidate start times of task τ_i, page 128
$\boldsymbol{\Xi}'_i$	Set of schedulable candidate start times of task τ_i, page 133

Department of Computer and Information Science
Linköpings universitet

Dissertations

Linköping Studies in Science and Technology
Linköping Studies in Arts and Science
Linköping Studies in Statistics
Linköpings Studies in Informatics

Linköping Studies in Science and Technology

No 14 **Anders Haraldsson:** A Program Manipulation System Based on Partial Evaluation, 1977, ISBN 91-7372-144-1.

No 17 **Bengt Magnhagen:** Probability Based Verification of Time Margins in Digital Designs, 1977, ISBN 91-7372-157-3.

No 18 **Mats Cedwall**: Semantisk analys av process-beskrivningar i naturligt språk, 1977, ISBN 91- 7372-168-9.

No 22 **Jaak Urmi:** A Machine Independent LISP Compiler and its Implications for Ideal Hardware, 1978, ISBN 91-7372-188-3.

No 33 **Tore Risch:** Compilation of Multiple File Queries in a Meta-Database System 1978, ISBN 91- 7372-232-4.

No 51 **Erland Jungert:** Synthesizing Database Structures from a User Oriented Data Model, 1980, ISBN 91-7372-387-8.

No 54 **Sture Hägglund:** Contributions to the Development of Methods and Tools for Interactive Design of Applications Software, 1980, ISBN 91-7372-404-1.

No 55 **Pär Emanuelson:** Performance Enhancement in a Well-Structured Pattern Matcher through Partial Evaluation, 1980, ISBN 91-7372-403-3.

No 58 **Bengt Johnsson, Bertil Andersson:** The Human-Computer Interface in Commercial Systems, 1981, ISBN 91-7372-414-9.

No 69 **H. Jan Komorowski:** A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation, 1981, ISBN 91-7372-479-3.

No 71 **René Reboh:** Knowledge Engineering Techniques and Tools for Expert Systems, 1981, ISBN 91-7372-489-0.

No 77 **Östen Oskarsson:** Mechanisms of Modifiability in large Software Systems, 1982, ISBN 91- 7372-527-7.

No 94 **Hans Lunell:** Code Generator Writing Systems, 1983, ISBN 91-7372-652-4.

No 97 **Andrzej Lingas:** Advances in Minimum Weight Triangulation, 1983, ISBN 91-7372-660-5.

No 109 **Peter Fritzson:** Towards a Distributed Programming Environment based on Incremental Compilation, 1984, ISBN 91-7372-801-2.

No 111 **Erik Tengvald:** The Design of Expert Planning Systems. An Experimental Operations Planning System for Turning, 1984, ISBN 91-7372- 805-5.

No 155 **Christos Levcopoulos:** Heuristics for Minimum Decompositions of Polygons, 1987, ISBN 91-7870-133-3.

No 165 **James W. Goodwin:** A Theory and System for Non-Monotonic Reasoning, 1987, ISBN 91-7870-183-X.

No 170 **Zebo Peng:** A Formal Methodology for Automated Synthesis of VLSI Systems, 1987, ISBN 91-7870-225-9.

No 174 **Johan Fagerström:** A Paradigm and System for Design of Distributed Systems, 1988, ISBN 91-7870-301-8.

No 192 **Dimiter Driankov:** Towards a Many Valued Logic of Quantified Belief, 1988, ISBN 91-7870-374-3.

No 213 **Lin Padgham:** Non-Monotonic Inheritance for an Object Oriented Knowledge Base, 1989, ISBN 91-7870-485-5.

No 214 **Tony Larsson:** A Formal Hardware Description and Verification Method, 1989, ISBN 91-7870-517-7.

No 221 **Michael Reinfrank:** Fundamentals and Logical Foundations of Truth Maintenance, 1989, ISBN 91-7870-546-0.

No 239 **Jonas Löwgren:** Knowledge-Based Design Support and Discourse Management in User Interface Management Systems, 1991, ISBN 91-7870-720-X.

No 244 **Henrik Eriksson:** Meta-Tool Support for Knowledge Acquisition, 1991, ISBN 91-7870-746-3.

No 252 **Peter Eklund:** An Epistemic Approach to Interactive Design in Multiple Inheritance Hierarchies, 1991, ISBN 91-7870-784-6.

No 258 **Patrick Doherty:** NML3 - A Non-Monotonic Formalism with Explicit Defaults, 1991, ISBN 91-7870-816-8.

No 260 **Nahid Shahmehri:** Generalized Algorithmic Debugging, 1991, ISBN 91-7870-828-1.

