UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Towards the Design of Fault-Tolerant Distributed Real-Time Systems

Dissertation

A thesis submitted to the Faculty of Computer Science, Electrical Engineering and Mathematics of the University of Paderborn in partial fulfillment of the requirements for the degree of Dr. rer. nat.

by

Kay Klobedanz

Paderborn, 2014

Supervisors:
Prof. Dr. Franz-Josef Rammig, University of Paderborn
Prof. Dr. Achim Rettberg, CvO University Oldenburg

Date of public examination: March 13., 2014.

Abstract

The number and complexity of embedded computer systems is rapidly increasing. Many of these embedded systems are real-time systems, i.e. the functional correctness depends not only on the provided results but also on their timeliness. Moreover, most embedded real-time systems are distributed systems that are composed of multiple networked processing nodes cooperating on a common function or set of functions. In the design of such systems, the given software functions have to be deployed to the different processing nodes in the network topology. This implies a task mapping and message mapping resulting in corresponding schedules which strongly influence each other. Especially for large systems, the determination of a feasible deployment is a complex and time-consuming task for the designer which cannot be performed without tool support. Therefore, in this thesis we present a design approach for distributed real-time systems that supports the designer to determine an appropriate deployment.

Distributed systems with hard real-time constraints are so-called safety-critical systems. In safety-critical systems, missing a hard deadline may cause catastrophic consequences on the environment or even humans. But beside safety, the system must also reliably provide its intended functionality. Fault tolerance enables a system to continue with safe and reliable operation even in presence of a fault. In this thesis, we focus on the compensation of hardware faults during system runtime resulting in network failures and processor failures. All fault-tolerant techniques require additional redundant elements integrated into the system to detect and compensate faults. Hardware redundancy is perhaps the most common form of redundancy and can be categorized into static and dynamic redundancy. In this thesis, we apply dynamic redundancy by means of reconfiguration to realize fault tolerance. For this purpose, we present concepts for a reconfigurable network topology and for an efficient coordination of the required reconfigurations. Based on these concepts, we extend our approach towards the design of fault-tolerant distributed real-time systems. Moreover, we describe the application of our approach to the de facto standard for automotive system design and approve its feasibility for realistic systems by means of a real-world case study.

Kurzzusammenfassung

Die Anzahl und Komplexität eingebetteter Systeme nimmt stetig zu. Viele dieser Systeme sind Echtzeitsysteme, deren funktionale Korrektheit nicht nur von gelieferten Ergebnissen sondern auch von deren Pünktlichkeit abhängt. Außerdem sind eingebettete Echtzeitsysteme oftmals verteilte Systeme aus mehreren vernetzten Mikroprozessoren, die für eine oder mehrere Funktionalitäten kooperieren. Beim Entwurf solcher Systeme muss die Softwarefunktionalität auf die Prozessoren im Netzwerk verteilt werden. Dies beinhaltet Task- und Nachrichtenzuordnungen sowie die daraus resultierenden Ablaufplanungen, die sich gegenseitig stark beeinflussen. Insbesondere bei großen Systemen ist die Ermittlung einer passenden Verteilung eine komplexe und zeitaufwändige Aufgabe für den Entwickler, die nicht ohne Werkzeugunterstützung durchführbar ist. Daher präsentieren wir in dieser Arbeit einen Ansatz zum Entwurf eingebetteter Echtzeitsysteme, der den Systementwickler bei der Ermittlung einer geeigneten Lösung unterstützt.

Verteilte Systeme mit strikten Echtzeitanforderungen sind sogenannte sicherheitskritische Systeme, bei denen die Verletzung einer harten Zeitschranke der Umgebung oder sogar Menschen Schaden zufügen kann. Neben der nötigen Sicherheit muss ein solches System aber auch zuverlässig die vorgesehene Funktionalität liefern. Fehlertoleranz ermöglicht eine sichere und zuverlässige Fortsetzung des Systembetriebs im Fehlerfall. In dieser Arbeit betrachten wir die Kompensation von Hardwarefehlern zur Systemlaufzeit, die zu einem Netzwerk- oder Prozessorausfall führen können. Alle Ansätze zur Fehlertoleranz benötigen dabei redundante Systemkomponenten zur Erkennung und Kompensation von Fehlern. Hardwareredundanz ist der wohl am weitesten verbreitete Ansatz und kann in statische und dynamische Redundanz unterteilt werden. Zur Realisierung einer dynamischen Fehlerredundanz im Rahmen dieser Arbeit präsentieren wir Konzepte für eine rekonfigurierbare Netzwerkarchitektur und zur effizienten Koordination der notwendigen Rekonfigurationen. Basierend auf diesen Konzepten erweitern wir unseren Ansatz zum Entwurf fehlertoleranter verteilter Echtzeitsysteme. Zusätzlich beschreiben wir die Verwendung des hier vorgestellten Ansatzes im Quasi-Standard zum Entwurf automobiler Systeme und bestätigen dabei die praktische Anwendbarkeit anhand eines realistischen Fallbeispiels.

Acknowledgements

This dissertation is the result of research at C-Lab and the University of Paderborn. Here I would like to express my gratitude to many people, who supported me with their comments, suggestions, criticism, and patience during my work. Without them, this thesis would not have been possible.

First of all, I would like to thank my supervisor Prof. Dr. Franz J. Rammig for his invaluable guidance and constructive feedback on my work. I also thank Prof. Dr. Achim Rettberg for vice-supervising and strongly influencing my dissertation. Furthermore, I would like to thank Prof. Dr. Sybille Hellebrand, Prof. Dr. Marco Platzner, and Prof. Dr. Christian Plessl for taking part in my examination board.

I also would like to thank my C-LAB group leaders Dr. Lisa Kleinjohann, Dr. Bernd Kleinjohann, and especially Dr. Wolfgang Müller for their advice and support on my daily work and research topics.

This work would not have been possible without the fruitful discussions I had with my good colleagues. Therefore, my sincere thanks go to all of them. In this regard, I would like to express my particular gratitude to my colleagues Markus Becker and Jan Jatzkowski who supported me throughout the development of this thesis with their comments, suggestions, and corrections.

Last but not least, I would like to thank my family for their strong support during my education and studies. In particular, my thoughts are with my parents and grandparents who unfortunately could not experience the completion of this dissertation. I am very grateful to my family, my many close friends, and especially to my girlfriend Claudia who guided me through difficult and good times in the last several years.

Paderborn, April 2014

Contents

Chapter 1

Introduction

This thesis presents a novel approach towards the design of fault-tolerant distributed real-time systems. Initially, this chapter introduces into the field of distributed real-time systems and motivates the necessity for a fault-tolerant design. Based on this motivation it presents the objectives of the thesis with the resulting main requirements and contributions. Finally, it provides the structure of the thesis by giving a brief description of each chapter.

1.1 Motivation

Nowadays, the number and complexity of embedded computer systems is rapidly increasing. In contrast to a classical general-purpose PC, an embedded system is a microprocessor-based system that is built into a device or environment and is designed to control a specific function or set of functions [Hea02]. Embedded systems range from small and simple systems like the microprocessor within a digital TV set-top box to very large and complex systems like control systems embedded in process plants. Also the controllers for the ABS system of a car or the operation of its engine as well as the automatic pilot system of an aircraft are embedded systems. In general, by now hardware and software systems are embedded within everyday products and places. Today, already 90% of computing devices are in embedded systems and not in general purpose PCs. The growth rate in embedded systems is more than 10% per year and it is predicted that there will be over 40 billion devices worldwide by 2020. Today, 20% of the value and cost of each car results from embedded electronics and this will increase to an average of 35-50% by 2020 [3TU10].

Many of the embedded systems are real-time systems, i.e. the correctness of the intended functionality depends not only on the provided results but also on the timeliness of the result provisioning by the system. This means the

correct result must be available at a specified deadline. Moreover, nowadays most embedded real-time systems are distributed systems [Kop11]. In contrast to a embedded system with a single processing unit, a distributed system is composed of multiple networked processing nodes cooperating on a common function or set of functions [BW09]. A typical example for a large and complex distributed real-time system is the vehicle network of a modern car as depicted in Figure 1.1.

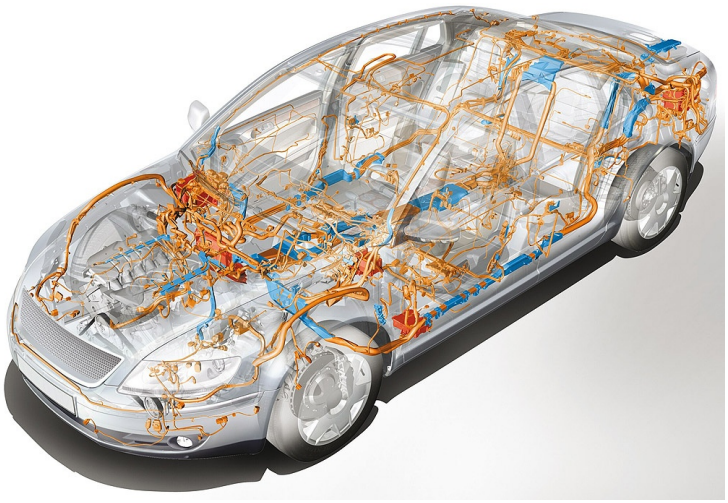


Figure 1.1: Vehicle network of a modern car [Eng12].

Today's modern automotive vehicles contain up to 100 or even more distributed and networked processors which execute thousands of software functions [VBK10]. The functions in a car network are associated to the subsystems powertrain, chassis, body, and multimedia with their corresponding subnetworks [SZ06]. In the design of such a distributed system, the given software functions respectively their corresponding tasks have to be deployed to the different processing nodes in the network topology. Consequently, the deployed tasks on the different distributed processing nodes have to exchange data messages over the network infrastructure, e.g. a communication bus. As described above in a distributed real-time system the timeliness of the task executions and the data transmissions have to be ensured to guarantee a correct system behavior.

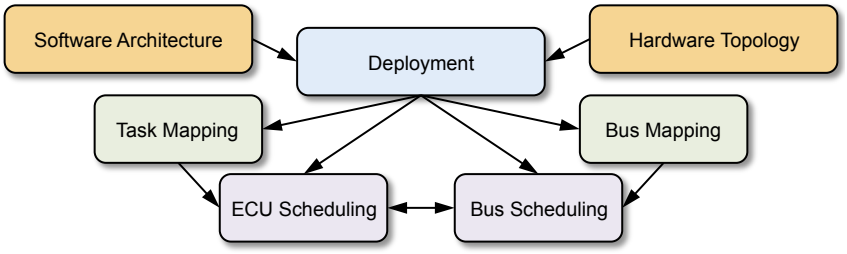


Figure 1.2: Design flow steps for distributed systems [SR08].

Figure 1.2 illustrates the resulting design flow steps for distributed systems. It shows, that the deployment implies a task mapping and message mapping resulting in corresponding schedules which strongly influence each other. This problem of mapping and scheduling tasks and messages in a distributed system is NP-hard [Bur91]. Especially for large systems, the determination of a feasible deployment is a complex and time-consuming task for the designer which cannot be performed without tool support. Therefore, in this thesis we present a design approach for distributed real-time systems that supports the designer to determine an appropriate deployment.

Distributed systems with hard real-time constraints are also called safety-critical systems. In a safety-critical system, missing a specified deadline may cause catastrophic consequences on the environment or even humans. Two of the major domains where safety and reliability are of high importance are the avionic and the automotive domain. In a car typical safety-critical functions are member of the powertrain, chassis, or partly the body subsystem, e.g. engine control, ABS, and airbag control [ZS07]. Obviously, in safety-critical systems the main requirement is to avoid conditions that can cause the catastrophic consequences mentioned before. However, the system must also reliably provide its intended functionality to be useful, as Burns and Wellings illustrate by the following example [BW09]:

> "*In many ways, the only safe airplane is one that never takes off, however, it is not very reliable.*"

Hence, beside hard real-time constraints, a safety-critical system has to consider further requirements and safety concepts to also ensure its reliability. A common concept to improve reliability and safety is fault tolerance [LA90]. Fault tolerance enables a system to continue with a safe and reliable operation even in presence of a fault. This is important since due to the rising size and complexity of embedded systems defects caused by this electronic systems are increasing rapidly [EJ09]. In this thesis we focus on hardware faults during system runtime. In a distributed system these faults may result in two types of component failures: network failures and processor respectively node failures [CJ97]. All fault-tolerant techniques require additional redundant elements integrated into the system to detect and recover from faults. Hardware redundancy is perhaps the most common form of redundancy applied in systems [Pra96]. It can be distinguished between static and dynamic redundancy. Static redundancy typically requires multiple redundant instances of the hardware components, e.g. processors, whereas dynamic redundancy applies reconfiguration to realize fault tolerance.

Figure 1.3 illustrates a CAD model of the Steer-By-Wire system which Nissan installed in their current luxury car model Infiniti Q50. It shows three redundant processing nodes and a mechanical steering column as an additional

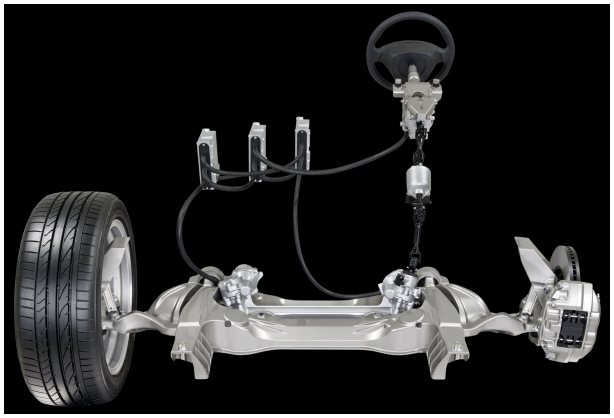


Figure 1.3: CAD model of upcoming Steer-By-Wire system from Nissan (Image: Nissan).

mechanical backup to realize fault tolerance by means of static redundancy [Spe12]. The drawback of this kind of redundancy is that especially, in huge distributed systems with many components the hardware overhead may result in an inappropriate rise of cost and weight. Thus, the aim of fault tolerance is to minimize redundancy while maximizing the reliability [BW09]. Therefore, in this thesis we apply dynamic redundancy by means of reconfiguration to realize fault tolerance. For dynamic redundancy, in case of a detected error the corresponding faulty component must be detected. The faulty component has to be removed and its functionality has to be resumed by another to regain the operational status of the system, i.e. a reconfiguration has to be performed. To offer the required redundancy, we perform task replication, i.e. allocation of redundant task instances on the nodes. Task replication is a fundamental method for improving the reliability of a distributed system [Kim+11]. To enable a flexible task replication and reconfiguration for distributed real-time systems which interact with the environment, this thesis presents concepts for a reconfigurable network topology and for an efficient coordination of the required reconfigurations. Considering these concepts, we extend our approach for the design of fault-tolerant distributed real-time systems. In this way it determines the initial deployment for normal system mode and based on that for all required reconfigurations to compensate possible component faults.

The next section briefly summarizes the objectives and contributions of the thesis to address the challenges described above.

1.2 Objectives of the Thesis

The main objective of this thesis is to develop an approach for the design of fault-tolerant distributed real-time systems that supports the system designer by determining feasible solutions for the required deployments. For

this purpose we extend and refine the design flow steps for distributed systems depicted in Figure 1.2 and present further concepts to enable dynamic redundancy by means of reconfiguration. In the following we shortly present the resulting objectives and requirements for this thesis.

Compensation of arbitrary operational hardware faults During system runtime, different possible operational hardware faults can be distinguished by means of their occurrence and duration and result in different behaviors of the faulty component. In a distributed system these faults may result in either network or node failures. We describe that the compensation of one arbitrary component fault is sufficient in most cases to ensure fault tolerance. Hence, in this thesis we provide means to compensate one arbitrary component fault.

Reconfigurable distributed system topology Current distributed systems exchange data with the environment via hardwired sensors and actuators. This results in placement constraints for tasks which have to exchange data with these sensors or actuators. A failure of such a hardwired processing node cannot be compensated with dynamic redundancy by means of reconfiguration because the required connections get lost. Therefore, we propose a reconfigurable distributed system topology which avoids these placements constraints and offers the necessary flexibility for our fault-tolerant design approach.

Coordination of reconfiguration To realize fault tolerance through dynamic redundancy, a component which coordinates and performs the required reconfiguration steps has to be integrated in the system. Thus, we discuss several methods how to realize this coordination and propose a novel distributed coordinator concept which allows the self-reconfiguration of the system by detecting faulty components and activating redundant task instances on other nodes to resume the functionality and regain the operational status of the system.

Extended schedulability tests Task scheduling in distributed systems has to cover the local level of each processor and the global scheduling, i.e. the mapping of tasks to nodes. For local scheduling we consider RM/DM scheduling which is still most frequently applied in many important real-time domains and EDF scheduling which has some significant advantages over RM/DM scheduling. To determine feasible candidates for the task mapping, the corresponding schedulibality tests for the local scheduling have to be performed. To consider possibly resulting communication delays in a distributed system, we propose extended versions of the existing strategies.

Modeling of system properties To perform an appropriate system design, our approach requires information about the properties of the given software architecture and hardware topology as well as the resulting real-time constraints and communication properties. Thus, we define and present a graph-based modeling of the system properties as input for our approach. These models are annotated with the corresponding timing properties and constraints.

Deployment and design result Fault-tolerant system deployment is an essential part of our approach and shall provide a feasible design result. Hence, we present different algorithms for the determination of feasible task and bus mappings for the initial configuration and based on that for all necessary reconfigurations to design a fault-tolerant system. Finally, our approach returns the resulting design as feedback for the designer. Depending on the given input, our deployment approach is able to return a complete feasible solution for all possible configurations or for a subset. Based on this feedback, the designer can design the system according to the determined solution or perform a further iteration of the design approach.

Application to the design of automotive systems Modern automotive vehicles are a major area of application for large and complex distributed real-time systems. AUTOSAR is the established de facto standard for the development of automotive systems. Hence, we describe the application of our approach to the fault-tolerant design of automotive systems and the deployment of real-time software based on AUTOSAR. Additionally, we propose a further concept to coordinate and realize the required reconfigurations that is based on existing AUTOSAR components.

1.3 Structure of the Thesis

This thesis is structured as follows. Chapter 2 provides the essential fundamentals and basic terms for the work presented in this thesis. This includes general properties of real-time systems, an introduction to task scheduling and communication in distributed real-time systems as well as basic concepts for modeling and specification of timing constraints for such systems. The Chapter is completed by an introduction to fault-tolerant system design.

Chapter 3 gives an overview of related work relevant for the work in this thesis. This includes work and publications dealing with the design and configuration of distributed real-time systems, an overview of publications and techniques regarding the scheduling of real-time tasks and messages in distributed systems, and the presentation of existing concepts and approaches for reconfigurable and fault-tolerant distributed real-time systems.

In Chapter 4 our design approach for fault-tolerant distributed real-time systems is described in detail. Its main objective is the determination of feasible and flexible fault-tolerant system designs with compensation of different usual fault types and a simultaneous reduction of required redundancy based on a reconfigurable distributed system topology with an additional concept for fault tolerance. Applying the presented graph-based input model of system properties, this chapter describes the system deployment and its design result as essential part of the approach.