No 264 **Nils Dahlbäck:** Representation of Discourse-Cognitive and Computational Aspects, 1992, ISBN 91-7870-850-8.

No 265 **Ulf Nilsson:** Abstract Interpretations and Abstract Machines: Contributions to a Methodology for the Implementation of Logic Programs, 1992, ISBN 91-7870-858-3.

No 270 **Ralph Rönnquist:** Theory and Practice of Tense-bound Object References, 1992, ISBN 91-7870-873-7.

No 273 **Björn Fjellborg:** Pipeline Extraction for VLSI Data Path Synthesis, 1992, ISBN 91-7870-880-X.

No 276 **Staffan Bonnier:** A Formal Basis for Horn Clause Logic with External Polymorphic Functions, 1992, ISBN 91-7870-896-6.

No 277 **Kristian Sandahl:** Developing Knowledge Management Systems with an Active Expert Methodology, 1992, ISBN 91-7870-897-4.

No 281 **Christer Bäckström:** Computational Complexity of Reasoning about Plans, 1992, ISBN 91-7870-979-2.

No 292 **Mats Wirén:** Studies in Incremental Natural Language Analysis, 1992, ISBN 91-7871-027-8.

No 297 **Mariam Kamkar:** Interprocedural Dynamic Slicing with Applications to Debugging and Testing, 1993, ISBN 91-7871-065-0.

No 302 **Tingting Zhang:** A Study in Diagnosis Using Classification and Defaults, 1993, ISBN 91-7871-078-2

No 312 **Arne Jönsson:** Dialogue Management for Natural Language Interfaces - An Empirical Approach, 1993, ISBN 91-7871-110-X.

No 338 **Simin Nadjm-Tehrani:** Reactive Systems in Physical Environments: Compositional Modelling and Framework for Verification, 1994, ISBN 91-7871-237-8.

No 371 **Bengt Savén:** Business Models for Decision Support and Learning. A Study of Discrete-Event Manufacturing Simulation at Asea/ ABB 1968-1993, 1995, ISBN 91-7871-494-X.

No 375 **Ulf Söderman:** Conceptual Modelling of Mode Switching Physical Systems, 1995, ISBN 91-7871-516-4.

No 383 **Andreas Kågedal:** Exploiting Groundness in Logic Programs, 1995, ISBN 91-7871-538-5.

No 396 **George Fodor:** Ontological Control, Description, Identification and Recovery from Problematic Control Situations, 1995, ISBN 91-7871-603-9.

No 413 **Mikael Pettersson:** Compiling Natural Semantics, 1995, ISBN 91-7871-641-1.

No 414 **Xinli Gu:** RT Level Testability Improvement by Testability Analysis and Transformations, 1996, ISBN 91-7871-654-3.

No 416 **Hua Shu:** Distributed Default Reasoning, 1996, ISBN 91-7871-665-9.

No 429 **Jaime Villegas:** Simulation Supported Industrial Training from an Organisational Learning Perspective - Development and Evaluation of the SSIT Method, 1996, ISBN 91-7871-700-0.

No 431 **Peter Jonsson:** Studies in Action Planning: Algorithms and Complexity, 1996, ISBN 91-7871-704-3.

No 437 **Johan Boye:** Directional Types in Logic Programming, 1996, ISBN 91-7871-725-6.

No 439 **Cecilia Sjöberg:** Activities, Voices and Arenas: Participatory Design in Practice, 1996, ISBN 91-7871-728-0.

No 448 **Patrick Lambrix:** Part-Whole Reasoning in Description Logics, 1996, ISBN 91-7871-820-1.

No 452 **Kjell Orsborn:** On Extensible and Object-Relational Database Technology for Finite Element Analysis Applications, 1996, ISBN 91-7871-827-9.

No 459 **Olof Johansson:** Development Environments for Complex Product Models, 1996, ISBN 91-7871-855-4.

No 461 **Lena Strömbäck:** User-Defined Constructions in Unification-Based Formalisms, 1997, ISBN 91-7871-857-0.

No 462 **Lars Degerstedt:** Tabulation-based Logic Programming: A Multi-Level View of Query Answering, 1996, ISBN 91-7871-858-9.

No 475 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av hur ekonomiska styrsystem utformas och används efter företagsförvärv, 1997, ISBN 91-7871-914-3.

No 480 **Mikael Lindvall:** An Empirical Study of Requirements-Driven Impact Analysis in Object-Oriented Software Evolution, 1997, ISBN 91-7871-927-5.

No 485 **Göran Forslund:** Opinion-Based Systems: The Cooperative Perspective on Knowledge-Based Decision Support, 1997, ISBN 91-7871-938-0.

No 494 **Martin Sköld:** Active Database Management Systems for Monitoring and Control, 1997, ISBN 91-7219-002-7.

No 495 **Hans Olsén:** Automatic Verification of Petri Nets in a CLP framework, 1997, ISBN 91-7219-011-6.

No 498 **Thomas Drakengren:** Algorithms and Complexity for Temporal and Spatial Formalisms, 1997, ISBN 91-7219-019-1.

No 502 **Jakob Axelsson:** Analysis and Synthesis of Heterogeneous Real-Time Systems, 1997, ISBN 91-7219-035-3.

No 503 **Johan Ringström:** Compiler Generation for Data-Parallel Programming Languages from Two-Level Semantics Specifications, 1997, ISBN 91-7219-045-0.

No 512 **Anna Moberg:** Närhet och distans - Studier av kommunikationsmönster i satellitkontor och flexibla kontor, 1997, ISBN 91-7219-119-8.

No 520 **Mikael Ronström:** Design and Modelling of a Parallel Data Server for Telecom Applications, 1998, ISBN 91-7219-169-4.

No 522 **Niclas Ohlsson:** Towards Effective Fault Prevention - An Empirical Study in Software Engineering, 1998, ISBN 91-7219-176-7.

No 526 **Joachim Karlsson:** A Systematic Approach for Prioritizing Software Requirements, 1998, ISBN 91-7219-184-8.

No 530 **Henrik Nilsson:** Declarative Debugging for Lazy Functional Languages, 1998, ISBN 91-7219-197-x.

No 555 **Jonas Hallberg:** Timing Issues in High-Level Synthesis, 1998, ISBN 91-7219-369-7.

No 561 **Ling Lin:** Management of 1-D Sequence Data - From Discrete to Continuous, 1999, ISBN 91-7219-402-2.

No 563 **Eva L Ragnemalm:** Student Modelling based on Collaborative Dialogue with a Learning Companion, 1999, ISBN 91-7219-412-X.

No 567 **Jörgen Lindström:** Does Distance matter? On geographical dispersion in organisations, 1999, ISBN 91-7219-439-1.

No 582 **Vanja Josifovski:** Design, Implementation and Evaluation of a Distributed Mediator System for Data Integration, 1999, ISBN 91-7219-482-0.

No 589 **Rita Kovordányi:** Modeling and Simulating Inhibitory Mechanisms in Mental Image Reinterpretation - Towards Cooperative Human-Computer Creativity, 1999, ISBN 91-7219-506-1.

No 592 **Mikael Ericsson:** Supporting the Use of Design Knowledge - An Assessment of Commenting Agents, 1999, ISBN 91-7219-532-0.

No 593 **Lars Karlsson:** Actions, Interactions and Narratives, 1999, ISBN 91-7219-534-7.

No 594 **C. G. Mikael Johansson:** Social and Organizational Aspects of Requirements Engineering Methods - A practice-oriented approach, 1999, ISBN 91-7219-541-X.

No 595 **Jörgen Hansson:** Value-Driven Multi-Class Overload Management in Real-Time Database Systems, 1999, ISBN 91-7219-542-8.

No 596 **Niklas Hallberg:** Incorporating User Values in the Design of Information Systems and Services in the Public Sector: A Methods Approach, 1999, ISBN 91-7219-543-6.

No 597 **Vivian Vimarlund:** An Economic Perspective on the Analysis of Impacts of Information Technology: From Case Studies in Health-Care towards General Models and Theories, 1999, ISBN 91-7219-544-4.

No 598 **Johan Jenvald:** Methods and Tools in Computer-Supported Taskforce Training, 1999, ISBN 91-7219-547-9.

No 607 **Magnus Merkel:** Understanding and enhancing translation by parallel text processing, 1999, ISBN 91-7219-614-9.

No 611 **Silvia Coradeschi:** Anchoring symbols to sensory data, 1999, ISBN 91-7219-623-8.

No 613 **Man Lin:** Analysis and Synthesis of Reactive Systems: A Generic Layered Architecture Perspective, 1999, ISBN 91-7219-630-0.

No 618 **Jimmy Tjäder:** Systemimplementering i praktiken - En studie av logiker i fyra projekt, 1999, ISBN 91-7219-657-2.

No 627 **Vadim Engelson:** Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing, 2000, ISBN 91-7219-709-9.

No 637 **Esa Falkenroth:** Database Technology for Control and Simulation, 2000, ISBN 91-7219-766-8.