In Chapter 5 we present how to utilize our approach for the design of fault-tolerant automotive real-time systems. Therefore, this chapter describes the application of our approach to the modeling and deployment of automotive real-time software based on AUTOSAR, a well-defined and standardized methodology for the development of automotive systems.

Chapter 6 concludes the thesis by summarizing the main contributions and providing an outlook for future work. Finally, the publications which have been published in the context of this thesis as basis for the presented work are listed on page 197.

1.4 Summary

This chapter provided an introduction to the work presented in this thesis. It motivated the necessity to develop our approach for the design of fault-tolerant distributed real-time systems. Based on this motivation, the objectives of the thesis were presented. The last section of this chapter described the structure of this thesis and provided a brief description of each following chapter.

Chapter 2

Fundamentals and Basic Terms

This chapter provides a detailed description of fundamental and basic terms essential for the work presented in this thesis. It starts with properties of real-time systems, tasks, and their scheduling. This is proceeded by an introduction to task scheduling and communication in distributed real-time systems. Moreover, basic concepts for modeling and specification of timing constraints for distributed real-time systems are presented. The fundamentals are completed by an introduction to fault-tolerant system design.

2.1 Real-Time Systems

Today's number and complexity of *embedded* computer systems is rapidly increasing. In contrast to a classical general-purpose PC, an embedded system is a microprocessor-based system designed to control a specific function or range of functions [Hea02]. An embedded system is part of a self-contained product, e.g. an aircraft or an automobile, and generally bound to *real-time* constraints. Embedded real-time systems form the most important market segment for real-time technology and the computer industry in general [Kop11]. The term "real-time" means that the computer system is not longer controlling its own time domain. For a real-time system the progress of time of the environment (*external time*) defines how the *internal time* of the system has to progress. This external time may be the real physical time or also artificially generated by a surrounding environment. For the embedded system this makes no difference.

Summarized, Kopetz defines real-time systems as [Kop11]:

> "*A real-time computer system is a computer system where the correctness of the system behavior depends not only on the logical results of the computations, but also on the physical instant (time) when these results are produced. By system behavior we mean the sequence of outputs in time of a system.*"

This definition implies that in strict real-time systems a late result is also wrong. Here the meaning of "lateness" has to be defined dependent on the specific application. A real-time system must respond to stimuli from its environment in a certain way within time intervals dictated by its environment [Tan95]. The instant when a result must be produced is called *deadline*. Any given deadline has to be met under all circumstances – even the worst case. Thus, real-time also involves *predictability* and *determinism*. It is a wide spread myth that real-time systems have to be as fast as possible. In fact they just have to be fast enough to meet all given deadlines, but most of all they have to be predictable. Ensuring this determinism may even slow down a real-time system [Ram+09]. Therefore, real-time systems can be characterized by the strictness of their real-time restrictions [But11]:

- A *hard* real-time system – also called *safety-critical* real-time system – must meet at least one hard deadline. Missing this deadline may cause catastrophic consequences on the environment or people. Typical application areas are automotive systems, like air-bag control or steer-by-wire.
- In a *firm* real-time system the missing of a deadline makes the result useless, but does not cause severe consequences. Typical application areas are forecast systems for weather or stock exchange.
- For a *soft* real-time system the meeting of deadlines is desirable – e.g. for performance or quality reasons – but missing does not cause useless results. Here, typical areas are video and audio streaming or comfort and body electronics in automotive vehicles.

Because of the described strictness of real-time restrictions, the design of a hard real-time system is fundamentally different from the design of a soft real-time system. While a hard real-time system must sustain a guaranteed temporal behavior under all – even worst case – circumstances, it is permissible for a soft real-time computer system to miss a deadline occasionally. The focus of this thesis is on the more complex and demanding design of safety-critical systems with hard real-time constraints.

The process of executing a functionality or an algorithm by a processing unit is called *computation* or *task*. These computations are performed by

the *components* of the real-time system. A component is a self-contained hardware/software unit that interacts with its environment exclusively by the exchange of messages. To be aware of the progression of time, a real-time system contains a real-time clock. After power-up, a component enters a *ready-for-start* state to wait for a *triggering signal* that indicates the start of task execution. When started it reads input messages, updates its internal state and produces output messages, until the computation gets terminated. Afterwards, the component enters the ready-for-start state again to wait for the next triggering signal.

A triggering signal can be associated either with the occurrence of a significant *event* by an *event-triggered* control, or with a specified point in time by a *time-triggered* control [Kop11]. The significant events that form the basis of event-triggered control can be, for example, the arrival of a particular message, the completion of an activity inside a component, or the occurrence of an external interrupt. Time-triggered control signals are derived from the progression of the global time in the real-time system. Time-triggered control signals are typically cyclic. A *cycle* can be characterized by its *period*, i.e. the real-time interval between two successive *cycle starts*, and by its *phase*, which is the interval between the start of the period and the cycle start. We assume that a cycle is associated with each time-triggered activity. To guarantee the required predictability and determinism, safety-critical real-time systems generally utilize time-triggered control, e.g. for sensory data acquisition or control loops. Therefore, in this thesis we focus on cyclic and periodic tasks.

2.1.1 Properties of Real-Time Tasks

In contrast to other computational tasks, real-time tasks additionally consider the notion of *time*. This implies that the correctness of the system depends not only on logical results but also on the time when these results are produced. Concerning the timing restrictions defined above, a real-time task J_i can be characterized by the following properties depicted in Figure 2.1:

- **Release time** r_i: The *release time* r_i is the time at which J_i becomes ready for execution. It is also called *request time* or *arrival time*, denoted by a_i.
- **Computation time** C_i: The *computation time* C_i is the time necessary for the execution of J_i without interruption by other tasks. For this, we consider the *worst case execution time* (WCET), which has to be determined in advance for the task on the executing processor. The WCET is the upper bound on the execution times of a task on a specific hardware platform. Computing such a bound is undecidable in the general case [Mar03]. However, different techniques exist for estimating the

WCET. Those estimates are typically pessimistic, meaning that the estimated WCET is known to be higher than the real one. Consequently, much work on WCET analysis is on reducing the pessimism and computing tight bounds.[1] However, estimation of WCETs is not the focus of our work and we consider them as predetermined values.

- **Deadline d_i (absolute) and D_i (relative)**: The deadline is the time at which a task has to be finished to avoid damage to the system or users. We distinguish between *absolute deadline*, denoted by d_i and *relative deadline*, denoted by D_i. Absolute deadline means related to the global time of the whole system while relative deadline means relative to the release time r_i of J_i.

- **Start time s_i**: The *start time* s_i is the time at which a task starts its execution. A task execution cannot be started before its release time. Furthermore, the execution can be delayed by other tasks. Therefore, $r_i \leq s_i$ always holds.

- **Finishing time f_i**: The *finishing time* f_i is the time at which a task finishes its execution and can be calculated as $f_i = s_i + C_i$. The least possible value results for $s_i = r_i$. But due to delays and interruptions caused by other tasks the finishing time may be later. Nevertheless, $f_i \leq d_i$ must always hold.

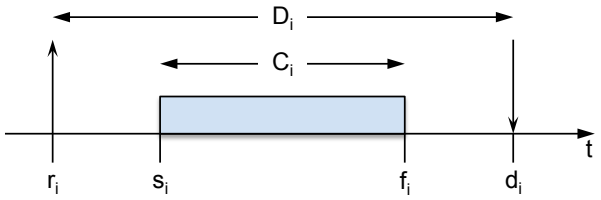


Figure 2.1: Parameters of a real-time task J_i [Ram+09].

Task Dependencies and Precedence Constraints

The tasks of a given task set may be independent or dependent. In many real-time systems the tasks cannot be executed in arbitrary order but have to respect *precedence* relations resulting from task dependencies defined at system design.[2] Obviously task dependencies corresponding to synchronization or communication among tasks introduce additional *precedence constrains*. Therefore, we consider dependent task sets with precedence constraints in this thesis.

[1] For instance, AbsInt offers tools for WCET analysis [Abs12].

[2] "From a practical point of view, results on how to schedule tasks with precedence and mutual exclusion constraints are much more important than the analysis of the independent task model." [Kop11].

A task J_i depends on task J_k, if J_i cannot be started before J_k has been finished. Thus, the notion $J_k \rightarrow J_i$ specifies that J_k is a *direct predecessor* of J_i. Dependencies between tasks can be defined by a *task dependency graph* (TDG), which is a directed acyclic graph (DAG). In a TDG: $G_{\text{TDG}} = (V, E)$ the tasks are represented by a set of vertices V and precedence relations are represented by directed edges E. A TDG induces a partial order on the task set. Figure 2.2 depicts an exemplary TDG with its precedence constraints, e.g. $J_1 \rightarrow J_2$ and $J_3 \rightarrow J_6$.

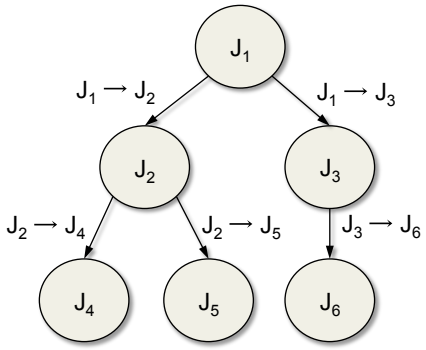


Figure 2.2: Example task dependency graph with precedence constraints.

Periodic Tasks

Two main classes of tasks can be identified: *periodic* and *aperiodic* tasks. Both types are generic, i.e. a sequence of instances is generated over time. Usually such a task-instance is called *job*. Because all jobs share the same code they have the same WCET C_i. In case of a periodic task these instances arrive with a fixed period, denoted by T_i. The first arrival time, i.e. the release time of the first instance is usually called the phase ϕ_i of this generic task. Periodic tasks directly reflect the "sense-execute-act" loop in control applications [Ram+09] and offer the required predictability and determinism for safety-critical real-time systems. These periodic tasks typically arise from sensory data acquisition or control loops which have to be executed time-triggered and cyclically at specific rates derived from the functional requirements of an application. Therefore, a real-time system may contain tasks with different rates resulting in a so-called *multirate system*.

Summarized, a set Γ of periodic real-time tasks can be characterized by the following set of parameters, which are commonly used in literature:

τ_i: A Generic task, where different instances of this task may exist over time.

$\tau_{i,j}$: The jth-instance of τ_i.

T_i: The period of τ_i is the interval between two consecutive arrivals of τ_i.

C_i: The computation time respectively WCET of τ_i is identical for each instance.

$r_{i,j}$: The release time of $\tau_{i,j}$ is an absolute value and specific for each instance j.

ϕ_i: The phase of τ_i is the release time of the first instance ($\phi_i = r_{i,1}$).

D_i: The relative deadline of τ_i. All instances of τ_i have the same relative deadline. For independent tasks this is often equal to the period T_i. For dependent tasks the relative deadline may be less than T_i. Summarized, $D_i \leq T_i$ always holds.

$d_{i,j}$: The absolute deadline is a specific property of each task instance $\tau_{i,j}$. It can be derived from the relative deadline by $d_{i,j} = \phi_i + (j-1) \cdot T_i + D_i$. The time interval $\Psi_{i,j} = [r_{i,j}, d_{i,j}[$ defines the *available execution time interval* in which $\tau_{i,j}$ must be executed.

$s_{i,j}$: The start time is an absolute value, which is specific for each task instance $\tau_{i,j}$. It holds $s_{i,j} \geq r_{i,j}$ for all instances.

$f_{i,j}$: The finishing time is an absolute value, which is specific for each task instance $\tau_{i,j}$. It holds $f_{i,j} \leq d_{i,j}$ for all instances.

Based on these parameters a set of n independent periodic tasks can be defined as [But11]:

$$\Gamma = \{\tau_i = (\phi_i, T_i, C_i) \,|\, i = 1, \ldots, n\}. \tag{2.1}$$

The release time $r_{i,j}$ and absolute deadline $d_{i,j}$ of the j-th instance then can be calculated as:

$$\begin{aligned} r_{i,j} &= \phi_i + (j-1) \cdot T_i, \\ d_{i,j} &= r_{i,j} + T_i = \phi_i + j \cdot T_i. \end{aligned}$$

Due to the resulting precedence constraints for dependent tasks the release times and deadlines of task instances do not just depend on the periods, but also on the minimum finishing times and maximum start times of the direct

predecessors respectively successors. Consequently, these values have to be calculated as described above. Additionally, we can define a *response time* $R_{i,j}$ for each task instance. This is the time – measured from the release time – at which an instance finishes execution, i.e. it responses to the task request:

$$R_{i,j} = f_{i,j} - r_{i,j}.$$

Obviously, the response time increases if the execution of a task instance is delayed or interrupted by other tasks. Therefore, Liu and Layland introduced the *critical instant* of a task. This is the time at which a release of this task will produce the largest response time [LL73]:

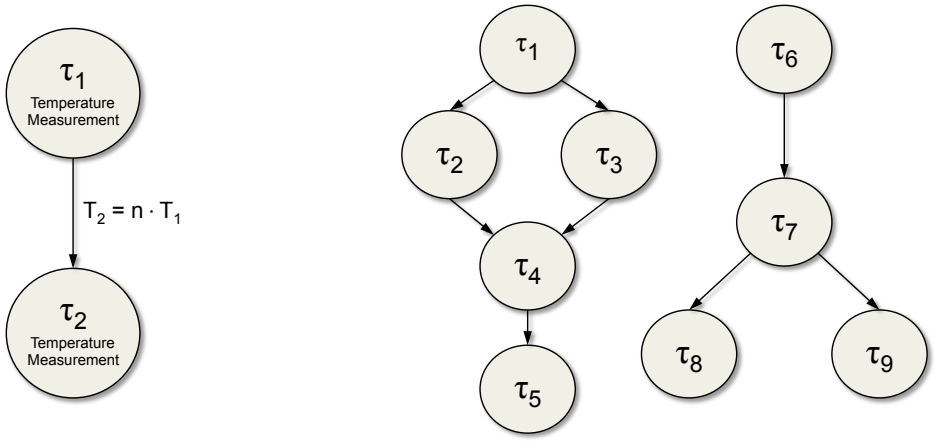
> "*A critical instant for any task occurs whenever the task is requested simultaneously with requests for all higher priority tasks.*"

In any case the release time for each instance must not be larger than the relative deadline, i.e. $R_{i,j} \leq D_i \, \forall j$. If all its instances fulfill this constraint and finish within their deadlines a periodic task τ_i is called *feasible*. A task set Γ is called *schedulable* – or feasible – if all tasks in Γ are feasible.

Precedence Constraints for Periodic Tasks

If we consider a multirate system with different task periods, complex precedence relationships may arise, where n successive instances of a task can precede one instance of another task, or one instance of a task precedes m instances of another task [Cot+02]. Figure 2.3(a) depicts an example for communicating tasks with different periods. Task τ_2 calculates the average value of the input data provided by task τ_1 over n samples and therefore has a period of $T_2 = n \cdot T_1$. To facilitate the description of the precedence constraint problem, in [Cot+02] Cottet et. al propose to consider only *simple precedence constraints*, i.e. if a task τ_i has to communicate the result of its processing to another task τ_j, these tasks have to be scheduled in such a way that the execution of the k-th instance of task τ_i precedes the execution of the k-th instance of task τ_j. Thus, these tasks have the same period ($T_i = T_j$). This means, all tasks belonging to the same connected subgraph of the TDG must have the same period. For instance, on the TDG graph represented in Figure 2.3(b), tasks τ_1 to τ_5 must have the same period and tasks τ_6 to τ_9 also must have the same period. If the periods of connected tasks are different, these tasks will run at the lowest rate sooner or later. As a consequence the task with the shortest period will miss its deadline [Cot+02]. However, in real world systems often functions and tasks with different periods are executed and exchanging data resulting in *complex precedence constraints*. Therefore,

our approach also supports precedence between subgraphs with different periods (cf. Section 4.2.3).



(a) Example of a complex precedence relation of two tasks with different periods.

(b) Example of a TDG composed of two subgraphs with simple precedence constraints.

Figure 2.3: Examples of complex (a) and simple (b) precedence relations in TDGs for periodic tasks [Cot+02].

In [Bla76] Blazewicz stated that if we have to ensure $\tau_i \rightarrow \tau_j$, the release times ($r_i$,$r_j$) and the priorities ($Prio_i$, $Prio_j$) of the tasks must fulfill the following rules:

- $r_j \geq r_i$,
- $\text{Prio}_i \geq \text{Prio}_j$ in accordance with the scheduling algorithm.

In the following section we will present commonly used basic real-time scheduling algorithms where a modification of task parameters shall lead to an execution order that respects the precedence constraints.

2.1.2 Real-Time Scheduling

In this thesis we focus on real-time systems where periodic tasks represent the main workload. A real-time system may contain tasks with different timing constraints. For instance, tasks with different periods have to be considered and dependent tasks additionally imply precedence constraints. Summarized, task executions have to be scheduled properly considering all given constraints, to guarantee that each periodic task instance is regularly activated according to its rate and finished within its deadline. Therefore, in this section we describe well-known and widely used basic algorithms for periodic task scheduling: *Rate Monotonic Priority Assignment* (RM) respectively *Deadline Monotonic Priority Assignment* (DM) and *Earliest Deadline*

First (EDF). The descriptions also provide a schedulability analysis for each algorithm to derive a priori guarantee tests for generic task sets.

Obviously, all these real-time scheduling algorithms strictly rely on priorities. Thus, analogous to the priority Prio_i of a generic task, the priority of a task instance is denoted by $\text{Prio}_{i,j}$. This means, that at any point in time the task instance $\tau_{i,j}$ with the highest priority among all active instances is executed.

Rate Monotonic and Deadline Monotonic Priority Assignment

The Rate Monotonic Priority Assignment (RM) is a so-called *fixed-priority* scheduling algorithm presented by Liu and Layland in [LL73]. Fixed means, that priorities are statically assigned a priori and not modified dynamically during system runtime. RM defines a simple rule that assigns priorities to tasks according to their period, i.e. tasks with higher rates have higher priorities. Since periods are constant, RM is a fixed-priority assignment. Here, it is assumed that relative deadlines of tasks are identical to their periods ($D_i = T_i$). Consequently, RM supports the handling of tasks with different rates in a multirate system. Since it may happen that a task instance is executed when a new instance of a task with higher priority arrives, RM is intrinsically preemptive . In this case the currently running task is preempted and the task with higher priority is executed. Figure 2.4 illustrates an example of a RM schedule in a *Gantt Chart*.[3]

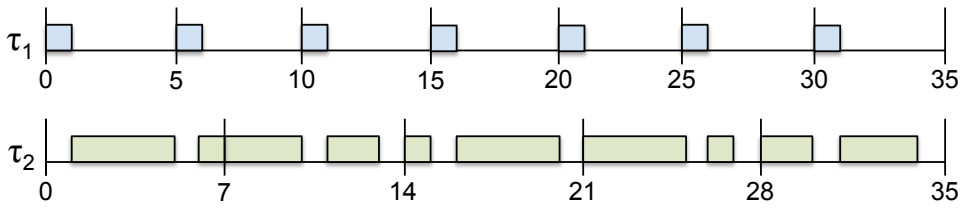


Figure 2.4: Example of a RM schedule for two tasks.