No 639 **Per-Arne Persson:** Bringing Power and Knowledge Together: Information Systems Design for Autonomy and Control in Command Work, 2000, ISBN 91-7219-796-X.

No 660 **Erik Larsson:** An Integrated System-Level Design for Testability Methodology, 2000, ISBN 91-7219-890-7.

No 688 **Marcus Bjäreland:** Model-based Execution Monitoring, 2001, ISBN 91-7373-016-5.

No 689 **Joakim Gustafsson:** Extending Temporal Action Logic, 2001, ISBN 91-7373-017-3.

No 720 **Carl-Johan Petri:** Organizational Information Provision - Managing Mandatory and Discretionary Use of Information Technology, 2001, ISBN -91-7373-126-9.

No 724 **Paul Scerri:** Designing Agents for Systems with Adjustable Autonomy, 2001, ISBN 91 7373 207 9.

No 725 **Tim Heyer:** Semantic Inspection of Software Artifacts: From Theory to Practice, 2001, ISBN 91 7373 208 7.

No 726 **Pär Carlshamre:** A Usability Perspective on Requirements Engineering - From Methodology to Product Development, 2001, ISBN 91 7373 212 5.

No 732 **Juha Takkinen:** From Information Management to Task Management in Electronic Mail, 2002, ISBN 91 7373 258 3.

No 745 **Johan Åberg:** Live Help Systems: An Approach to Intelligent Help for Web Information Systems, 2002, ISBN 91-7373-311-3.

No 746 **Rego Granlund:** Monitoring Distributed Teamwork Training, 2002, ISBN 91-7373-312-1.

No 757 **Henrik André-Jönsson:** Indexing Strategies for Time Series Data, 2002, ISBN 917373-346-6.

No 747 **Anneli Hagdahl:** Development of IT-supported Interorganisational Collaboration - A Case Study in the Swedish Public Sector, 2002, ISBN 91-7373-314-8.

No 749 **Sofie Pilemalm:** Information Technology for Non-Profit Organisations - Extended Participatory Design of an Information System for Trade Union Shop Stewards, 2002, ISBN 91-7373-318-0.

No 765 **Stefan Holmlid:** Adapting users: Towards a theory of use quality, 2002, ISBN 91-7373-397-0.

No 771 **Magnus Morin:** Multimedia Representations of Distributed Tactical Operations, 2002, ISBN 91-7373-421-7.

No 772 **Pawel Pietrzak:** A Type-Based Framework for Locating Errors in Constraint Logic Programs, 2002, ISBN 91-7373-422-5.

No 758 **Erik Berglund:** Library Communication Among Programmers Worldwide, 2002, ISBN 91-7373-349-0.

No 774 **Choong-ho Yi:** Modelling Object-Oriented Dynamic Systems Using a Logic-Based Framework, 2002, ISBN 91-7373-424-1.

No 779 **Mathias Broxvall:** A Study in the Computational Complexity of Temporal Reasoning, 2002, ISBN 91-7373-440-3.

No 793 **Asmus Pandikow:** A Generic Principle for Enabling Interoperability of Structured and Object-Oriented Analysis and Design Tools, 2002, ISBN 91-7373-479-9.

No 785 **Lars Hult:** Publika Informationstjänster. En studie av den Internetbaserade encyklopedins bruksegenskaper, 2003, ISBN 91-7373-461-6.

No 800 **Lars Taxén:** A Framework for the Coordination of Complex Systems´ Development, 2003, ISBN 91-7373-604-X

No 808 **Klas Gäre:** Tre perspektiv på förväntningar och förändringar i samband med införande av informationssystem, 2003, ISBN 91-7373-618-X.

No 821 **Mikael Kindborg:** Concurrent Comics - programming of social agents by children, 2003, ISBN 91-7373-651-1.

No 823 **Christina Ölvingson:** On Development of Information Systems with GIS Functionality in Public Health Informatics: A Requirements Engineering Approach, 2003, ISBN 91-7373-656-2.

No 828 **Tobias Ritzau:** Memory Efficient Hard Real-Time Garbage Collection, 2003, ISBN 91-7373-666-X.

No 833 **Paul Pop:** Analysis and Synthesis of Communication-Intensive Heterogeneous Real-Time Systems, 2003, ISBN 91-7373-683-X.

No 852 **Johan Moe:** Observing the Dynamic Behaviour of Large Distributed Systems to Improve Development and Testing – An Empirical Study in Software Engineering, 2003, ISBN 91-7373-779-8.

No 867 **Erik Herzog:** An Approach to Systems Engineering Tool Data Representation and Exchange, 2004, ISBN 91-7373-929-4.

No 872 **Aseel Berglund:** Augmenting the Remote Control: Studies in Complex Information Navigation for Digital TV, 2004, ISBN 91-7373-940-5.

No 869 **Jo Skåmedal:** Telecommuting's Implications on Travel and Travel Patterns, 2004, ISBN 91-7373-935-9.

No 870 **Linda Askenäs:** The Roles of IT - Studies of Organising when Implementing and Using Enterprise Systems, 2004, ISBN 91-7373-936-7.

No 874 **Annika Flycht-Eriksson:** Design and Use of Ontologies in Information-Providing Dialogue Systems, 2004, ISBN 91-7373-947-2.

No 873 **Peter Bunus:** Debugging Techniques for Equation-Based Languages, 2004, ISBN 91-7373-941-3.

No 876 **Jonas Mellin:** Resource-Predictable and Efficient Monitoring of Events, 2004, ISBN 91-7373-956-1.

No 883 **Magnus Bång:** Computing at the Speed of Paper: Ubiquitous Computing Environments for Healthcare Professionals, 2004, ISBN 91-7373-971-5

No 882 **Robert Eklund:** Disfluency in Swedish human-human and human-machine travel booking dialogues, 2004, ISBN 91-7373-966-9.

No 887 **Anders Lindström:** English and other Foreign Linguistic Elements in Spoken Swedish. Studies of Productive Processes and their Modelling using Finite-State Tools, 2004, ISBN 91-7373-981-2.

No 889 **Zhiping Wang:** Capacity-Constrained Production-inventory systems - Modelling and Analysis in both a traditional and an e-business context, 2004, ISBN 91-85295-08-6.

No 893 **Pernilla Qvarfordt:** Eyes on Multimodal Interaction, 2004, ISBN 91-85295-30-2.

No 910 **Magnus Kald:** In the Borderland between Strategy and Management Control - Theoretical Framework and Empirical Evidence, 2004, ISBN 91-85295-82-5.

No 918 **Jonas Lundberg:** Shaping Electronic News: Genre Perspectives on Interaction Design, 2004, ISBN 91-85297-14-3.

No 900 **Mattias Arvola:** Shades of use: The dynamics of interaction design for sociable use, 2004, ISBN 91-85295-42-6.

No 920 **Luis Alejandro Cortés:** Verification and Scheduling Techniques for Real-Time Embedded Systems, 2004, ISBN 91-85297-21-6.

No 929 **Diana Szentivanyi:** Performance Studies of Fault-Tolerant Middleware, 2005, ISBN 91-85297-58-5.

No 933 **Mikael Cäker:** Management Accounting as Constructing and Opposing Customer Focus: Three Case Studies on Management Accounting and Customer Relations, 2005, ISBN 91-85297-64-X.

No 937 **Jonas Kvarnström:** TALplanner and Other Extensions to Temporal Action Logic, 2005, ISBN 91-85297-75-5.

No 938 **Bourhane Kadmiry:** Fuzzy Gain-Scheduled Visual Servoing for Unmanned Helicopter, 2005, ISBN 91-85297-76-3.

No 945 **Gert Jervan:** Hybrid Built-In Self-Test and Test Generation Techniques for Digital Systems, 2005, ISBN: 91-85297-97-6.

No 946 **Anders Arpteg:** Intelligent Semi-Structured Information Extraction, 2005, ISBN 91-85297-98-4.

No 947 **Ola Angelsmark:** Constructing Algorithms for Constraint Satisfaction and Related Problems - Methods and Applications, 2005, ISBN 91-85297-99-2.

No 963 **Calin Curescu:** Utility-based Optimisation of Resource Allocation for Wireless Networks, 2005, ISBN 91-85457-07-8.

No 972 **Björn Johansson:** Joint Control in Dynamic Situations, 2005, ISBN 91-85457-31-0.

No 974 **Dan Lawesson:** An Approach to Diagnosability Analysis for Interacting Finite State Systems, 2005, ISBN 91-85457-39-6.

No 979 **Claudiu Duma:** Security and Trust Mechanisms for Groups in Distributed Services, 2005, ISBN 91-85457-54-X.

No 983 **Sorin Manolache:** Analysis and Optimisation of Real-Time Systems with Stochastic Behaviour, 2005, ISBN 91-85457-60-4.

No 986 **Yuxiao Zhao:** Standards-Based Application Integration for Business-to-Business Communications, 2005, ISBN 91-85457-66-3.

No 1004 **Patrik Haslum:** Admissible Heuristics for Automated Planning, 2006, ISBN 91-85497-28-2.