It contains two tasks (τ_1: $T_1 = 5, C_1 = 1$; τ_2: $T_2 = 7, C_2 = 5$) and shows how, by means of RM priority assignment, the instances of τ_2 are preempted by those of τ_1 with higher priority. The given task periods result in the *hyperperiod* $H_\Gamma = 35$. The hyperperiod is the minimum interval of time after which the schedule repeats itself. For such an interval with length H_Γ, the schedule in $[0, H_\Gamma]$ is the same as that in $[kH_\Gamma, (k+1)H_\Gamma]$ for any integer $k > 0$. Hence, for n periodic tasks synchronously released at time $t = 0$, the hyperperiod H_Γ is given by the least common multiple of the periods [But11]:

$$H_\Gamma = \text{lcm}(T_1, \ldots, T_n).$$

[3] A Gantt Chart is a horizontal bar chart that represents a set of tasks. The Gantt chart displays time on the horizontal axis and arranges tasks on the vertical axis [SCR09].

RM is optimal among all fixed-priority scheduling algorithms [But11]. This means, that no other fixed-priority algorithm can schedule a task set that cannot be scheduled by RM. The schedulability test for RM compares the *utilization factor* of a given task set with the utilization factor of the worst possible task set which is still schedulable by RM. This results in the *least upper bound* of utilization (U_{lub}) for all task sets that fully utilize the processor.

Given a set Γ of n periodic tasks, the utilization factor U is the fraction of processor time spent in execution of the task set [LL73]. Summing over the fractions of processor time (C_i/T_i) spent for executing each task τ_i, the utilization factor for n tasks is given by:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}. \tag{2.2}$$

The utilization of the worst case task set is given by $U_{\text{lub}} = n(2^{\frac{1}{n}} - 1)$ which converges towards $U_{\text{lub}} = ln(2) \approx 0.69$ with increasing n [But11]. Since the number n of tasks in the given task set Γ is known a priori, the schedulability test can be performed before system runtime by calculating U_{lub}. It is important to note that $U_{\text{lub}} = n(2^{\frac{1}{n}} - 1)$ is sufficient but not necessary to guarantee the schedulability of a given task set. This means that for task sets with $U_{\text{lub}} \leq U \leq 1$ nothing about the schedulability can be said.

RM allows a task to be executed anywhere within its period, i.e. $D_i = T_i$. However, in some cases a more tightened deadline is necessary. For instance, if a task has a constrained response time which is shorter than its period. Therefore, in [LW82] the *Deadline Monotonic Priority Assignment* (DM) was introduced as an extension of RM to enable scheduling of tasks with relative deadlines less or equal to their period ($D_i \leq T_i$). A sufficient schedulability test for DM derived from RM is given by:

$$\sum_{i=1}^{n} \frac{C_i}{D_i} \leq n(2^{\frac{1}{n}} - 1). \tag{2.3}$$

Nevertheless this test is just sufficient and even more pessimistic for DM than for RM. Therefore, in [Aud+91], [Aud+93] Audsley et. al proposed an efficient method for a sufficient and necessary schedulability test for DM. This test is based on the so-called *Response Time Analysis* (RTA). At its critical instant, the response time R_i of a periodic task τ_i, is given by the sum of its WCET and the interference (I_i) resulting from preemption by higher-priority tasks, i.e. $R_i = C_i + I_i$. To calculate I_i for all higher priority tasks τ_j ($j < i$, $\text{Prio}_j > \text{Prio}_i$) we have to determine the number of interferences given by $\lceil R_i/T_j \rceil$ and the WCET of the respective interference C_j. Based on that, I_i

can be calculated summing up over all interrupting tasks. Therefore, R_i is defined by:

$$R_i = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j. \tag{2.4}$$

Since Equation 2.4 is not analytically solvable for R_i, no simple solution exists for this equation. However, by applying an iterative algorithm we can calculate the least fixpoint of the equation [But11]. If the calculated value is less or equal to the relative deadline, i.e. $R_i \leq D_i$, the feasibility of τ_i can be guaranteed. The schedulability test for a task set Γ is positive if all tasks τ_i are feasible. It can also be performed a priori before system runtime.

RM and DM with Precedence Constraints

Considering precedence constraints for RM and DM the corresponding task parameters have to be modified to realize a proper task prioritization. The basic idea of these modifications is that a task cannot start before its predecessors and cannot preempt its successors. For RM this implies that for a precedence relation $\tau_i \rightarrow \tau_j$ the release time and priority must be modified as [Cot+02]:

- $r_j^* \geq \max(r_j, r_i^*)$, where r_i^* is the modified release time of τ_i, and
- $\text{Prio}_i \geq \text{Prio}_j$ in accordance with RM algorithm.

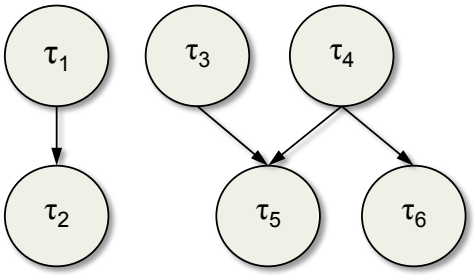


Figure 2.5: Example TDG with a set of six tasks in two subgraphs [Cot+02].

Here it is important to note that, if all tasks of the considered TDG have the same period, RM allows a free choice of priorities that we use to impose the precedence order [Cot+02]. Figure 2.5 shows a TDG for six tasks with simultaneous release times consisting of two subgraphs describing their precedence relations. Table 2.2 provides an exemplary corresponding priority

Task	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6
Priority	6	3	5	4	2	1

Table 2.2: Example of a RM-based priority assignment for the TDG in Figure 2.5 considering precedence constrains.

mapping that considers the given precedence constraints and still schedulable with RM.

For DM scheduling additionally to the release times the relative deadlines have to be modified to respect the priority assignment. This means, that for a precedence relation $\tau_i \rightarrow \tau_j$ the following parameters have to be modified [Cot+02]:

- $r_j^* \geq \max(r_j, r_i^*)$, where r_i^* is the modified release time of τ_i,
- $D_j^* \geq \max(D_j, D_i^*)$, where D_i^* is the modified relative deadline of τ_i, and
- $\text{Prio}_i \geq \text{Prio}_j$ in accordance with DM algorithm.

Consequently, these modifications clearly ensure the preservation of the precedence relations between two tasks.

Earliest Deadline First

In contrast to RM and DM, Earliest Deadline First (EDF) scheduling is a dynamic priority assignment approach presented by Horn [Hor74]. Here every task instance $\tau_{i,j}$ gets a dynamically assigned priority inverse proportional to its absolute deadline $d_{i,j}$. This means, that tasks with shorter deadline will be executed with higher priorities. EDF is a dynamic priority assignment approach, because the absolute deadline of a periodic task τ_i depends on the current j-th instance as:

$$d_{i,j} = \phi_i + (j-1) \cdot T_i + D_i.$$

Moreover, EDF is also intrinsically preemptive, i.e. a currently executed task is preempted if another periodic instance with shorter deadline and higher priority gets activated. This implies that the priorities have to be recalculated and reassigned at each task release. Consequently, the priority of a task may be changed dynamically during runtime. Since EDF does not make any specific assumption on the periodicity of the tasks it can also be applied for the scheduling of aperiodic tasks [But11]. Figure 2.6 depicts an example

EDF schedule for the same two tasks used for the RM example in Figure 2.4. It shows that in contrast to the RM schedule the instances of τ_2 are not preempted that often by τ_1, because the scheduling depends on the absolute deadlines at the release times of the task. Furthermore, in case of equal absolute deadlines the currently running task keeps executing as illustrated by the last task instances of the hyperperiod.

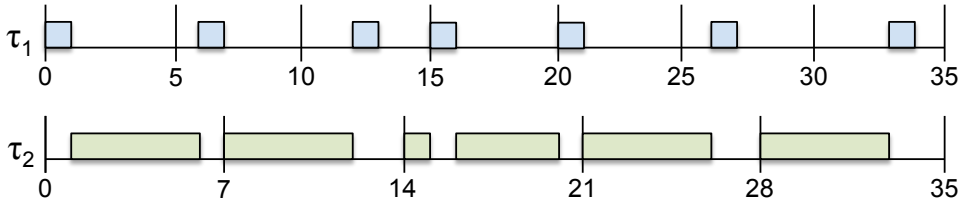


Figure 2.6: Example of a EDF schedule for two tasks.

EDF is optimal among all periodic task scheduling approaches [But11]. This means, that if EDF cannot provide a feasible schedule no other periodic task scheduling algorithm can. A further advantage of EDF is the fact, that – for independent tasks – it guarantees feasible schedules for utilization factors up to 1, i.e. fully utilized processors. Therefore, the schedulability test for a task set scheduled by EDF is given by [LL73; Spu+95]:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1.$$

EDF with Precedence Constraints

In case of EDF a modification of the release times and the absolute deadlines is necessary to handle precedence relations. This technique was originally only proposed for aperiodic tasks with deadline and precedence constraints in [CSB90]. It is optimal in the sense that a valid schedule can be found for the original task set if and only if a valid schedule can be found for the modified task set. Consequently, schedulability can be tested by applying an EDF schedulability test on the encoded task set. The modification technique directly applies to the case of periodic tasks with constrained deadlines and simple precedence constraints remaining optimal [For+10].

Summarized, for two dependent tasks τ_i and τ_j with $\tau_i \rightarrow \tau_j$ the following modifications and conditions must be performed and satisfied to fulfill the given precedence constraints [But11]:

Modification of release times:

$s_j \geq r_j$: Task τ_j must start its execution not earlier than its release time.

$s_j \geq r_i + C_i$: Furthermore, task τ_j must start its execution not earlier than the minimum finishing time of τ_i.

Therefore, the modified release time r_j^* can be set to the maximum of r_j and $r_i + C_i$:

$$r_j^* = \max(r_j, r_i + C_i).$$

Modification of deadlines:

$f_i \leq d_i$: Task τ_i must finish its execution within its deadline.

$f_i \leq d_j - C_j$: Furthermore, task τ_i must finish its execution not later than the maximum start time of τ_j.

Therefore, the modified deadline d_i^* can be set to the minimum of d_i and $d_j - C_j$:

$$d_i^* = \min(d_i, d_j - C_j).$$

By means of these modifications of timing parameters a given set Γ of dependent tasks gets transformed to a corresponding independent task set Γ^* [CSB90]. Because of these modifications EDF with precedence constraints is also called EDF*. Figure 2.7 provides an example for the modifications of task parameters. Here, the release time and the deadline of τ_2 have to be modified.

The modification of the absolute deadlines results in $d_i \leq T_i$ for the tasks in Γ^*. Hence, similar to DM, the schedulability analysis becomes more complex for EDF*. Therefore, Baruah, Rosier, and Howell introduced the *processor demand criterion* (PDC) [BHR90]. It says that if relative deadlines are no longer than periods and periodic tasks are simultaneously activated at time $t = 0$ – i.e., $\phi_i = 0$ for all the tasks – then the number of instances η_i contributing to the demand in a time interval $[0, L]$ is given by [But11]:

$$\eta_i(0, L) = \left\lfloor \frac{L + T_i - D_i}{T_i} \right\rfloor.$$

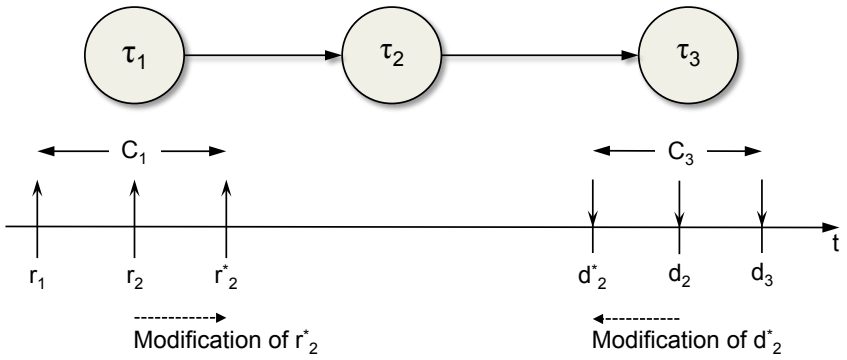


Figure 2.7: Modifications of task parameters for EDF with precedence constraints [Cot+02].

Thus, the processor demand g in the time interval $[0,L]$ can be calculated as:

$$g(0,L) = \sum_{i=1}^{n} \left\lfloor \frac{L+T_i-D_i}{T_i} \right\rfloor \cdot C_i.$$

Consequently, the schedulability test for a synchronous periodic task set with relative deadlines less than or equal to task periods is given by:

$$\forall L > 0 \quad \sum_{i=1}^{n} \left\lfloor \frac{L+T_i-D_i}{T_i} \right\rfloor \cdot C_i \leq L. \tag{2.5}$$

For a reduced complexity, the number of intervals in which the schedulability test has to be checked can be decreased significantly. For this we make use of three observations:

1. Since the tasks are periodic and simultaneously activated at time $t = 0$, the schedule repeats itself after the hyperperiod H_Γ and the schedulability test needs to be checked only for intervals with $L \leq H_\Gamma$.

2. The processor demand $g(0,L)$ is a step function which remains constant if L lies between to deadlines d_k and d_{k+1}. This implies, that if $g(0,L) < L$ holds for $L = d_k$, then it also holds for all L with $d_k \leq L < d_{k+1}$. Consequently, the schedulability test needs to be checked only for values $L = d_k$.

3. Since in any case for the utilization factor $U < 1$ holds, the demand is trivially satisfied after some time instant L^*. It can be shown that the value for L^* can be derived and calculated as [But11]:

$$L^* = \frac{\sum_{i=1}^{n}(T_i - D_i)U_i}{1-U}.$$

Summarizing these observations and considering that it must be checked at least until the largest relative deadline $D_{\max}$, this results in the following schedulability test based on Equation 2.5:

$$\begin{aligned} &\forall L \in \mathcal{D} \quad \sum_{i=1}^{n} \left\lfloor \frac{L+T_i-D_i}{T_i} \right\rfloor \cdot C_i \leq L, \\ &\text{with} \\ &\mathcal{D} = \{d_k | d_k \leq \min[H_\Gamma, \max(D_{\max}, L^*)]\}. \end{aligned} \tag{2.6}$$

A comparison of EDF and RM respectively DM shows that EDF allows a higher processor utilization because RM can only guarantee feasibility for task sets with utilization $U \leq 0.69$. Moreover, in spite of the additional computation needed by EDF for updating the absolute deadline at each job activation, EDF introduces less runtime overhead than RM. Specifically, to enforce the fixed priority order, the number of preemptions utilizing RM is typically much higher than applying EDF. Despite this big advantages the fixed priority algorithms still are widely spread in many real-time systems [Ram+09]. Beside other arguments it is often argued that EDF is more complicated to implement because it dynamically assigns priorities during runtime. However, it has been shown that most of those arguments are not relevant in practical systems [But05].

2.2 Distributed Real-Time Systems

In the previous section we considered real-time systems consisting of a single processing unit. However, nowadays most embedded real-time systems are so-called *distributed real-time systems* [Kop11]. A distributed system is defined to be a system of multiple networked processing elements – generally referred to as *nodes* – cooperating on a common function or set of functions [BW09]. In general, distributed systems have a number of advantages over centralized systems [SZ06]:

Spatial Distribution: The functionality of a embedded system utilizes input data from and generates output data for the environment. This data exchange with the environment is realized through *sensors* and *actuators*. For instance, in the vehicle body of a car these sensors and actuators for common functions are often spread widely. In contrast to a centralized component, a spatial distribution of processing elements near to the involved sensors and actuators can reduce the cabling effort significantly.

Extensibility and Scalability: In a distributed system the extension with additional components and functions is much easier. This can be realized

by connecting additional processing elements instead of replacing a whole centralized system. As a result, this enables scalability and efficient composition of a overall system by combining modular subsystems which implement different functions.

Fault Tolerance: The availability of multiple processors enables the application to become tolerant to processor failures by avoiding a *Single-Point-of-Failure*[4] (SPOF). The system design should enable the implemented functionality to exploit this redundancy [BW09]. Fault tolerance is an important attribute for the reliability and safety of a (distributed) system. In Section 2.3.1 these aspects of system design are described in more detail.

There are two ways of viewing a distributed system: defined by the physical system components (*physical model*) and defined from the view of processing or computation (*logical model*) [Jal94]. The logical model defines the functionality whose correct behavior must be ensured. For a real-time system the correct behavior also depends on timing. The physical model describes the distributed components on which the computations are performed. Beside the advantages described above distributed systems also introduce new challenges. In the following we consider distributed real-time systems requiring a schedulability analysis which has to consider communication delays to fulfill timing constraints of communicating tasks. Moreover, fault tolerance gets more complex, which makes the problem of tolerating faults while respecting timing constraints even more difficult [Cot+02]. Although the availability of multiple processors enables the application to become tolerant of processor failures by avoiding a SPOF, it also introduces the possibility of more faults occurring in the system which would not occur in a centralized single-processor system. These faults are associated with partial system failure and the logical model must either be shielded from them, or be able to tolerate them [BW09]. Therefore, the goal of our work is a fault-tolerant design of distributed real-time systems preserving the correct behavior – including timing constraints – in the logical model despite the failure of components in the physical system.

It is useful to classify distributed systems as either *tightly coupled*, i.e. nodes, have access to a common memory, and *loosely coupled*, i.e. they do not [BW09]. In a tightly coupled system synchronization and communication can be realized by techniques based on the use of shared variables, whereas in a loosely coupled system some form of message communication is indispensable. In this thesis the term ”distributed system” refers to a loosely coupled topology communicating over a network. Furthermore, we assume that each node has its own clock and that these clocks are synchronized for

[4] Any single component within a system whose failure will lead to a failure of the system is called a Single-Point-Of-Failure [Pra96].

real-time issues. The manner in which the different nodes in a system are connected is called the *network topology* [Jal94]. A quite popular topology of distributed systems is a *bus*, to which all the different networks are connected, instead of a point-to-point communication network. An additional classification of a distributed system can be based on the variety of processors in the network topology. In a *homogeneous* system all processors are of the same type; a *heterogeneous* system contains processors of different types [BW09]. As we show in Chapter 4 our approach supports homogeneous as well as heterogeneous systems to a certain extent.

2.2.1 Task Scheduling in Distributed Real-Time Systems

Task scheduling in distributed systems has to deal with two levels. The *local scheduling* on the local level of each processor and the *global scheduling* on the level of the allocation of tasks to the processors [Cot+02]. Task allocation is often also called task mapping. Local scheduling means the assignment of the processor to tasks. For a real-time system this implies the fulfillment of timing constraints. Therefore, scheduling algorithms like the ones described in Section 2.1.2 can be utilized. Global scheduling means the allocation of tasks to the processors composing the distributed system. Obviously, the task mapping must be performed in such a way that the local scheduling can guarantee the fulfillment of the tasks timing constraints. In general, the problem of finding an optimal feasible allocation of n tasks to p processors is known to be NP-hard [Bur91]. Therefore, this problem is often addressed by searching for solutions which respect the initial constraints as much as possible, and then choosing the best solution, if several solutions are found [Cot+02].