No 1005 **Aleksandra Tešanovic:** Developing Reusable and Reconfigurable Real-Time Software using Aspects and Components, 2006, ISBN 91-85497-29-0.

No 1008 **David Dinka:** Role, Identity and Work: Extending the design and development agenda, 2006, ISBN 91-85497-42-8.

No 1009 **Iakov Nakhimovski:** Contributions to the Modeling and Simulation of Mechanical Systems with Detailed Contact Analysis, 2006, ISBN 91-85497-43-X.

No 1013 **Wilhelm Dahllöf:** Exact Algorithms for Exact Satisfiability Problems, 2006, ISBN 91-85523-97-6.

No 1016 **Levon Saldamli:** PDEModelica - A High-Level Language for Modeling with Partial Differential Equations, 2006, ISBN 91-85523-84-4.

No 1017 **Daniel Karlsson:** Verification of Component-based Embedded System Designs, 2006, ISBN 91-85523-79-8

No 1018 **Ioan Chisalita:** Communication and Networking Techniques for Traffic Safety Systems, 2006, ISBN 91-85523-77-1.

No 1019 **Tarja Susi:** The Puzzle of Social Activity - The Significance of Tools in Cognition and Cooperation, 2006, ISBN 91-85523-71-2.

No 1021 **Andrzej Bednarski:** Integrated Optimal Code Generation for Digital Signal Processors, 2006, ISBN 91-85523-69-0.

No 1022 **Peter Aronsson:** Automatic Parallelization of Equation-Based Simulation Programs, 2006, ISBN 91-85523-68-2.

No 1030 **Robert Nilsson:** A Mutation-based Framework for Automated Testing of Timeliness, 2006, ISBN 91-85523-35-6.

No 1034 **Jon Edvardsson:** Techniques for Automatic Generation of Tests from Programs and Specifications, 2006, ISBN 91-85523-31-3.

No 1035 **Vaida Jakoniene:** Integration of Biological Data, 2006, ISBN 91-85523-28-3.

No 1045 **Genevieve Gorrell:** Generalized Hebbian Algorithms for Dimensionality Reduction in Natural Language Processing, 2006, ISBN 91-85643-88-2.

No 1051 **Yu-Hsing Huang:** Having a New Pair of Glasses - Applying Systemic Accident Models on Road Safety, 2006, ISBN 91-85643-64-5.

No 1054 **Åsa Hedenskog:** Perceive those things which cannot be seen - A Cognitive Systems Engineering perspective on requirements management, 2006, ISBN 91-85643-57-2.

No 1061 **Cécile Åberg:** An Evaluation Platform for Semantic Web Technology, 2007, ISBN 91-85643-31-9.

No 1073 **Mats Grindal:** Handling Combinatorial Explosion in Software Testing, 2007, ISBN 978-91-85715-74-9.

No 1075 **Almut Herzog:** Usable Security Policies for Runtime Environments, 2007, ISBN 978-91-85715-65-7.

No 1079 **Magnus Wahlström:** Algorithms, measures, and upper bounds for Satisfiability and related problems, 2007, ISBN 978-91-85715-55-8.

No 1083 **Jesper Andersson:** Dynamic Software Architectures, 2007, ISBN 978-91-85715-46-6.

No 1086 **Ulf Johansson:** Obtaining Accurate and Comprehensible Data Mining Models - An Evolutionary Approach, 2007, ISBN 978-91-85715-34-3.

No 1089 **Traian Pop:** Analysis and Optimisation of Distributed Embedded Systems with Heterogeneous Scheduling Policies, 2007, ISBN 978-91-85715-27-5.

No 1091 **Gustav Nordh:** Complexity Dichotomies for CSP-related Problems, 2007, ISBN 978-91-85715-20-6.

No 1106 **Per Ola Kristensson:** Discrete and Continuous Shape Writing for Text Entry and Control, 2007, ISBN 978-91-85831-77-7.

No 1110 **He Tan:** Aligning Biomedical Ontologies, 2007, ISBN 978-91-85831-56-2.

No 1112 **Jessica Lindblom:** Minding the body - Interacting socially through embodied action, 2007, ISBN 978-91-85831-48-7.

No 1113 **Pontus Wärnestål:** Dialogue Behavior Management in Conversational Recommender Systems, 2007, ISBN 978-91-85831-47-0.

No 1120 **Thomas Gustafsson:** Management of Real-Time Data Consistency and Transient Overloads in Embedded Systems, 2007, ISBN 978-91-85831-33-3.

No 1127 **Alexandru Andrei:** Energy Efficient and Predictable Design of Real-time Embedded Systems, 2007, ISBN 978-91-85831-06-7.