Task mapping can be performed as *static allocation* or *dynamic allocation*. At static allocation, there cannot be any additional allocation or reallocation of tasks during system runtime, i.e. the initial allocation of tasks is fixed. At dynamic allocation the scheduling strategy assigns a node to a task guaranteeing the timing constraints when a task arrives. Dynamic allocation can increase the fault tolerance of a distributed system, e.g. in case of a node failure [Cot+02]. However, it can also lead to non-determinism or missing of timing constraints if the allocation is performed at runtime. For instance, the global scheduling algorithm finds no feasible solution because of its additionally required computation time. Our approach combines the advantages of both allocation concepts. On the one hand it keeps the determinism of the static allocation and guarantees the timing constraints; on the other hand it increases the fault tolerance by including possible reallocations in the predetermined solution to compensate node failures. Additionally, global scheduling may partition the computation of one task to different processors by utilizing *task migration*, i.e. a task can change node during its execution. Task migra-

tion consists of transferring its context[5], which continuously changes during execution, and, if required, its code[6], which does not change [Cot+02]. To minimize the migration time, the code of the tasks can be replicated on the nodes on which it shall be executed. Thus, in the case of migration, only the context must be transmitted.

2.2.2 Communication in Distributed Real-Time Systems

In addition to reliability and determinism, the timeliness of transmitted data has to be ensured for distributed real-time systems. This is the most important difference between a real-time communication system and a non-real-time communication system. Consequently, the real-time communication infrastructure utilized in such a system has to fulfill numerous requirements:

Reliability: To improve the communication reliability for distributed real-time systems several different techniques are utilized. For instance, the use of robust channel encoding, the use of error-correcting codes for forward error correction, or the transmission of replicated messages over redundant communication channels. In many non-real-time communication systems, reliability is achieved by time redundancy, i.e. retransmission of a lost message. However, this strategy is mostly inappropriate especially for hard real-time systems. If you consider a scenario, where a sensor component sends periodically, e.g. every millisecond, a message with sensory data to a control component. In case the message is corrupted or lost, it makes more sense to wait for the next message that contains a more recent observation than to retransmit the lost message with the outdated observation [Kop11].

Determinism: Determinism is a major property of a real-time system. Therefore the communication behavior must be deterministic, too. This means, that the order of messages must be the same on all communication channels and the arrival times of replicated messages over redundant channels are similar or at least close together.

Short Transmission Latency: A *distributed real-time transaction* generally starts with the reading of an input value via sensor and terminates with the output of the results to an actuator [Kop11]. The duration of such a distributed real-time transaction results from the time needed for the computations within the components and the time needed for the message transmission between the involved components. This duration is called *end-to-end delay* because it starts at one end of the system (source) and ends at the other end (destination) [Cot+02]. Since the

[5] Data, processor registers, etc.
[6] Instructions composing the task program

end-to-end delay should be as small as possible, the worst case transmission latency of a message should be small for real-time communication. The transmission latency is also called *transmission delay* or *communication delay*.

Clock Synchronization: In a distributed system, the temporal accuracy can only be checked if the duration between the time of the input value acquisition observed by the sensor node, and the data output determined by the actuator node, can be measured. This requires the availability of a global time base of proper precision among all involved nodes. Consequently, the communication system must establish such a global time and synchronize the nodes.

Real-Time Messages

Similar to the definition of tasks two main classes of messages can be defined for real-time communication: periodic and aperiodic messages. An aperiodic, also called asynchronous message m_i is generated by an aperiodic task and is just characterized by its length L_i and its deadline D_i. Periodic, also called synchronous messages are generated and consumed by periodic tasks. Their characteristics are similar to the characteristics of their respective source tasks [Cot+02]. Hence, a set $\mathcal{M}$ of periodic real-time messages can be characterized by the following set of parameters illustrated in Figure 2.8:

m_i: A generic message, where different instances of this message may exist over time. A message can also be notated as $m_{(\tau_{tx},\tau_{rx})}$. This notation contains the sender task τ_{tx} and the receiver task τ_{rx} of a message m_i.

$m_{i,j}$: The j-instance of m_i.

T_{m_i}: The period of m_i at which instances of a message m_i are generated. Thereby, this period depends directly on the period of the sender task.

L_{m_i}: The *data length* of m_i which represents the original data of m_i. This means that additional overhead depending on the communication protocol has to be considered for the transmission time and the required *payload length* of m_i.

$r_{m_{i,j}}$: The release time of $m_{i,j}$ is an absolute value and specific for each instance. We define that a sender task τ_{tx} must finish its computations before a message can be released. Consequently, $r_{m_{i,j}} = f_{tx}$, the finishing time of τ_{rx}, always holds.

$d_{m_{i,j}}$: The absolute deadline of $m_{i,j}$ is a specific value for each instance. We define that m_i must be transmitted before the receiver task τ_{rx} can start it execution. Therefore, $d_{m_{i,j}} = s_{rx}$, the start time of τ_{rx}, always holds.

$D_{m_{i,j}}$: The relative deadline of $m_{i,j}$ represents the time interval $\Upsilon_{m_{i,j}} = [r_{m_{i,j}}, d_{m_{i,j}}[= [f_{tx}, s_{rx}[$, i.e. the *available transmission time interval* for $m_{i,j}$.

$s_{m_{i,j}}$: The transmission start time of $m_{i,j}$ with $s_{m_{i,j}} \geq r_{m_{i,j}}$ is an absolute value, which is specific for each instance.

$f_{m_{i,j}}$: The transmission finishing time of $m_{i,j}$ with $f_{m_{i,j}} \leq d_{m_{i,j}}$ is an absolute value, which is specific for each instance.

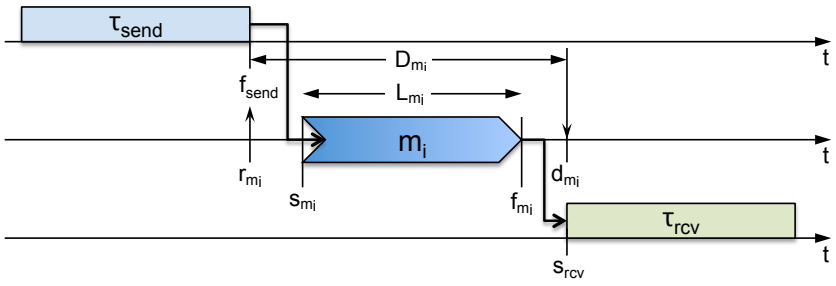


Figure 2.8: Parameters of a periodic real-time communication message.

The tasks executed within a distributed real-time system are spread over the different nodes in the network. Depending on their distribution, these tasks exchange data by means of either local *intra-node communication* or remote *inter-node communication*. Consequently, the resulting communication delay of a message between two tasks depends on their placement. The transmission delay between two tasks placed on the same machine is often considered to be negligible, since it just implies access a data structure shared by the communicating tasks, i.e. shared variables or similar concepts. The communication delay between distant tasks, i.e. tasks placed on different nodes depends on the message scheduling and the properties of the message and the given communication infrastructure, e.g. message size and transmission rate [Cot+02].

Similar to the occurrence of triggering signals and depending on the temporal properties of the communication infrastructure messages can also be distinguished as event-triggered and time-triggered [Kop11]:

Event-Triggered Messages: The messages are produced sporadically whenever a significant event occurs at the sender. There is no minimum time between messages established. Furthermore, no temporal guarantees about the delay between sending a message and receiving a message can be given. If the sender produces more messages than the communication system can handle messages may get lost.

Time-Triggered Messages: Sender and receiver agree a priori on the exact instants (points in time) when messages are sent and received. Therefore, an appropriate communication system must establish a global time base and synchronize the nodes. Consequently, the communication infrastructure guarantees that the messages will be delivered at the specified instant based on the global time.

Scheduling of Real-Time Messages

The scheduling of real-time messages shall allocate the medium between several nodes in such a way that the timing constraints of messages are respected. Thus, according to the system properties and the timing constraints of a given message set we can distinguish three different scheduling strategies [Cot+02]:

Guarantee Strategy: If no failure of the communication system occurs, any message accepted for transmission is sent by respecting its timing constraints. Therefore, it is also called *deterministic strategy*. A deterministic strategy is generally applied for messages with critical timing constraints.

Probabilistic Strategy: The timing constraints of messages are guaranteed at a probability known a priori. With this strategy, the messages can miss their deadlines. Hence, it is utilized for messages with firm timing constraints whose missing of deadlines have no serious consequences.

Best-effort Strategy: Here, no guarantee is provided for the delivery of messages. The communication system will try to do its best to guarantee the timing constraints of the messages. This strategy is applied to messages with soft timing constraints or without timing constraints.

In this thesis we focus on safety-critical systems with hard real-time constraints. Best-effort and probabilistic strategies cannot be applied in such

systems, since the uncontrollable delay or loss of messages must be avoided. Hence, we have to consider a communication infrastructure which offers a guarantee-based scheduling strategy. Time-triggered communication is deterministic and supports such a strategy. Moreover, it is well suited for the implementation of fault tolerance [Kop11]. For this, time-triggered protocols establish a global time base among all communicating nodes. Based on this global time a time interval called *slot* is assigned to every message. The start of transmission of a message is triggered exactly when the global time reaches the start of the assigned slot. This means, that for this time slot the communication system is an exclusive resource for the sender node. This avoids collision of messages and guarantees determinism for the communication. Consequently, in the avionic and automotive domain, where safety is of high importance, time-triggered protocols are utilized for communication in distributed systems to fulfill hard real-time constraints. For instance, the *Time Triggered Protocol* (TTP) [KG94] is deployed in the A 380 and the Boeing 787 aircraft and other aerospace and industrial control applications [Kop11]. In the automotive domain, the *FlexRay* protocol [Fle05a], which shares similar basic concepts with TTP, is the emerging standard for safety-critical systems. Compared to FlexRay, TTP provides no flexibility regarding the multiple transmission slots for a single sender node [Par07]. However, we need a high flexibility for the transmission assignment in our design approach we present in Chapter 4. Furthermore, we apply it to the automotive domain in Chapter 5. Therefore, we utilize the FlexRay protocol for our work in this thesis.

2.2.3 FlexRay Communication Protocol

The FlexRay protocol is a communication protocol for distributed real-time systems. It was developed by the FlexRay consortium which was founded in 1999. Initially, it was introduced exclusively for the automotive domain. However, there are efforts to introduce FlexRay also for other industrial applications [SJ08]. The aim was to develop a flexible and fault-tolerant communication protocol. A FlexRay network consists of several FlexRay nodes, connected to a communication *bus*[7]. Such a network is called FlexRay *cluster* and represents a distributed real-time system. A FlexRay cluster has one (*single-channel*) or two communication channels (*dual-channel*). Each channel provides transmission rates up to 10 Mbit/s[8]. The current version (V3.0.1) of the protocol specification added the support of 2.5 and 5 Mbit/s [Fle10]. In a dual-channel topology, each node can be connected to either one or both channels. Since we want to increase fault tolerance, we consider a bus topol-

[7] Beside a bus topology, FlexRay also supports star and hybrid topology. However, in this thesis we focus on bus topology.

[8] Due to two additional protocol bits per byte, the transmission of 1 byte needs 1μs at a transmission rate of 10 Mbit/s.

ogy with a dual-channel redundant system. This means, in the following, we assume that every FlexRay node is connected to both channels of the bus to enable concurrent transmission of redundant data, in case that one of the two channels fails.

FlexRay Node

As shown in Figure 2.9, a FlexRay node basically consist of a *host*, a *communication controller* (CC) and one or two bus drivers. A host is a microcontroller hosting one or more applications outside the scope of FlexRay using the FlexRay communication system to communicate with other applications. Since FlexRay originates from the automotive domain a FlexRay node is also often called *electronic control unit* (ECU) which is a hardware component containing a host processor to execute application software and a communication module [Fle05b]. Therefore, we use these terms synonymous in this thesis. The communication controller includes the FlexRay *protocol engine* that implements the majority of the FlexRay protocol including transmitting and receiving frames, maintaining clock synchronization with other nodes in the cluster, and error-detection. Moreover, it provides the *controller host interface* (CHI) for the host to configure, control, and monitor the protocol engine and exchange message data and status between the application and the protocol engine. A FlexRay node contains a bus driver for each implemented channel providing physical access to the bus. The purpose of a bus driver is to transmit and receive data over the bus. Thus, several manufacturers also call it FlexRay *transceiver* [Sem12], [Vec12]. Here, we consider nodes with dual-channel redundancy, i.e. one driver for channel A and one for channel B. To increase fault tolerance we also consider *bus guardians* which are optional in a FlexRay node. Bus guardians are small devices, which are placed between the communication controller and the bus driver. They avoid so-called *babbling-idiot* failures, where a certain defect node ignores the given slot assignments, transmits all the time and potentially consumes all slots [Fle05c].

Another important component we make use of for our work are the *message buffers*. Message buffers are implemented as shared memory between the host and the communication controller in the node as depicted in Figure 2.9. The message buffer concept essentially enables the application associated with a FlexRay node to [RS06]:

- Put data to be transmitted into a region of shared memory (*transmit buffer*) and be sure that the corresponding FlexRay frame will be transmitted on the FlexRay bus in its assigned slot.

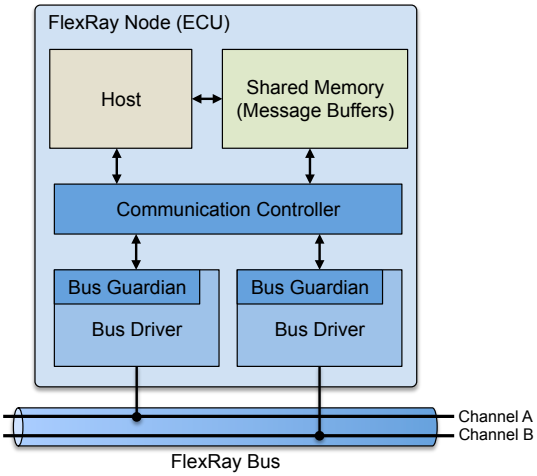


Figure 2.9: Structure of a FlexRay node (ECU).

- Get notified that a valid FlexRay frame has been received and retrieve the corresponding data from its assigned region in the shared memory (*receive buffer*).

Summarized the message buffer functionality of FlexRay can be described as [RS06]:

> "*In a FlexRay node, message buffers are the means by which the application can decouple its message transmission and reception from the actual FlexRay protocol, allowing asynchronous operation of the application with respect to the timing of FlexRay bus.*"

This means, that the message buffer concept of FlexRay allows to transmit and receive a message in any available slot within the available transmission time interval of a message. The data will be stored by the sender task in the transmit buffer of the sender node until the slot starts. After the reception by the receiver node it will be stored in the receive buffer until the receiver task starts and retrieves the data. As we present in Chapter 4, this decoupling of application and communication increases the flexibility of our approach.

Message Scheduling

The FlexRay protocol applies a time-triggered message scheduling. Therefore, the bus write access is scheduled by a *time-division multiple access* (TDMA) scheme, which means that write accesses are only allowed in exactly specified and pre-defined time slots. Each node has a configurable amount of assigned slots in which it is allowed to transmit a message. This *slot-to-node* (slot-to-ECU) assignment is fixed for the whole system runtime

and cannot be changed without a bus restart. This ensures a static and deterministic timing behavior of the communication. Obviously, the read access to a slot does not influence the scheduling. Thus the number of receiver nodes reading from one slot can be configured freely as required. The writing and reading accesses can be summarized in a *communication matrix* (COM matrix) that clarifies the slot-to-node assignments for the whole schedule [SZ06]. Table 2.4 gives an example of a COM matrix for a FlexRay cluster with four ECUs. It illustrates the assignment of each slot for transmission to just one sender (tx) and the arbitrary number of assigned receivers per slot (rx). For instance, in slot s1 a message is transmitted by ECU 1 and received by ECU 2 and ECU 3 while ECU 4 is not involved in this communication. From such a COM matrix the FlexRay schedule can directly be derived.

Slot	s1	s2	s3	s4	...	
ECU 1	**tx**	rx	rx	-	...	
ECU 2	rx	**tx**	rx	-	...	
ECU 3	rx	-	**tx**	rx	...	
ECU 4	-	rx	-	**tx**	...	

Table 2.4: Example of a communication matrix.

Communication Cycle

Similar to the hyperperiod at periodic task set scheduling the FlexRay message schedule repeats itself over time. Therefore, the FlexRay protocol makes use of a recurring *communication cycle*. More specifically, FlexRay communication is based on a recurring sequence of 64 communication cycles numbered from 0 to 63. Each node in the cluster keeps track of the current cycle count independently in a local variable which counts from 0 to 63, then resets to 0 and increments again. Although each node tracks the cycle count independently, the current cycle count is the same for all nodes in the cluster. The length of a communication cycle is initially configured and fixed. It must be between 10μs and 16 ms. Figure 2.10 illustrates the basic segments of a communication cycle. It contains a *static segment*, an optional *dynamic segment*, an optional *symbol window*, and a *network idle time* (NIT) which are described in the following.

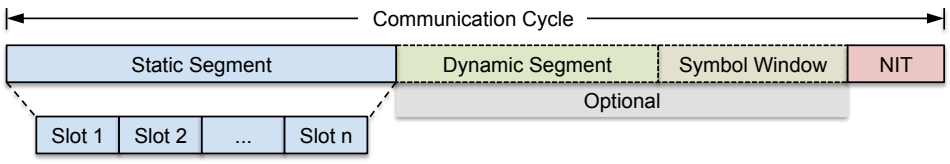


Figure 2.10: FlexRay Communication Cycle.

Static Segment The static segment is present in every communication cycle and realizes the TDMA-based communication scheme based on statically assigned communication slots. It consists of a fixed initially defined number of equally sized *static slots*. Consequently, the length of the static segment depends on the configured number and size of the slots. The number of slots in the static segment must be at least 2 and at most 1023. There are always at least two static slots in the static segment because a cluster has at least two nodes which must be able to exchange messages for start-up and clock synchronization. The maximum of 1023 was defined and results from the relation of maximum cycle length (16.000μs) to minimum slot length. Theoretically, considering a payload of zero, the minimum slot size would be 12μs resulting in 1333 static slots. Practically, the slot length must be configured to be at least 13μs to compensate clock drifts and signal delays. This would allow a maximum of 1230 slots. However, communication with no payload is useless and higher payload results in longer slots and less slots per cycle. Considering a payload size of 4 bytes the allowed number of slots is reduced to less than 1000. Therefore, the defined maximum of 1023 static slots is reasonable. Moreover, considering a transmission rate of 10 Mbit/s, this also defines the maximum available payload per slot θ as [Rau08]:

$$\text{maxPayload}(\theta) = \text{size}(\theta) - 13\,\text{bytes}.$$

In general, the payload size for each slot can be between 0 and 254 bytes and every slot has the same payload length. Similar to the cycle count, each node in the cluster keeps track of the current slot count to keep synchronous. In the static segment, the current slot count is always the same for channel A and channel B, since the number of slots and slot time duration are the same for each channel [RS06]. Due to its deterministic timing behavior the static segment is very well applicable to safety-critical real-time systems.

Dynamic Segment The dynamic segment is optional in the communication cycle and is used for event-triggered communication. In contrast to the static segment there is no fixed slot-to-node assignment in the dynamic segment. The bus write access is scheduled by a *flexible time-division multiple access* (FTDMA) scheme based on *minislots*. Therefore, the dynamic segment also consists of a fixed initially defined number of equally sized minislots. However, minislots are much smaller than static slots. Hence, there are up to 7986 minislots allowed. Minislots can be combined to *dynamic slots* to transmit frames with variable length. Their numbering continues from the last slot number in the static segment and they are assigned to frames by means of their ID. For this purpose all nodes compare their internal slot counter to the present frame-IDs. If both values match a dynamic communication starts within the current minislot. Obviously, the length of a dynamic slot depends on the size of the payload in the associated frame. The maximum payload

length in the dynamic segment is 254 bytes, too. However, the actual transmitted payload length may vary from one dynamic slot to another [RS06]. The comparison of slot counter and frame-ID implements a prioritization of frames. A frame with a lower ID will be transmitted with a higher probability than a frame with a higher ID. This means that the dynamic segment realizes a probabilistic scheduling strategy which is not applicable for safety-critical systems with hard real-time constraints. Thus, we consider a *pure static* configuration, i.e. a FlexRay cycle without dynamic segment, for our work in this thesis [Fle05b].

Symbol Window The symbol window is also optional in the communication cycle and is used to transmit FlexRay-defined symbols. Symbols are communication elements with protocol internal information. They are used to exchange information about the status and functionality of the nodes in the cluster without interfering the message communication. There are three different symbols defined in FlexRay [Rau08]:

- Collision avoidance symbol (CAS),
- Media test symbol (MTS), and
- Wake-up symbol.

The CAS and the wake-up symbol are exclusively used at the start-up respectively wake-up phase. Media test symbols are transmitted to test bus guardians.

Network Idle Time The NIT is a communication-free period at the end of the communication cycle and is basically used by each node to calculate and apply necessary local clock corrections to keep the clocks synchronized [Fle05b]. Since no communication is performed during the NIT it can also be used for other internal cluster and communication cycle related operations. Depending on the configuration of the FlexRay bus, the duration of the NIT may be between $2\mu s$ and $4830\mu s$ [Fle05a]. To increase the available bandwidth and flexibility for our approach we keep the NIT as long as necessary, but also as short as possible.

The allowed number of slots in a communication cycle is limited. Moreover, their number and size is initially defined and fixed. To increase the flexibility and the utilization of the FlexRay bus *cycle multiplexing* can be applied, as proposed in [AUT13j]. Cycle multiplexing means, that an associated node may transmit different frames in different cycles. The cycle multiplexing of a frame is defined by the *base cycle* and the *cycle repetition*. The base cycle defines the offset in cycles for the first occurrence of the respective frame. The

cycle repetition denotes the recurrence of a frame in the multiplexing. Initially, the value of the cycle repetition was defined to be 2^n with $n \in \{0,\ldots,6\}$ to allow a periodic occurrence in the 64 cycles [Luk+09]. Thus, for a given base cycle B and cycle repetition Rep, the cycle numbers where a frame is transmitted can be calculated as a so-called *cycle count filter*:

$$\text{cycle number} = (B + n \cdot Rep) \bmod 64, \text{ with } n \in \mathbb{N}_0,\ B < Rep.$$

Cycle multiplexing significantly increases the flexibility of the communication system. For instance, it can be applied if there are tasks that have longer message transmission periods than the configured cycle length. Considering a cycle length of 2 ms, a task with a period of 8 ms could transmit its message each fourth cycle.

The current protocol specification of FlexRay further increases the flexibility for the message scheduling by adding *slot multiplexing* and extending the possible cycle repetitions by the values of 5, 10, 20, 40, and 50 [Fle10]. Cycle multiplexing combined with slot multiplexing does not only support the transmission of different frames from the same node but also the assignment of a slot to different nodes in different communication cycles. In that regard, however, the configuration of the slot-to-node assignments must ensure that the transmissions in the cluster are conflict-free to guarantee a deterministic communication behavior. The utilization of cycle and slot multiplexing results in an advanced COM matrix. Table 2.5 gives an example of a COM matrix with cycle and slot multiplexing. In this example cycle multiplexing is utilized to transmit different frames from ECU 1 in slot 0, i.e frame *a* in cycle 0 and frame *e* in cycle 1. Slot multiplexing is utilized for slot 2. In cycle 0 it is assigned to ECU 3 and in cycle 1 to ECU 4.

Cycle	0					1					...
Slot	0	1	2	3	...	0	1	2	3	...	...
ECU 1	**tx:a**	rx	rx	-	...	**tx:e**	rx	rx	-	...	...
ECU 2	rx	**tx:b**	rx	-	...	rx	**tx:b**	rx	-	...	...
ECU 3	rx	-	**tx:c**	rx	...	rx	-	-	rx	...	...
ECU 4	-	rx	-	**tx:d**	...	-	rx	**tx:f**	**tx:d**	...	...

Table 2.5: Example of a communication matrix with cycle and slot multiplexing.

Frame Format

Figure 2.11 gives an overview of the FlexRay frame format. The frame consists of three segments, namely *header segment*, *payload segment*, and *trailer segment*. In the following we provide a brief description of these segments.

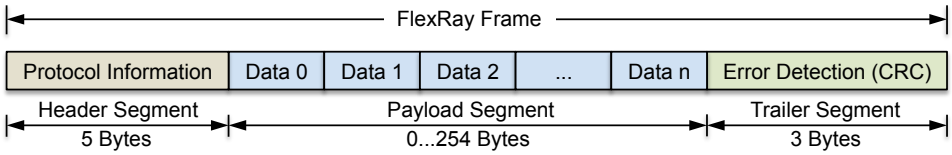


Figure 2.11: FlexRay Frame Format.

Header Segment The header segment consists of 5 bytes with protocol relevant information. These bytes contain several control bits like null frame indicator, sync frame indicator, and startup frame indicator. Another protocol relevant information in the header segment is the frame ID. The unique frame ID determines the slot in which a frame (contained in a message buffer) will be transmitted on the FlexRay bus [RS06]. Furthermore, a cycle count field is present in the header segment. The cycle count indicates the transmitting node's view of the value of the cycle counter at the time of frame transmission [Fle05a]. This information can be used for the avoidance of errors due to asynchronous local cycle counts of nodes. Additionally the header contains information about the length of the payload segment and a *cyclic redundancy check code* (CRC) for error detection. For a complete and detailed description of the control bits and the other parts of the header segment refer to [Fle05a].

Payload Segment The payload segment consists of 0 to 254 bytes, or more precisely 0 to 127 two-byte words of data. Since the payload length in the header segment is given as number of two-byte words, the payload segment always contains an even number of bytes. The bytes of the payload segment are identified numerically, starting at 0 for the first byte after the header segment and increasing by one with each subsequent byte. The individual bytes are referred to as "Data 0", "Data 1", "Data 2", etc. up to the configured number of bytes. Each slot respectively frame in the static segment has the same payload size. Thus, FlexRay utilizes *padding bytes*[9] to fill the remaining bytes, if the message to transmit is smaller than the configured payload size. This is also applied for null frames which are utilized by the protocol and contain just zeros and no application-relevant data.

Trailer Segment The FlexRay trailer segment contains a single field 24-bit (3 bytes) CRC for the frame. This cyclic redundancy check code is computed over the header segment and the payload segment of the frame for transmission error detection. The Frame CRC calculation is done inside the communication controller before transmission or after reception of a frame. When a communication controller receives a frame it shall perform the frame CRC computations based on the header and payload field values received and check the computed value against the frame CRC value received in the frame.

[9]This means, that the remaining words are set to the padding pattern "0x00" [Fle05a].

If both values match the error check passes, otherwise it fails. For a complete and detailed description of the error detection refer to [Fle05a].

2.2.4 Timing Constraints for Distributed Real-Time Systems

As described in Section 2.1 the execution of a real-time system functionality is triggered by its environment. Thereby, the triggering signal is associated with an event occurrence. In time-triggered real-time systems these triggering signals respectively events occur at specified periodic points in time. A distributed real-time transaction typically starts with the event of periodic data acquisition from sensors and ends with the event of data output to actuators. The duration between these two events is called end-to-end delay (cf. Section 2.2.2). To guarantee a correct behavior a distributed real-time system has to fulfill hard timing constraints regarding event occurrences and delays. Thus, the modeling and analysis of timing is an important part of the design.

The ITEA2 project TIMMO and its follow-up TIMMO-2-USE[10] developed solutions for timing modeling and analysis in automotive system design. Although these projects were initially addressing the automotive domain, their results are generally applicable for embedded real-time systems. A major result of these projects is the *Timing Augmented Description Language* (TADL) respectively its extension TADL2 that offers capabilities for the modeling of timing information and constraints [Per+12b]. In [Klo+09] and [Klo+10b] we applied this language to model and analyze timing properties of a Steer-By-Wire system which we designed and implemented for the validation of the TIMMO and TIMMO-2-USE project results.

TADL2 is a language for imposing timing constraints on events and their occurrences in structural system models on different levels of abstraction [Per+12b]. Beside events, TADL2 also defines the *event chain* which indicates causal relationship between a *stimulus* and a *response* event [Blo+12a]. It offers several basic and derived timing constraints for events and event chains. Since we consider time-triggered real-time systems in this thesis we focus on the *periodic constraint* and particularly on the *delay constraint*. A periodic constraint describes an event that occurs periodically and constrains the distance of its occurrences. An example for that is the periodic activation of a task. A delay constraint imposes limits between the occurrences of a *source* event and a *target* event [Per+12b]. For instance, in a distributed real-time system the data input can be modeled as a source event and the data output as a target event. These events also represent the stimulus and response event of an event chain. Based on that, we can constrain the delay defining a *maximum end-to-end delay* to guarantee a correct system behavior.

[10] TIMing MOdel - TOols, algorithms, languages, methodology, and USE cases [Con12].

In the following we provide exemplary TADL2 timing constraints for the Brake-By-Wire system we defined in [Blo+12b]. Beside other timing constraints, the authors define a periodic constraint `TrPosition` for the periodic reading of the wheel position by a task `WheelCtrl` and a delay constraint for the subsequent sending of a data value `SpeedFactor` by the tasks `TransmissionCtrl` and `Adapter`. TADL2 defines formal timing specifications representing internal time bases based on a common external physical time dimension [Blo+12a].

```
1  TimingSpecification ts1 {
2    Dimension physicalTime {
3      Units {
4        micros{factor 1.0 offset 0.0},
5        ms{factor 1000.0 offset 0.0 reference micros}
6        second{factor 1000000.0 offset 0.0 reference micros}
7      }
8    }
9    TimeBase universal_time {
10     dimension physicalTime
11     precisionFactor 0.1
12     precisionUnit micros
13   }
14 }
```

Listing 2.1: Example of a TADL2 timing specification.

Listing 2.1 provides an exemplary dimension declaration in TADL2 from [Blo+12a]. A list of units and attributes for the conversion between microsecond (`micros`), millisecond (`ms`) and second (`second`) in the `physicalTime` dimension are given. A time base `universal_time` is specified which is the internal reference time base for the whole system. The `precisionFactor` and `precisionUnit` define that `universal_time` is able to specify values with a precision of $0.1\mu s$. Based on this, timing constraints are formalized.

```
1  Event PositionRecvByWheelCtrl {
2  //Wheel position received by WheelCtrl task event
3  }
4
5  PeriodicConstraint TrPosition {
6  //Periodic constraint for reading of wheel position
7    event PositionRecvByWheelCtrl
8    period = 2.0 ms on universal_time
9    minimum = 0
10   jitter = 0
11 }
```

Listing 2.2: Example of a TADL2 periodic constraint.

Listing 2.2 shows the periodic constraint `TrPosition` for the `WheelCtrl` task. It defines that the distance between repeated occurrences of the event `PositionRecvByWheelCtrl` may be up to 2*ms*, i.e. the wheel position has to be read periodically every 2*ms*. The lower limit `minimum`, which is also possible to specify, defaults to 0 if not present [Per+12b]. Also the parameter `jitter` is set to 0 which means that no allowed value for the deviation of the period is defined.

```
1  Event SpeedFactorSentByTransmCtrl {
2  //SpeedFactor sent by TransmCtrl task event
3  }
4
5  Event SpeedFactorRecvByAdapter {
6  //SpeedFactor sent by TransmCtrl task event
7  }
8
9  Event SpeedFactorSentByAdapter {
10 //SpeedFactor sent by Adapter task event
11 }
12
13 DelayConstraint SpeedFactorDelay {
14 //Delay constraint between source/stimulus
      (SpeedFactorSentByTransmCtrl) and target/response
      (SpeedFactorSentByAdapter) events
15 source SpeedFactorSentByTransm
16 target SpeedFactorSentByAdapter
17 lower = 0
18 upper = 1.0 ms on universal_time
19 }
```

Listing 2.3: Example of a TADL2 delay constraint.

Listing 2.3 shows a delay constraint `SpeedFactorDelay` for a source event `SpeedFactorSentByTransm` and target event `SpeedFactorSentByAdapter`. The events are also the stimulus and response of an event chain which is a composition of two included event chains. A sequence of event chains is constructed such that the response event of the first event chain is the stimulus of the second one [Blo+12a]. The delay constraint defines that the delay between the sending of the data value `SpeedFactor` including the reception by the `Adapter` task may not exceed 1*ms*. Here, the value for `lower` is also not specified.

The examples above show the capabilities of TADL2 to define timing constraints on structural models of distributed real-time systems. We utilize it to specify timing constraints for the software architecture model by augmenting the TDG as input for our design approach in Section 4.2.3.

2.3 Fault-Tolerant System Design

Beside many functional and performance objectives the design of a safety-critical real-time system must satisfy further requirements. Fault tolerance is an important system attribute, which is utilized to fulfill such requirements. This section provides a description of the fundamental terms concerning dependability, fault types, and means for fault-tolerant system design.

2.3.1 Main Attributes of Dependable Systems

The *dependability* of a system defines the quality of service provided by this system. It encapsulates different concepts to quantify the dependability of a system. These concepts are reliability, availability, saftey, maintainability, performability, and testability [Pra96]. For safety-critical systems, the main attributes for the dependability are *reliability* and *safety*. In [RLT78] Randall et al. defined the reliability of a system as:

> "*A measure of the success with which the system conforms to some authoritative specification of its behavior.*"

In the ideal case this specifications should be complete, consistent, comprehensible, and unambiguous [BW09]. Furthermore, timing constraints are an important part of the specification for a real-time system. When the actual behavior of a system deviates from the specifications, this is called a system *failure* [RLT78].

Failures result from unexpected problems within the system which eventually imply an undesired and unexpected external behavior of the system. These (internal) problems are called *errors* and their mechanical, electronic, or algorithmic (external) causes are termed as *faults* [BW09]. Because systems are often composed of components which are themselves sub-systems, a failure of one sub-system may imply a chain reaction – i.e. failure $\rightarrow$ fault $\rightarrow$ error $\rightarrow$ failure $\rightarrow$ fault – finally resulting in a malfunction of the whole system. Figure 2.12 depicts these impairments potentially negatively influencing the dependability of a system. Beside reliability ensuring the fulfillment of the specified functionality, safety issues are a very important aspect for the dependability of a system.

Safety can be defined as [Lev86]:

> "*Freedom from those conditions that can cause death, injury, occupational illness, damage to (or loss of) equipment (or property), or environmental harm.*"

However, this definition considers most systems as unsafe which have an element of risk associated with their usage [BW09]. Therefore, the safety of a system is often considered in terms of *mishaps*. A mishap is an unplanned event or series of events that may result in one or more of the conditions mentioned above. To improve safety, a system should be free from these conditions, because they can cause damages, accidents, injuries, or even death. Summarized, safety defines the probability for the non-occurrence of (catastrophic) conditions leading to mishaps, whether the intended function is performed correctly or not [BW09].

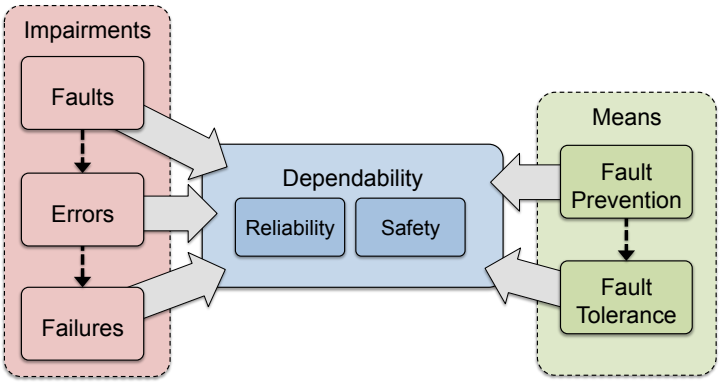


Figure 2.12: Main attributes of a dependable system with potential impairments and means.

2.3.2 Fault Types

Different fault types can be distinguished by means of their occurrence and their duration in a system [BW09]:

Transient faults: This fault occurs at a particular point in time, remains in the system for some unspecific time period and disappears. For instance, hardware components which have an adverse reaction to radioactivity may have transient faults.

Intermittent faults: These faults are a special kind of transient faults that occur from time to time. For example, this kind of fault can be observed at hardware components that are heat sensitive. They work correctly, stop working when overheating, cool down for a while and start working again. This scenario can repeat over and over again.

Permanent faults: A fault is called permanent fault, if it stays in the system even until a shutdown. Even after a restart this fault remains in the system, if it does not get repaired. Often, these are hardware faults, like broken wires or defect sensors.

To create a dependable system all these types of faults must be prevented from causing erroneous system behavior, i.e. failures, and conditions leading to mishaps possibly causing damages, accidents, injuries, or even death. Faults can also be distinguished by the phase in which they are introduced [LAK92]. The faults mentioned above are *operational faults* which occur during system lifetime and are caused by physical impact; *design faults* are introduced already at system design or during system modification. Design faults are much harder to detect and tolerate and require different means than operational faults [Jal94]. Moreover, operational faults may occur even if design faults are prevented. In the next section we describe different means for dependable systems and clarify why we focus on operational faults in this thesis.

2.3.3 Means for Dependable Systems

Figure 2.12 also provides means to increase the dependability of a system by a decreased level of susceptibility to faults. There are two major means to protect a system against the impairments described above and improve reliability and safety [LA90]. On the one hand *fault prevention* shall eliminate any possibility of faults in a system design before it goes operational; on the other hand *fault tolerance* shall enable a system to continue with a reliable and safe operation even in the presence of a fault.

Fault Prevention There are two stages to realize fault prevention: *fault avoidance* and *fault removal* [BW09]. Fault avoidance shall avoid or at least limit the introduction of potentially faulty components during system design. For hardware components this can be realized by [RLT78]:

- using the most reliable components within the given cost and performance constraints,
- applying thoroughly-refined techniques for interconnection of components and assembly of subsystems, and
- protecting the hardware from expected forms of interference.

In software components faults can be avoided or at least limited by rigorous or even formal requirement specifications and the use of proven design

methodologies. In spite of these concepts for avoidance, design faults will very likely be present in the hardware and software components after their construction. Therefore, fault removal can be applied as second stage of fault prevention. Although this includes concepts for finding and removing the causes of errors, like design reviews and program verification, usually system testing is performed. However, testing has some major limitations and will very likely not remove all potential faults. A test can only prove the presence of faults, not their absence. Furthermore, it is sometimes impossible to test under realistic conditions; most tests are simulations and it is difficult to guarantee that these are accurate. The purpose of verification is a consistency check of an implementation model against its specifications. Verification can be performed multiple times during design and development of a system until the required quality is achieved. A common technique is formal verification which proves the consistency between model aspects and formal specifications analytically [Kru09]. Formal verification approaches are for instance symbolic model checking and equivalence checking. Symbolic model checking verifies the validity of a property specification expressed by a temporal logic formula with respect to a synchronous finite state machine model [CGP99]. Equivalence checking proves the behavioral equivalence of two design models [KSL95], [Kue+02]. Nevertheless, despite testing and verification techniques hardware components may still fail caused by operational faults. Because of this fact and the described limitations of fault prevention, system designers should consider fault tolerance [BW09].