No 1139 **Per Wikberg:** Eliciting Knowledge from Experts in Modeling of Complex Systems: Managing Variation and Interactions, 2007, ISBN 978-91-85895-66-3.

No 1143 **Mehdi Amirijoo:** QoS Control of Real-Time Data Services under Uncertain Workload, 2007, ISBN 978-91-85895-49-6.

No 1150 **Sanny Syberfeldt:** Optimistic Replication with Forward Conflict Resolution in Distributed Real-Time Databases, 2007, ISBN 978-91-85895-27-4.

No 1155 **Beatrice Alenljung:** Envisioning a Future Decision Support System for Requirements Engineering - A Holistic and Human-centred Perspective, 2008, ISBN 978-91-85895-11-3.

No 1156 **Artur Wilk:** Types for XML with Application to Xcerpt, 2008, ISBN 978-91-85895-08-3.

No 1183 **Adrian Pop:** Integrated Model-Driven Development Environments for Equation-Based Object-Oriented Languages, 2008, ISBN 978-91-7393-895-2.

No 1185 **Jörgen Skågeby:** Gifting Technologies - Ethnographic Studies of End-users and Social Media Sharing, 2008, ISBN 978-91-7393-892-1.

No 1187 **Imad-Eldin Ali Abugessaisa:** Analytical tools and information-sharing methods supporting road safety organizations, 2008, ISBN 978-91-7393-887-7.

No 1204 **H. Joe Steinhauer:** A Representation Scheme for Description and Reconstruction of Object Configurations Based on Qualitative Relations, 2008, ISBN 978-91-7393-823-5.

No 1222 **Anders Larsson:** Test Optimization for Core-based System-on-Chip, 2008, ISBN 978-91-7393-768-9.

No 1238 **Andreas Borg:** Processes and Models for Capacity Requirements in Telecommunication Systems, 2009, ISBN 978-91-7393-700-9.

No 1240 **Fredrik Heintz:** DyKnow: A Stream-Based Knowledge Processing Middleware Framework, 2009, ISBN 978-91-7393-696-5.

No 1241 **Birgitta Lindström:** Testability of Dynamic Real-Time Systems, 2009, ISBN 978-91-7393-695-8.

No 1244 **Eva Blomqvist:** Semi-automatic Ontology Construction based on Patterns, 2009, ISBN 978-91-7393-683-5.

No 1249 **Rogier Woltjer:** Functional Modeling of Constraint Management in Aviation Safety and Command and Control, 2009, ISBN 978-91-7393-659-0.

No 1260 **Gianpaolo Conte:** Vision-Based Localization and Guidance for Unmanned Aerial Vehicles, 2009, ISBN 978-91-7393-603-3.

No 1262 **AnnMarie Ericsson:** Enabling Tool Support for Formal Analysis of ECA Rules, 2009, ISBN 978-91-7393-598-2.

No 1266 **Jiri Trnka:** Exploring Tactical Command and Control: A Role-Playing Simulation Approach, 2009, ISBN 978-91-7393-571-5.

No 1268 **Bahlol Rahimi:** Supporting Collaborative Work through ICT - How End-users Think of and Adopt Integrated Health Information Systems, 2009, ISBN 978-91-7393-550-0.

No 1274 **Fredrik Kuivinen:** Algorithms and Hardness Results for Some Valued CSPs, 2009, ISBN 978-91-7393-525-8.

No 1281 **Gunnar Mathiason:** Virtual Full Replication for Scalable Distributed Real-Time Databases, 2009, ISBN 978-91-7393-503-6.

No 1290 **Viacheslav Izosimov:** Scheduling and Optimization of Fault-Tolerant Distributed Embedded Systems, 2009, ISBN 978-91-7393-482-4.

No 1294 **Johan Thapper:** Aspects of a Constraint Optimisation Problem, 2010, ISBN 978-91-7393-464-0.

No 1306 **Susanna Nilsson:** Augmentation in the Wild: User Centered Development and Evaluation of Augmented Reality Applications, 2010, ISBN 978-91-7393-416-9.

No 1313 **Christer Thörn:** On the Quality of Feature Models, 2010, ISBN 978-91-7393-394-0.

No 1321 **Zhiyuan He:** Temperature Aware and Defect-Probability Driven Test Scheduling for System-on-Chip, 2010, ISBN 978-91-7393-378-0.

No 1333 **David Broman:** Meta-Languages and Semantics for Equation-Based Modeling and Simulation, 2010, ISBN 978-91-7393-335-3.

No 1337 **Alexander Siemers:** Contributions to Modelling and Visualisation of Multibody Systems Simulations with Detailed Contact Analysis, 2010, ISBN 978-91-7393-317-9.