Fault Tolerance Fault tolerance is the ability of a system to continue performing its tasks after the occurrence of a fault. *Fault masking* can be applied which prevents faults within a system from introducing errors resulting in a failure [Pra96]. In other words, the goal of fault tolerance is the avoidance of a system failure even if a fault is present. In this context, it is important to note that the overall system cannot be made fault tolerant against its own failures, but against the failure of its components. Thus, fault tolerance masks the failure of a subsystem to prevent a failure of the whole system [Jal94]. Another efficient technique for fault tolerance is *reconfiguration*. Reconfiguration detects and locates a fault and reconfigures the system to remove or rather replace a faulty component. This technique consists of four steps: *Fault detection*, *fault location*, *fault containment* and *fault recovery*. As a first step, fault detection recognizes the occurrence of a fault. This is the necessary to perform the next steps. Once a fault is detected, fault location determines where the fault arises to identify an appropriate recovery strategy. Fault containment is required in all fault-tolerant designs, because it isolates the fault to avoid its propagation possibly resulting in a failure of the overall system. As the final step, fault recovery ensures that even in the presence of a fault, the system remains operational or regains operational status by means of reconfiguration [Pra96].

Generally, three different levels of fault tolerance can be realized for a system [BW09]:

- **Fail Safe:** The system maintains its integrity while accepting a temporary stopping of its operation.
- **Fail Soft:** The system continues to operate in the presence of errors and accepts a partial degradation of functionality or performance during the time needed for recovery or repair.
- **Full Fault Tolerance:** The system continues to operate in the presence of faults – at least for a limited time period – with no significant loss of functionality or performance.

The required level of fault tolerance of a system depends on the application. In practice fail soft is sufficient for some distributed systems, however most safety-critical systems require full fault tolerance [BW09]. Since we focus on safety-critical real-time systems, our approach realizes full fault tolerance by means of reconfiguration.

Redundancy All fault-tolerant techniques require additional elements integrated into the system to detect and recover from faults [BW09]. Since these components are not required for normal system operation they are called *redundant* components. Based on this so-called *protective redundancy* in [BW09] Burns and Wellings define the aim of fault tolerance as:

> "*The aim of fault tolerance is to minimize redundancy while maximizing the reliability provided, subject to the cost and size constraints of the system.*"

Hardware redundancy is perhaps the most common form of redundancy applied in systems [Pra96]. It can be distinguished between *static* (passive) and *dynamic* (active) redundancy [LA90]. Static redundancy utilizes voting mechanisms to mask fault occurrences. A widely spread form of static hardware redundancy is *Triple Modular Redundancy* (TMR). TMR consist of triplicated hardware components, e.g. a processor, and a majority voting component to determine the correct output and masking the fault of a single processor. To mask faults from more than one component more redundancy is necessary. Here, *N Modular Redundancy* (NMR) can be applied. Obviously, the voting mechanism is the primary disadvantage with TMR and NMR because it is a SPOF. To overcome possible voter failures this component can also be multiplied, e.g. triplicated. Nevertheless, particularly for a distributed system with many ECUs these redundancy concepts result in a significant rise of complexity, size, and cost. Dynamic redundancy applies

reconfiguration to realize fault tolerance. It does not realize a fault masking, i.e. it does not prevent faults from producing errors within a system. These errors and the corresponding faults can be detected. Clearly, if a fault and its produced errors remain undetected this will probably result in a system failure. Thus, the error must be detected to locate its source, i.e. the faulty component. The faulty component has to be removed and its functionality has to be resumed by another to regain the operational status of the system. This clarifies the importance of appropriate fault detection and location capabilities for dynamic redundancy. Dynamic redundancy is best applied for applications where temporary faults and errors are acceptable as long as the system regains its operational status in a sufficiently short period of time. However, it is usually preferable to apply dynamic redundancy than providing the large quantities of additional hardware to achieve static redundancy [Pra96].

2.4 Summary

This chapter presented fundamentals and basic terms required for the design of fault-tolerant systems. The fundamentals of real-time systems in general contain the characteristics of tasks and their scheduling. Based on these basic terms, the fundamentals of distributed real-time systems presented additional properties and constraints for communication and timing issues. Finally, the basic terminology and major concepts for fault-tolerant system design were introduced.

Chapter 3

Related Work

In Chapter 1 we motivated the work presented in this thesis by describing the increasing importance and complexity of distributed real-time systems. Due to the growing importance of distributed real-time systems, there exists a lot of related work in this context. Therefore, this chapter provides an overview of the most relevant publications related to the work presented in this thesis. It starts with existing work dealing with the design and configuration of distributed real-time systems. Afterwards, it presents a broad overview of publications regarding the scheduling of real-time tasks and messages in distributed systems. Since this thesis presents a design approach for fault-tolerant distributed real-time systems, this chapter concludes with the presentation of existing approaches for fault tolerance in distributed real-time systems.

3.1 Design of Distributed Real-Time Systems

As mentioned before, the design and configuration of distributed real-time systems is a large research area. Two popular and often cited books of reference in this domain are [BW09] and [Kop11]. Burns and Wellings [BW09] provide an in-depth analysis of the requirements for the design and implementation of embedded real-time systems, while Kopetz [Kop11] is considered as a reference book that explains the fundamental concepts of this field.

As described in Chapter 1, two of the major domains of distributed real-time systems are the avionic and the automotive domain. Hence, several papers and articles addressing the design and configuration of such systems were published in the context of these domains. Hammett [Ham03] motivates the necessity to reexamine the way avionics systems are built because the trend to very highly integrated systems is leading to systems that are too complex to be trusted. Therefore, he proposes that aerospace developers shall leverage

the research investments of the automotive industry in distributed systems to be able to build distributed avionics systems that provide increased functionality at a manageable level of complexity and to avoid SPOFs. Navet et al. present the current state of the art, trends, and open issues for the design of automotive distributed systems with focus on the communication in [NNa+05]. The authors mention that in terms of criticality future automotive X-by-wire systems can reasonably be compared with flight-by-wire systems in the avionic field and that according to [Tea98], the probability of encountering a critical safety failure in vehicles must not exceed $5 \cdot 10^{-10}$, but other studies consider 10^{-9}. Hence, they identify two main challenges: (i) to reach such dependability with respect to cost constraints, and (ii) to develop design methodologies adapted to the specific automotive constraints.

Scheickl and Rudorfer [SR08] emphasize the importance of model-based approaches with consideration of timing constraints for the design of automotive real-time systems. Therefore, they propose a timing-augmented system model containing different types of information about the real-time system. The authors identify an appropriate mapping and scheduling of tasks and messages in a distributed system as a main challenge regarding timing. Thus, in the next section we give a broad overview of related work on this research field. The work presented in this thesis addresses all the design issues mentioned above. We present a design approach for fault-tolerant distributed real-time systems which provide the intended functionality, avoid a SPOF, and offer cost-efficient dependability. This approach utilizes timing-augmented system models and is also applied to a standardized design methodology respecting the specific constraints of the automotive domain. An appropriate deployment with a holistic fault-tolerant mapping and scheduling of tasks and messages that fulfills the given timing constraints is an essential part of the design approach.

As described in Section 2.2.2, time-triggered communication is deterministic and the time-triggered FlexRay protocol that is considered in this thesis is the emerging standard for real-time communication in safety-critical automotive systems. Jang et al. [Jan+11] mention that designing a FlexRay network is a complex and difficult task because the FlexRay protocol has more than 70 configuration parameters which correlate with each other. However, they identify two main parameters that are highly relevant to the application software: (i) the static slot length and (ii) the communication cycle length. Park and Sunwoo describe a parameter optimization method for these parameters in [PS11] which was integrated in a design framework by Jang et al. in [Jan+11]. The framework is predicated on principles for optimizing the network utilization and the worst case response time (WCRT) of transmitted frames. The method consists of two steps which determine the optimal static slot length and the optimal communication cycle length. These steps are supported by a frame packing and a scheduling algorithm. In the first step of the

proposed framework, the optimal static slot length is determined by means of the network utilization and frame packing results. Based on the results from the first step, the second step determines the optimal communication cycle length based on the scheduling results and the WCRT analysis of the frames. The authors mention that they do not consider multirate communication and in-cycle repetition that allows the scheduling of messages with a lower period than the duration of one communication cycle. In contrast, we support multirate communication and multiple instances of a message in one communication cycle as well as cycle multiplexing to enable a flexible configuration of these main FlexRay parameters for our design approach (cf. Section 4.2.5).

Hagiescu et al. [Hag+07] propose a compositional framework to model and analyze ECUs interconnected by a FlexRay bus. The authors provide a method to analyze the performance of the FlexRay bus by means of the maximum end-to-end delay, required amount of buffer at each commincation controller, and utilization of ECUs and bus. However, they focus on the non-deterministic dynamic segment in the performance analysis which is not applicable for safety-critical distributed systems we are considering here. Pop et al. [Pop+06] present a schedulability analysis for the FlexRay communication protocol and propose a network parameter design process. Timing properties of static messages have been established by building a static cyclic scheduling and the FlexRay message analysis has been integrated in the context of a holistic schedulability analysis that determines the timing properties for all the tasks and messages in the system. This analysis is also the basis for a follow-up publication proposing an optimized scheduling of FlexRay messages that we present in the following section.

3.2 Scheduling in Distributed Real-Time Systems

Numerous authors like Ramamritham and Stankovic in [RS94] argue that an appropriate scheduling of the performed activities to meet their timing constraints is one major problem in real-time computing systems. Hence, there exists numerous literature addressing this topic. The textbook [But11] from Buttazzo is considered as reference literature that introduces the basic concepts of real-time computing and means to design real-time systems. This includes a detailed description of the commonly applied scheduling strategies for periodic real-time tasks which we also consider in this thesis.

In a distributed real-time system tasks and messages have to be scheduled to ensure their timeliness. Cottet et al. mention in [Cot+02] that task scheduling in distributed systems is dealt with at two levels: (i) on the local level of each processor and (ii) on the global level of mapping tasks to processors. Davis and Burns [DB11] provide a survey on hard real-time scheduling

algorithms and schedulability analysis techniques for homogeneous multiprocessor systems.[11] The paper also reviews the key results in this research field. As described in Section 2.2.1, global scheduling may imply dynamic allocation by means of migration. Therefore, the article provides a classification of scheduling algorithms from [Car+04] according to their allocation respectively migration properties. The authors call scheduling algorithms where no migration is permitted *partitioned* and those where migration is permitted *global*. Beside the additional communication loads required for task respectively job migrations, the authors mention that from a practical perspective, the main advantage of using a partitioning approach is that, once an allocation of tasks to processors has been achieved, a wealth of real-time scheduling techniques and analyses for uniprocessor systems can be applied.

The authors present the properties of several partitioning approaches which are based on local scheduling with EDF from [Hor74] and RM from [LL73] (cf. Section 2.1.2). In works like [DL78], [DD86], [OS95], and [Lie+95], these strategies are examined combined with bin packing heuristics such as first fit, next fit, best fit, and worst fit, as well as task orderings such as decreasing utilization for task allocation. The resulting partitioned algorithms have been further analyzed regarding their utilization bounds for schedulability analysis in publications like [ABJ01], [AJ03], and [LDG04]. Furthermore, in their publication [BF07] Baruah and Fisher present an EDF-based algorithm based on ordering tasks by increasing relative deadline which utilizes a sufficient test based on a linear upper bound for the processor demand criterion to determine schedulability. Fisher et al. [FBB06] apply a similar approach for partitioning with fixed task priority scheduling based on DM. The resulting algorithm is based on ordering tasks by decreasing relative deadline and uses a sufficient test based on a linear upper bound for a defined processor request bound function to determine schedulability. However, as mentioned above the presented strategies consider homogeneous processors and communication delays between distributed processors are not explicitly considered in the presented algorithms and schedulability tests. In contrast, in Section 4.2.4 we extend the existing schedulability tests for the considered local scheduling algorithms EDF and RM/DM to support heterogeneous systems to a certain extend and even more important explicitly integrate communication delays.

The main disadvantage of the partitioning approach is that the task allocation problem is analogous to the bin packing problem and known to be NP-Hard [GJ79]. Moreover, Anderson and Jonsson [AJ00] describe that there are typically fewer context switches respectively preemptions when global scheduling is applied because the scheduler will only preempt a task when there are

[11] The term multiprocessor systems encapsulates multicore systems as well as distributed systems. In the context of scheduling, these systems differ mainly in their underlying communication infrastructure with their resulting delays.

no processors idle. Hence, Davis and Burns [DB11] also give an overview about the key research results in global scheduling with dynamic allocation and task migration. They describe that Dhall and Liu already in [DL78] considered global scheduling of periodic task sets with implicit deadlines on multiprocessor systems and analyzed the utilization bound of global EDF. There exist several publications like [SB02], [GFB03], and [Bak05] which also consider task sets with implicit deadlines and present different variants of global EDF scheduling with their corresponding utilization bounds. Also for task sets with constraint and arbitrary deadlines different variants of global EDF have been proposed and analyzed, e.g. in [BCL05], [BMS08], and [BB09].

However, in general EDF is known to not be optimal for global scheduling on multiprocessor platforms [EDB10]. Therefore, Davis and Burns also provide an overview of existing global dynamic priority scheduling algorithms which are optimal for periodic task sets with implicit deadlines [DB11]. For instance, Baruah et al. [Bar+96] introduce the Proportionate Fair (Pfair) algorithm. Pfair scheduling divides the execution timeline into equal length quanta and allocates tasks to processors at each time quantum. The authors show that Pfair is optimal for periodic task sets with implicit deadlines and has a utilization bound of m for m processors. However, in practical use Pfair incurs very high overheads by making scheduling decisions at each time quantum [DB11]. Numerous variants of Pfair with improved efficiency have been introduced in publications like [AS00] and [AS01]. The Boundary Fair (BF) algorithm presented by Zhu et al. in [ZMM03] is similar to Pfair but it only makes scheduling decisions at period boundaries which reduces the number of scheduling points to typically at most 50% of the number required for Pfair. The authors prove that BF is also an optimal algorithm for periodic task sets with implicit deadlines. Beside these articles, there are numerous further publications regarding task scheduling on multiprocessor systems, but their extensive discussion is beyond the scope of this thesis. Hence, for further details, we refer to [DB11].

Even though the global dynamic priority scheduling algorithms mentioned above are optimal for periodic task sets with implicit deadlines, there exist no optimal online algorithms for the preemptive scheduling of sporadic task sets on multiprocessors. Moreover, their practical use can be problematic due to the potentially excessive overheads caused by frequent preemption and migration [DB11]. This means, that dynamic allocation can lead to nondeterminism or missing of timing constraints if the allocation is performed at runtime. For instance, the global scheduling algorithm finds no feasible solution because of its additionally required computation time. However, global scheduling is flexible regarding possible changes in the system topology. This implies that dynamic allocation can increase the fault tolerance of a distributed system in case of a node failure [Cot+02]. Therefore, our ap-

proach combines the advantages of both allocation concepts. On the one hand it keeps the determinism of the static allocation and guarantees the timing constraints and on the other hand, it increases the fault tolerance by including possible reallocations in the predetermined solution to compensate node failures. This means, like in the partitioning approaches presented above, we determine a static allocation for each configuration and based on that consider EDF and RM/DM for the resulting local scheduling. Thus, according to [Cot+02], in the following we use the term global scheduling synonymous for the mapping of tasks to processing nodes. Furthermore, it is important to remind that the presented allocation and scheduling strategies consider homogeneous processors and do not consider communication delays, whereas our approach supports heterogeneous systems to a certain extend and integrates resulting communication delays.

In distributed real-time systems, besides tasks also messages have to be scheduled to ensure their timeliness. Depending on the bus mapping and the message scheduling, this results in a communication delay of a message respectively its receiving task which should be small and deterministic for real-time communication. As described above, we consider deterministic time-triggered communication in this thesis. Thus, for the related work in this field we focus on publications regarding time-triggered bus communication and in particular FlexRay and its static segment. As described in Section 2.2.3 FlexRay has been introduced in 1999. At the same time Pop et al. [PEP99] already presented an approach for scheduling in time-triggered embedded systems based on TTP communication and applied it to an automotive case study. This work is proceeded in [PEP00] where the authors develop different message scheduling policies and compare them by means of an aircraft control system. Pop et al. [Pop+07] also propose first techniques to optimize the FlexRay bus access, based on their analyses from [Pop+06], so that the hard real-time deadlines are met for all tasks and messages in the system. For this purpose the authors propose different heuristics and algorithms to optimize the bus access and compared the results to reference values produced by a simulated annealing based design space exploration. To further evaluate the proposed techniques, they also consider a real-life example implementing a vehicle cruise controller.

During the last years, several other papers regarding scheduling and optimization in the static segment of FlexRay have been published. Amrion and Ektebasi [AE12] provide a survey on this topic. The authors present different approaches categorized by their scheduling techniques. One technique is bin packing which is used with respect to the static segment of FlexRay to minimize the number of slots that are allocated to ECUs in order to maximize the utilization of the bus. Lukasiewycz et al. present a two dimensional bin packing technique for scheduling of the static segment in [Luk+09]. Moreover, the authors introduce a fast greedy heuristic algorithm as well as an

integer linear programming approach for optimization of this bin packing. Another applied technique are genetic algorithms. Ding et al. have initially proposed their scheduling method based on genetic algorithms in [Din+05] and proceed their work in [DTT08]. Besides the primary objective of meeting all task deadlines, the secondary objective is defined that the schedule can optimize communication buses to minimize hardware cost. The authors consider different modeling paradigms and constraints we also consider in this thesis: The software architecture is modeled as a TDG, each inter-node message must be scheduled separately on the bus, and the periods of the tasks in the TDG imply the end-to-end delay of the system (cf. Chapter 4). Ding [Din10] extended this work and presents a hybrid algorithm that combines a worst-fit bin packing approach with a genetic algorithm to achieve a minimum number of used slots. The performed experiments by means of a automotive safety system show that the hybrid algorithm is superior to the individual algorithms.

Above, we already mentioned linear programming as an applied optimization strategy. Schmidt and Schmidt also apply linear programming techniques in [SS09] to solve two identified subproblems for scheduling in the static segment: (i) Data signals have to be packed into equal-size messages while using as little bandwidth as possible and (ii) a message schedule has to be determined such that the periodic messages are transmitted with minimum jitter, i.e. a low deviation from the periodicity. The presented frame packing and scheduling approaches are applied to a benchmark signal set to illustrate and analyze the results. Schmidt and Schmidt [SS10] proceed their work. Similar to previous work, the authors seek to allocate a minimum number of frame IDs respectively slots in the static segment. But in addition they want to minimize the message jitter. For this purpose, they formulate a linear integer programming problem whose solution is the desired message schedule and apply it to an example adopted from an automotive application. Zeng et al. [Zen+11] provide solutions for a task and signal scheduling problem based on a mixed-integer linear programming optimization framework. Like in this thesis, the authors study the problem of the ECU and FlexRay bus scheduling from the perspective of the designer, but interested in optimizing the scheduling subject to timing constraints with respect to latency and extensibility of the system. For evaluation the presented method has been applied to an automotive X-by-wire system as a case study.

As described in Section 2.2.3, the FlexRay protocol has several parameters and constraints. Sun et al. [Sun+10] present a constraint programming approach for optimization of the objective of minimizing the used static slots through task and message scheduling. Therefore, the authors identify and define six important configuration parameters for the configuration constraints: transmission rate, cycle length, number of channels, number of static slots, slot length, and maximum message size. All of these parameters are also con-

sidered in this thesis and configured according to the given system properties and constraints (cf. Section 4.2.5 and 4.3.2). To model system properties, in [Sun+10] the authors reuse the modeling from [Zen+09] which consists of a task graph and a linear bus topology. These models are similar to the modeling of system properties in this thesis by means of task dependency graphs and hardware architecture graphs which we present in Section 4.3.1. However, we define additional graph-based models to represent available slots, performed task mappings, and the whole design result. As case study, the authors consider modified software models of real-world automotive control applications from [KHM03].

All publications above define similar objectives for the presented approaches. Beside the determination of a feasible scheduling which guarantees the timeliness of the messages and meets the task deadlines, the authors want to minimize the number of used slots. This reduction of required slots assigned to an ECU is even more important for our fault-tolerant design approach. Especially in huge distributed systems with many ECUs communicating over the bus, an efficient and flexible bus mapping is required to reduce the resulting communication delays and determine a feasible schedule. Therefore, our approach analyzes the inter-ECU communication of the initial configuration and all reconfigurations to perform a combined bus mapping that utilizes message respectively slot reuse and frame packing for inter-ECU messages with the same sender ECU (cf. Section 4.4.2). Moreover, for evaluation nearly all presented approaches were applied to case studies which are real-world examples form the automotive or avionic domain. As mentioned above, in Chapter 5 we also apply our approach to an automotive case study to approve its feasibility for realistic systems.

3.3 Fault Tolerance in Distributed Real-Time Systems

In Section 2.3 we mentioned that for safety-critical systems, the main attributes for dependability are reliability and safety. Fault tolerance is a common mean to increase reliability and safety in a computer system [LA90]. Several textbooks like [BW09], [Jal94], [Pra96], and [Kop11] describe basic concepts and properties of fault tolerance.

In the previous section we mentioned numerous publications regarding the scheduling of real-time tasks and messages in multiprocessor and distributed systems. Additionally, there also exist several works addressing dependable, reliable, and fault-tolerant task scheduling. Oh and Son [OS97] study the problem of scheduling a set of real-time tasks to meet their deadlines even in the presence of processor failures. The authors prove that the problem of scheduling a set of non-preemptive tasks on more than two processors to tolerate one arbitrary processor failure is NP-complete even when the tasks

share a common deadline. They propose a heuristic algorithm to determine a schedule that can tolerate one arbitrary processor failure in the worst case. But the authors assume that all tasks share a common deadline which is an unrealistic condition for a distributed system. Lauzac et al. present a task scheduling algorithm based on Pfair that tolerates transient and permanent faults in [LM98]. The authors show that this fault-tolerant algorithm yields a processor utilization close to the optimal when the task set consists of many tasks with small utilization. However, they also define strict conditions for the task set. For instance, a task τ_i must be finished at least C_i time units before its deadline d_i to ensure that there is enough time left for the re-execution of the faulty task. A modified Pfair-based approach for fault-tolerant global scheduling is presented by Liberato et al. in [Lib+99]. The authors formulate similar conditions but introduce a recovery time V_i at which the task τ_i has to be finished before d_i to ensure that there is enough time left for the recovery of the faulty task. These conditions are too much restrictive and not applicable for the design of cost-efficient distributed systems addressed in this thesis.

Further fault-tolerant scheduling algorithms for distributed systems are presented by Srinivasan and Jha in [SJ99] and by Qin et al. in [Q+99]. However, the presented approaches are all based on homogeneous systems and consider independent tasks. Therefore, Qin et al. also present a fault-tolerant algorithm for heterogeneous distributed systems in [Q+00]. Qin et al. [QJS02] extend their work and present an algorithm that addresses the issues of real-time, fault-tolerance, reliability, heterogeneity, and precedence constraints. Extensive simulations indicate that the presented algorithm is considerably superior to the most relevant algorithms in the vast majority of cases. Even though this algorithm considers precedence constraints and communication delays, like the other presented approaches it does not address remote inter-node message scheduling.

There also exist some publications addressing dependable and fault tolerant communication and message scheduling. Kandasamy et al. [KHM03] address the design of low-cost communication networks to meet performance, timing, and fault tolerance requirements for distributed real-time systems. The authors propose a TDMA-based multiple-bus network topology to enable fault-tolerant message allocation. Redundant routes are provided for messages with specific fault-tolerance requirements, i.e. for a k-fault-tolerant message m, k replicas or copies are allocated to separate buses. The approach is applied to a software model case study involving some advanced automotive control applications and it has been shown that sharing transmission slots among multiple messages, i.e. frame packing, reduces bandwidth consumption while preserving predictable communication. Here, we also consider the redundant communication channel of the FlexRay bus for fault-tolerant messages and apply frame packing to reduce unnecessary slot assignments. Moreover, as mentioned above we also apply our design approach to soft-

ware models from [KHM03] in Section 5.2.1. However, in contrast to our approach, the authors do not address the fault-tolerant allocation of tasks to processors.

Brendle et al. present fault-tolerance strategies for implementing passive replication techniques in networked embedded systems based on FlexRay busses in [Bre+08]. Among other things, the authors propose to replicate not only the processes but also the messages and to reserve the required bandwidth a priori at design time. Since a dynamic reconfiguration of the static segment at runtime is not permitted in FlexRay, we also consider redundant slot assignments in our approach. Tanasa et al. [Tan+10] present a framework for generating fault-tolerant message schedules on the static segment. The generated fault-tolerant schedules achieve the reliability goal even in the presence of transient and intermittent faults. Moreover, their technique minimizes the required number of message retransmissions in reserved slots in order to achieve fault-tolerant schedules to optimize the bandwidth utilization. Therefore, they formulate the optimization problem in Constraint Logic Programming (CLP), which returns optimal results and propose an efficient heuristic. Experiments on synthetic test cases and real-world brake-by-wire and ACC case studies show that the heuristic scales significantly better than the CLP formulation. In contrast to our approach, the authors only utilize one channel of FlexRay and not the optional second one for additional fault tolerance to avoid a SPOF.

Tanasa et al. [Tan+11] extend their work by a novel frame packing method. The authors mention that the technique proposed in [Tan+10] cannot be directly applied at a system design from scratch, i.e. when signals are the only known units of communication. Hence, the proposed approach handles frame packing and frame scheduling for reliable transmission over FlexRay. Their method computes the required number of retransmissions of frames that ensures the specified reliability goal. The proposed frame packing method also guarantees the timeliness of all signals and meets the desired reliability goal to ensure fault-tolerance at the minimum bandwidth cost. As mentioned above, our approach also applies timely frame packing to avoid unnecessary bandwidth utilization. Hau et al. propose a novel holistic scheduling scheme in [HLH12] that can provide scalable fault tolerance by using flexible and dual channel communication in FlexRay. This scheme is built upon a novel slot pilfering technique to schedule and optimize the available slots. Hence, it offers two features: (i) Providing fault-tolerance and (ii) improving bandwidth utilization. Extensive experimental results based on synthetic examples and a real-world brake-by-wire test case show the efficiency and efficacy of the proposed scheme. However, in contrast to our approach all of these publications do not address the combined fault-tolerant allocation and scheduling of tasks and messages.

Izosimov et al. published several papers dealing with fault tolerance in distributed systems. In [Izo+05] they present an approach for the design optimization of fault-tolerant embedded systems for safety-critical applications given a limited amount of resources. Their approach decides the mapping of processes to processors and the assignment of fault-tolerant policies to processes, i.e. process re-execution and replication, such that transient faults are tolerated and the timing constraints of the application are satisfied. Several experiments, including a real-world ACC show that the presented approach is able to provide fault tolerance under limited resources. In [Izo+06a], the authors extend their previous work to handle checkpointing, which provides time-redundancy, simultaneously with replication, which provides space-redundancy. They also propose a novel scheduling approach for fault-tolerant embedded systems considering process re-execution for tolerating multiple transient faults in [Izo+06b]. The authors mention that the main contribution is the ability to handle performance versus transparency and memory trade-offs imposed by the designer. Transparency means that process recovery is performed such that the operation of other processes is not affected.

Izosimov et al. [Izo+08] present an approach to the synthesis of fault-tolerant schedules for embedded applications with soft and hard real-time constraints. The approach is intended to guarantee the deadlines for hard real-time processes even in the case of faults, while maximizing the overall utilization. Process re-execution is employed to recover from multiple faults. A single static schedule predetermined offline is not fault-tolerant and is pessimistic in terms of utilization, while a purely online approach, which computes a new schedule every time a process fails or completes, incurs an unacceptable overhead and may result in non-determinism. Thus, the authors propose a scheduling strategy that synthesizes fault-tolerant schedules off-line which are selected by the scheduler based on the occurrence of faults and the actual execution times of processes at runtime. This technique is similar to our fault-tolerant design approach where we predetermine different global scheduling reconfigurations, i.e. task mappings, which are activated at runtime after occurrence of a processor failure to enable fault-tolerant and timely local schedulings. However, the publications mentioned above do not address fault-tolerant scheduling of messages and an optimization of the communication channel. Moreover, they do not cover permanent hardware faults since a faulty task is re-executed on the same ECU whereas our approach handles both of these aspects.

Kim et al. [KLR10] address the problem of guaranteeing reliability requirements on fail-stop processors, i.e. also permanent faults, in fault-tolerant multiprocessor real-time systems. Therefore, they propose a task-partitioning strategy and a task allocation algorithm that allocates so-called Hot Standby and Cold Standby task replicas to meet system-level reliability requirements. Kim et al. extend their work in [Kim+11] by integrating a novel processor as-

signment approach into the standardized methodology for the development of automotive systems to guarantee the end-to-end delay of the application flow and the given reliability requirements. For our fault-tolerant deployment, we also consider Cold Standby replicas which consume processing time only when activated (cf. Section 4.2). Furthermore, we also apply our design approach to the design of automotive systems in Chapter 5. In contrast to the publications above, we also provide means for a fault-tolerant and efficient message allocation in addition to the fault-tolerant deployment of real-time tasks.

3.4 Summary

This chapter gave an overview of related work relevant in the context of this thesis. It started with publications dealing with the design and configuration of distributed real-time systems. Afterwards, it gave a broad overview of literature addressing the scheduling of tasks and messages in distributed real-time systems. Finally, this chapter presented existing concepts and approaches for dependable and fault-tolerant distributed real-time systems.

Chapter 4

Design Approach for Fault-Tolerant Distributed Real-Time Systems

This chapter describes our design approach for fault-tolerant distributed real-time systems. The concepts and ideas described in this chapter are based on work we presented in [KKM11], [Klo+11], and [KMR12]. It starts with a short overview of the design approach and presents general objectives and the properties of our concept and the definition of important constraints. The main objective is the determination of feasible and flexible fault-tolerant system designs with compensation of different usual fault types and a simultaneous reduction of required redundancy. Therefore, we present concepts and preconditions to enable fault tolerance by means of reconfiguration in a distributed system. This includes the proposal of a reconfigurable distributed system topology with an additional concept for fault tolerance by means of reconfiguration and dynamic redundancy. Afterwards, we describe concepts and techniques to specify and configure properties and constraints for the software architecture and the hardware topology to enable a feasible, efficient, and flexible system design. Based on these results, we present our graph-based modeling of system properties as input for our design approach. Finally, this chapter describes the performed system deployment and its design result as essential part of the approach.

4.1 Overview

Figure 4.1 depicts an overview of the design approach for fault-tolerant distributed real-time systems. It illustrates the extension and refinement of the design flow steps for distributed systems shown in Figure 1.2 and the resulting dependencies.

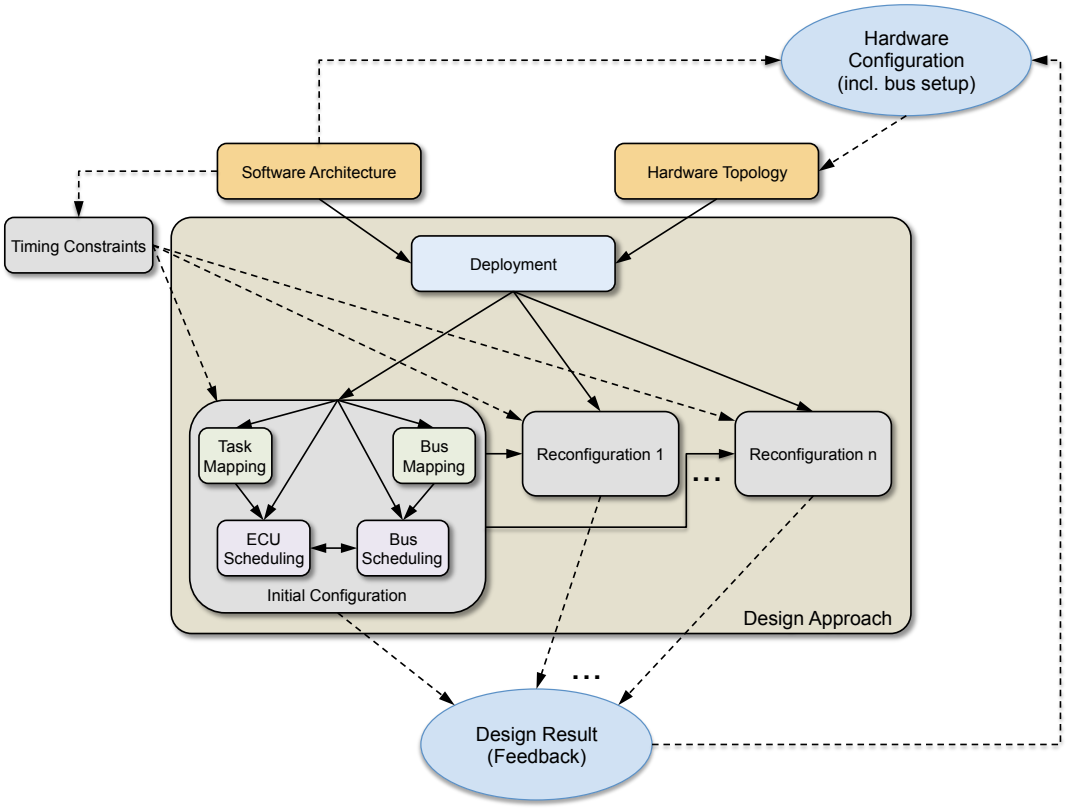


Figure 4.1: Overview of the design approach for fault-tolerant distributed real-time systems.

The inputs for the design of the system are graph-based models of the given software architecture and hardware topology. The software architecture is derived from the intended function or set of functions controlled by the embedded system. As described in Section 4.2.5 the given software architecture implies initial constraints and requirements for the hardware topology configuration including the setup of basic bus parameters. For this thesis we consider the FlexRay protocol as example but our approach is also applicable to other TDMA-based protocols (cf. Section 2.2.2). Moreover, corresponding timing constraints for the executed functions have to be specified to guarantee a correct system behavior of each configuration with respect to the given hard real-time constraints (cf. Section 4.2.3).

Based on the input models our design approach performs the system deployment. This comprises the determination of task and bus mapping as well as corresponding schedulings for the initial configuration and based on that for all necessary reconfigurations to design a fault-tolerant system. During the deployment the fulfillment of the timing constraints is analyzed to guarantee the correct behavior of the distributed real-time system even in case of an ECU failure.

As final design result the approach returns the determined system deployment. Ideally, this contains feasible task and bus mappings for the initial configuration and all required reconfigurations. Even if not for all configurations a feasible solution can be found, this feedback for the system designer can also be used as information for modifications on the input model, i.e. mainly the hardware configuration. Based on an adjusted hardware topology model as input, e.g. with additional or more powerful ECUs or with a changed communication bus setup, a further iteration of the design approach can be performed.

4.2 Objectives and Properties

The following sections provide detailed descriptions of the objectives, properties, and important constraints for our design approach. The main objective is the design of fault-tolerant distributed real-time systems. Since we focus on safety-critical real-time systems, fault tolerance has to be realized by means of reconfiguration, i.e. the system can continue its operation in presence of a fault – at least for a limited time period – with no significant loss of functionality (cf. Section 2.3.3).

As mentioned in Section 2.3.2, our approach is supposed to increase fault tolerance by compensating different types of operational hardware faults. These faults can be distinguished by means of their occurrence and their duration in a system as permanent faults, intermittent faults, and transient faults which may result in different behaviors of the faulty component. Obviously, in a distributed system these faults may result in two types of component failures: network failures and processor respectively node failures [CJ97]. To increase fault tolerance for the network components the FlexRay protocol already offers dual-channel redundancy and bus guardians (cf. Section 2.2.3). Nevertheless, the failure of a single node may result in a malfunction of the whole distributed system. Thus, to further increase fault tolerance possible ECU faults must also be compensated [KMR12].

Any of the fault types mentioned above may result in either a *fail-silent* respectively *fail-stop* failure or in a *Byzantine* failure of the ECU [CJ97]. On the one hand, we consider the concept of the fail-silent failure model where a system produces correct output and fulfills timing constraints until it fails [BW09]. An extension of this is the fail-stop failure model [Pra96] where a failed component is assumed to stop generating any data and a working component can resume control by detecting the lack of output from the failed component. The component that takes over then aims to meet the desired deadline of the failed component [Kim+11]. Our approach supports fail-stop failures because it allows other components to detect the fail-silent state of a failed component [BW09]. For instance, permanent hardware faults like

burnt out chips or processor faults may stop data output and result in a fail-stop failures. On the other hand we also consider Byzantine failures where the faulty component continues to run but generates incorrect output. If hardware components have an adverse reaction to electromagnetic interference or radioactivity they may be vulnerable to Byzantine failures. For instance, in [Dri+03] the authors reference an experiment from [Siv+03] where a time-triggered communication controller has been radiated with heavy ions. The reported fault manifestations were bit-flips in registers and RAM locations which can result in erroneous transmissions. In this case our approach must detect and ignore the incorrect output and a correctly working component resumes control. Section 4.2.2 describes the concepts we utilize for our approach to detect and compensate faults within the distributed system based on our proposal for a *reconfigurable distributed system topology* which we introduce in Section 4.2.1.

Failure mode	**Failure rate (FIT)**
Permanent hardware failures	10–100
Non-fail-silent permanent hardware failures	1–10
Transient hardware failures (strong dependence on environment)	1000–1000000

Table 4.1: Order of magnitude of hardware failure rates [PMH98].