No 1354 **Mikael Asplund:** Disconnected Discoveries: Availability Studies in Partitioned Networks, 2010, ISBN 978-91-7393-278-3.

No 1359 **Jana Rambusch**: Mind Games Extended: Understanding Gameplay as Situated Activity, 2010, ISBN 978-91-7393-252-3.

No 1374 **Jan-Erik Källhammer**: Using False Alarms when Developing Automotive Active Safety Systems, 2011, ISBN 978-91-7393-153-3.

No 1375 **Mattias Eriksson**: Integrated Code Generation, 2011, ISBN 978-91-7393-147-2.

No 1381 **Ola Leifler**: Affordances and Constraints of Intelligent Decision Support for Military Command and Control – Three Case Studies of Support Systems, 2011, ISBN 978-91-7393-133-5.

No 1386 **Soheil Samii**: Quality-Driven Synthesis and Optimization of Embedded Control Systems, ISBN 978-91-7393-102-1.

Linköping Studies in Arts and Science

No 504 **Ing-Marie Jonsson:** Social and Emotional Characteristics of Speech-based In-Vehicle Information Systems: Impact on Attitude and Driving Behaviour, 2009, ISBN 978-91-7393-478-7.

Linköping Studies in Statistics

No 9 **Davood Shahsavani:** Computer Experiments Designed to Explore and Approximate Complex Deterministic Models, 2008, ISBN 978-91-7393-976-8.

No 10 **Karl Wahlin:** Roadmap for Trend Detection and Assessment of Data Quality, 2008, ISBN 978-91-7393-792-4.

No 11 **Oleg Sysoev:** Monotonic regression for large multivariate datasets, 2010, ISBN 978-91-7393-412-1.

No 12 **Sackmone Sirisack:** Detection of Change-Points in Multivariate Time Series, 2011, ISBN 978-91-7393-109-0.

No 13 **Agné Burauskaite-Harju:** Characterizing Temporal Change and Inter-Site Correlation in Daily and Subdaily Precipitation Extremes, 2011, ISBN 978-91-7393-110-6.

Linköping Studies in Information Science

No 1 **Karin Axelsson:** Metodisk systemstrukturering- att skapa samstämmighet mellan informationssystemarkitektur och verksamhet, 1998. ISBN-9172-19-296-8.

No 2 **Stefan Cronholm:** Metodverktyg och användbarhet - en studie av datorstödd metodbaserad systemutveckling, 1998, ISBN-9172-19-299-2.

No 3 **Anders Avdic:** Användare och utvecklare - om anveckling med kalkylprogram, 1999. ISBN-91-7219-606-8.

No 4 **Owen Eriksson:** Kommunikationskvalitet hos informationssystem och affärsprocesser, 2000, ISBN 91-7219-811-7.

No 5 **Mikael Lind:** Från system till process - kriterier för processbestämning vid verksamhetsanalys, 2001, ISBN 91-7373-067-X.

No 6 **Ulf Melin:** Koordination och informationssystem i företag och nätverk, 2002, ISBN 91-7373-278-8.

No 7 **Pär J. Ågerfalk:** Information Systems Actability - Understanding Information Technology as a Tool for Business Action and Communication, 2003, ISBN 91-7373-628-7.

No 8 **Ulf Seigerroth:** Att förstå och förändra system-utvecklingsverksamheter - en taxonomi för metautveckling, 2003, ISBN91-7373-736-4.

No 9 **Karin Hedström:** Spår av datoriseringens värden – Effekter av IT i äldreomsorg, 2004, ISBN 91-7373-963-4.

No 10 **Ewa Braf:** Knowledge Demanded for Action - Studies on Knowledge Mediation in Organisations, 2004, ISBN 91-85295-47-7.

No 11 **Fredrik Karlsson:** Method Configuration method and computerized tool support, 2005, ISBN 91-85297-48-8.

No 12 **Malin Nordström:** Styrbar systemförvaltning - Att organisera systemförvaltningsverksamhet med hjälp av effektiva förvaltningsobjekt, 2005, ISBN 91-85297-60-7.

No 13 **Stefan Holgersson:** Yrke: POLIS - Yrkeskunskap, motivation, IT-system och andra förutsättningar för polisarbete, 2005, ISBN 91-85299-43-X.

No 14 **Benneth Christiansson, Marie-Therese Christiansson:** Mötet mellan process och komponent - mot ett ramverk för en verksamhetsnära kravspecifikation vid anskaffning av komponent-baserade informationssystem, 2006, ISBN 91-85643-22-X.