Table 4.1 lists orders of magnitude of typical hardware *failure rates* of large VLSI chips that are used in the industrial and automotive domain [Kop11]. The numbers in the table are originating from [PMH98]. It provides the failure rates of different hardware failure modes in the unit *Failures In Time* (FIT) where 1 FIT represents one failure per 1 billion (10^9) device-hours [MC89]. Table 4.1 shows that the failure rate for permanent failures of an industrial or automotive chip is in a range between 1 and 100 FITs, whereby the non-fail-silent failure rate is only up to 10 FITs. The failure rate for transient failures is orders of magnitude higher with a strong dependence on the surrounding environment, i.e. 1,000 up to 1,000,000 FITs. Considering the fact that current luxury cars contain up to 100 ECUs, based on Table 4.1 one would expect a permanent hardware failure at most every 11.4 years, that is:

$$100\ \text{ECUs} \cdot \left(\frac{100\ \text{failures}}{10^9\ \text{device-hours}} \right) = \frac{1\ \text{failure}}{10^5\ \text{hours}} \approx \frac{1\ \text{failure}}{11.4\ \text{years}}.$$

Assuming a car life expectancy of 12–13 years like the authors in [NW03], this means that a permanent hardware failure will statistically happen mostly once in the lifetime of a car.[12] However, depending on the environment, transient hardware failure can occur at most every 10 hours:

$$100\ \text{ECUs} \cdot \left(\frac{10^6\ \text{failures}}{10^9\ \text{device-hours}} \right) = \frac{1\ \text{failure}}{10\ \text{hours}}.$$

[12] The authors state 12–13 years as a typical car life expectancy in the UK.

Even if we consider an orders of magnitude lower failure rate of 10,000 FIT a transient hardware failure could be expected every 1000 hours, i.e. approximately 42 days of system runtime. In particular, transient or intermittent faults, which remain in the system for a quite long time period, will probably result in a failure of the system. Consequently, fault tolerance is an important issue for safety-critical distributed systems.

In Section 2.3.3 we explained that fault tolerance always requires additional redundant components to detect faults and compensate them. An objective of our approach is, in accordance to the definition of Burns and Wellings [BW09], to minimize the required redundancy with respect to the cost and size constraints of the system. Especially, in huge distributed systems with many ECUs an unnecessary hardware overhead may result in an inappropriate rise of cost and weight. Obviously, in cars additional weight implies increasing fuel consumption resulting in additional cost and environment pollution and should be avoided [Ric09]. Therefore, we realize dynamic redundancy instead of static redundancy which would require large quantities of additional hardware. Furthermore, we focus on the compensation of the failure of one arbitrary ECU. This is essentially due to the following reasons. Regarding the dependency of component failures in a distributed system Kopetz defines [Kop11]:

> "*Given proper engineering precautions concerning the power supply and the electrical isolation of process signals have been made, the assumption that components of a distributed system that are physically at a distance will fail independently is realistic.*"

This means that if a distributed system fulfills these conditions its components are fully independent, i.e. a failure of one component will not necessarily result in a failure of other components. Since all components in a car have the same battery as common power supply, it is not justified to assume that the failures of the distributed ECUs are fully independent by default. However, if the battery respectively power supply of a car fails, obviously the whole electrical system will completely shut down. Nevertheless, the physical separation of the ECUs in a distributed system reduces the probability for spatial proximity faults, such that a fault at a single location, e.g. impact in case of an accident or electromagnetic interference does not affect more than a single ECU [Kop11]. Furthermore, several FlexRay transceivers like [Vec12] offer galvanically respectively electrically isolated interfaces which significantly reduce the risk of connected component failures.

The failure rates in Table 4.1 are determined based on statistical time-to-failure distributions and therefore implicitly represent the probability that the corresponding failure can be observed in the given time interval [MC89]. Even if we consider a quite long duration of a transient fault in the distributed

system, based on these values the probability that in this time interval another fault occurs is very low.[13] Moreover, obviously the overall fault tolerance level of a distributed system is just as high as the lowest fault tolerance level of the system's components. As mentioned above beside node failures, network failures are also possible. Since FlexRay offers one redundant communication channel, the bus is able to compensate one network component fault. To further increase the fault tolerance of the network another bus between all ECUs would be necessary. This again would mean an immense wiring and hardware overhead resulting in additional cost and weight which we aim to avoid. Furthermore, Buttazzo defines in [But11] that a real-time system should be fault-tolerant against single hardware and software failures. Thus, we focus on the compensation of one arbitrary component fault.

To offer the required redundancy for our fault-tolerant distributed system topology we perform task replication, i.e. allocation of redundant task instances on the ECUs. Task replication is a fundamental method for improving the reliability of a distributed system [Kim+11]. By introducing task replicas on other nodes, tasks on failed ECUs can be compensated [KLR10]. In [KLR10] the authors define two techniques for task replication, namely *Hot Standby* and *Cold Standby*:

Hot Standby: This approach uses two or more active replicas of a task. This means several active instances of each task are running simultaneously. Each replica must be executed on a different processor in order to make sure that at least one of them is working when a processor failure occurs. When a failure occurs, a replica will resume the task functionality of a failed processor. Obviously, redundancy by means of active replicas results in additional required computational resources because they are executed simultaneously.

Cold Standby: In this case, replicas are not active and executed until a failure occurs. This means that they are inactive by default and only use memory, but no processor resources until they are triggered and activated on demand when a failure occurs. Since Cold Standby replicas are not running under normal conditions, only the state information of each initial task needs to be exchanged and updated among replicas. This is the concept of task context migration described in Section 2.2.1. When a failure occurs, it should be detected as soon as possible and the tasks on failed processors should be recovered using replicas within a deterministic time interval.

As mentioned above, one objective of our approach is to reduce the required hardware redundancy. Therefore, it utilizes the Cold Standby approach and

[13] In particular, the probability of two or more independent events is the product of the probabilities of the individual events [Dev12].

maps inactive task replicas to the ECUs of the reconfigurable distributed system topology. Here, we consider that faults are detected as soon as possible by our fault detection and compensation concepts which we present in Section 4.2.2. Furthermore, like the authors in [KLR10], we assume that all necessary task information is available in memory. Thus, the actual realization and implementation of task context migration is beyond the scope of this thesis.

It is important to note that our approach is a support concept for the designer and not a fully automated solution. This means that fundamental design decisions, e.g. software architecture design and hardware topology setup are still performed by the designer. As shown in Figure 4.1 our approach is fed with the necessary information about the software architecture and hardware topology and performs a deployment trying to determine a feasible design solution. This contains task and bus mapping as well as corresponding schedulings for the initial configuration and based on that for all necessary reconfigurations to design a fault-tolerant system. Here, it is also important to note that our design approach is supposed to determine a feasible solution which fulfills the real-time constrains for the initial configuration and all possible reconfigurations. Thus, it is not intended to optimize one specific single attribute of the system, e.g. task response time, processor utilization, or communication delays. In fact it has to calculate efficient and flexible solutions which consider and combine several different properties. However, the main goal is the guaranteed fulfillment of the given real-time constraints to ensure a correct system behavior. This means, that the necessary timing properties and constraints have to be part of the input for our approach. Section 4.2.3 describes how we annotate the tasks and messages of the TDG representing the given software architecture with their corresponding timing information as part of the input model described in Section 4.3.1.

As described in Section 2.2.1 task scheduling in distributed systems implies global scheduling, i.e. task-to-ECU mapping, and local scheduling on the node. One main part of our design approach is the task mapping described in Section 4.4.1 which realizes redundancy by means of dynamic allocation. Particularly, it combines the advantages of both allocation concepts. On the one hand, it keeps the determinism of the static allocation to guarantee the given timing constraints. On the other hand, it increases the fault tolerance by including possible reallocations in the predetermined design solutions to provide the required redundancy and compensate node failures. Obviously, the global scheduling for every solution must be performed in such a way that the local scheduling can fulfill the given timing constraints for each task. In this context it has to be taken into account, that depending on the performed task mappings the required inter-node communication may result in additional communication delays for the task executions. These delays are part of the the task response times and have to be considered by the applied local

task scheduling strategy. Thus, in Section 4.2.4 we introduce our extensions for the schedulability tests of the commonly used real-time scheduling algorithms EDF and RM which integrate the resulting communication delays.

Beside an appropriate task mapping and scheduling the deployment of a distributed system also includes a corresponding bus mapping and scheduling. Therefore, an adequate bus mapping is another main objective of our design approach which is described in Section 4.4.2. Obviously, the given hardware topology has a great impact on the system deployment and has to be considered for the task mapping as well as for the bus mapping (cf. Figure 4.1). Hence, in Section 4.2.1 we provide important properties and constraints to support a system designer at the configuration of an appropriate hardware topology. These properties are used as part of the input model for our design approach which we describe in Section 4.3.1. This includes the node configuration which must ensure sufficient computational resources and communication capabilities, as well as the FlexRay bus setup. Especially the message mapping is influenced by the configuration of the FlexRay bus. Since the FlexRay protocol applies a cyclic slot-based time-triggered message scheduling, the message-to-slot mapping and the resulting communication delay heavily depend on the configuration of the bus parameters. Based on these configurations as part of the hardware topology input model, our approach performs bus mappings for the initial configuration and all possible reconfigurations. To further increase the efficiency and flexibility of the bus mapping and reduce the resulting communication delays, it also applies existing concepts for combined message-to-slot mappings which we describe in Section 4.2.6. As mentioned above, the implementation of task context migration is beyond the scope of this thesis. However, the remaining available communication capacities could be utilized for this purpose in future works.

4.2.1 Reconfigurable Distributed System Topology

As described in Section 2.2 a distributed system is composed of multiple networked processing nodes or ECUs cooperating on a common function or set of functions and exchanging data with the environment through sensors and actuators. Though, in current distributed systems one or more of these ECUs are, beside hosting functions, hardwired to sensors or actuators to exchange *input/output* (I/O) with the environment. This results in so-called *placement constraints* for functional tasks which have to exchange data with sensors or actuators because they have to be executed on the ECUs which are connected to the corresponding I/O. Consequently, a failure of a hardwired node cannot readily be compensated with dynamic redundancy and reconfiguration because connections to the peripherals and required I/O data might get lost. Hence, as fundamental part of our design approach we utilize the modified distributed system topology which we proposed and presented in [KKM11],

[Klo+11], and [KMR12] to avoid I/O-related placement constraints and enable fault-tolerant reconfiguration. In this reconfigurable distributed system topology we distinguish between two different types of nodes. We consider nodes, we call *peripheral interface nodes*. These nodes are hardwired to the sensors and the actuators of the embedded system and provide the required connection to the I/O. They solely read and write I/O data from and to the bus and do not execute any other function of the distributed system. Since peripheral interface nodes do not execute any complex task, they only require low hardware capacities. Therefore, static hardware redundancy is cost-efficiently applicable for these components. However, in this thesis we focus on the ECUs which execute the actual functionality of the distributed system and exchange there data over the communication network, e.g. the FlexRay bus. These *functional nodes* offer the necessary flexibility for our approach and can be utilized for dynamic redundancy with reconfiguration.

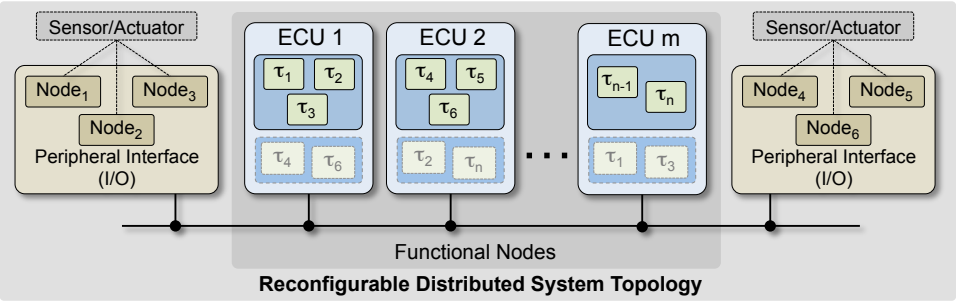


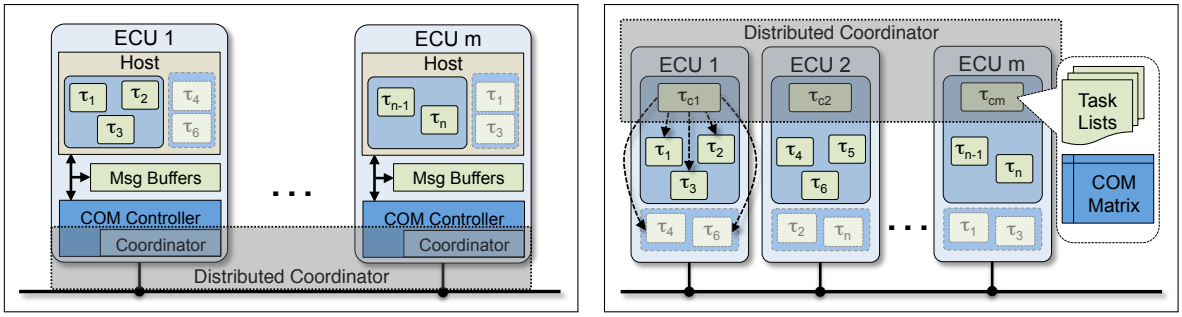
Figure 4.2: Concept for a reconfigurable distributed system topology.

Figure 4.2 depicts the schematic layout of the proposed reconfigurable distributed system topology. It contains peripheral interfaces which are composed of redundant interface nodes offering fault-tolerant exchange of I/O data with the sensors and actuators necessary for the interaction with the environment. Particularly with regard to our approach it offers functional nodes providing flexible and reconfigurable resources for the execution of the desired distributed system functionality. The number of ECUs depends on the number and performance requirements of the given functional tasks (cf. Section 4.2.5).

As illustrated in Figure 4.2, our design approach determines a feasible assignment of the n tasks and their redundant instances to the m functional ECUs to offer fault tolerance by means of dynamic redundancy. As mentioned in Section 4.2 we consider cold standby replicas for the redundant instances which are inactive by default and only use up memory capacities but no additional processor utilization [Kim+11]. Obviously, this reconfigurable distributed system topology allows homogeneous as well as heterogeneous functional nodes.

4.2.2 Distributed Coordinator Concept

To realize fault tolerance through dynamic redundancy in a distributed system, we have to detect errors, locate and disable faulty nodes before another resumes the execution of the functionality. Summarized, this means that a functionality which coordinates and performs the necessary steps for the reconfiguration has to be integrated in the system. This could be implemented by an auxiliary dedicated node as we described in [Klo+10c]. Obviously, the drawback of this solution is that a dedicated node is a possible SPOF. Hence, to retain fault tolerance such a coordinator ought to be replicated for static hardware redundancy. These required quantities of additional hardware components would result in increasing cost, size, and communication complexity of the system. Therefore, we propose a *distributed coordinator concept* (DCC) which allows the self-reconfiguration of the remaining ECUs in case of a node failure without the necessity for additional dedicated nodes.



(a) Distributed coordinator concept implemented in FlexRay communication controller as part of the FlexRay protocol.

(b) Distributed coordinator concept realized with additional tasks.

Figure 4.3: Two possible implementations of the distributed coordinator concept.

The distributed coordination of self-reconfiguration implies the addition of a coordinator component on each node in our proposed distributed system topology. Figure 4.3 illustrates two different possible implementations of the DCC. As shown in Figure 4.3(a) the coordinator component could be implemented in the FlexRay communication controller as an extension of the FlexRay protocol. This extension would make use of the controller host interface and the interface between CC and message buffers to reconfigure the task execution on the host and the corresponding transmit and receiver buffer assignments. However, this component is not part of the current FlexRay specification and therefore not implemented in any available FlexRay CC.

Hence, we propose another flexible and protocol-independent solution for the implementation of the DCC firstly presented in [Klo+10a]. As shown in Figure 4.3(b) we add distributed coordinator tasks to every node of the system. This DCC is implemented by lightweight tasks, which do not consume any

significant resources and runtime. Each instance of the distributed coordinator maintains task lists with running and redundant tasks of each node for every possible configuration and a COM matrix which provides necessary information about the corresponding communication dependencies. Because the memory demand for these data structures is negligible compared to the application data, it does not result in a mentionable hardware overhead to maintain it on each ECU. The distributed coordinator tasks must be configured to receive and monitor the messages of all relevant slots in order to detect node failures and to coordinate the appropriate configurations of the communication and the necessary task and slot assignments. Therefore, the task lists and the COM matrix are generated by means of the slot-to-node assignments and communication dependencies, determined with the deployment of our proposed design approach (cf. Section 4.4).

```
<taskLists>
  <taskListECU1>
    <initialConf>
      <activeTasks>
        <task>T_01</task>
        <task>T_02</task>
        <task>T_03</task>
      <\activeTasks>
      <redundantTasks>
        <task>T_04</task>
        <task>T_06</task>
      <\redundantTasks>
    <\initialConf>
    <reconfFailureECU2>
      <addActiveTasks>
        <task>T_04</task>
        <task>T_06</task>
      <\addActiveTasks>
    <\reconfFailureECU2>
    ...
  <\taskListECU1>
  <taskListECU2>
    <initialConf>
      <activeTasks>
        <task>T_04</task>
        <task>T_05</task>
        <task>T_06</task>
      <\activeTasks>
      <redundantTasks>
        <task>T_02</task>
        <task>T_n</task>
      <\redundantTasks>
    <\initialConf>
```

```
34      <reconfFailureECU1>
35        <addActiveTasks>
36          <task>T_02</task>
37          <\addActiveTasks>
38      <\reconfFailureECU1>
39      ...
40    <\taskListECU2>
41    ...
42  <\taskLists>
```

Listing 4.1: Excerpt of exemplary task lists.

Listing 4.1 gives an excerpt of XML-based task lists for the exemplary functional nodes of Figure 4.2. It shows parts of the active tasks and the inactive redundant tasks for the initial configuration and reconfigurations on ECU_1 and ECU_2. The lines 2 to 20 describe the task list for ECU_1 with the active tasks τ_1 (T_01), τ_2 (T_02), and τ_3 (T_03). Moreover, these lines define that if ECU_2 fails, the coordinator task on ECU_1 must additionally activate tasks τ_4 (T_04) and τ_6 (T_06). Analogous to this, if ECU_1 fails, ECU_2 must resume the execution of τ_2 (T_02) as defined in line 36.

Since the FlexRay protocol does not allow dynamic assignments of slots to senders during runtime (cf. Section 2.2.3), our design approach has to determine a priori the necessary communication reconfigurations for compensating node failures. Thus, the COM matrix contains auxiliary redundant transmission slots assigned to the ECUs, to be utilized after a reconfiguration of the communication. The total number of redundant transmission slots $|\Theta_{\text{redundant}}|$ in a COM Matrix can be calculated from the set of additional slots $\Theta_{\text{add}_{\text{reconf}_i}}$ required for a possible reconfiguration reconf_i:

$$|\Theta_{\text{redundant}}| = \left| \bigcup_{i=1}^{n} \Theta_{\text{add}_{\text{reconf}_i}} \right| \leq \sum_{i=1}^{n} \left| \Theta_{\text{add}_{\text{reconf}_i}} \right| . \tag{4.1}$$

Equation 4.1 states that the total sum of redundant slots is the union of additional slots required for all reconfigurations. Obviously, this number is less or equal to the sum of additional slots per reconfiguration. Our design approach reduces the number of redundant transmission slots by applying local intra-node communication, *slot reuse*[14], and frame packing to decrease the resulting communication overhead and increase the flexibility.

Figure 4.4 illustrates the determination of a COM matrix by means of an initial configuration and appropriate reconfigurations from our design approach. It illustrates an example for a communication matrix resulting from an initial configuration and one reconfiguration to compensate the failure of ECU_2. In

[14] Slot reuse means that a message-to-slot assignment is reused in one or more reconfiguations.

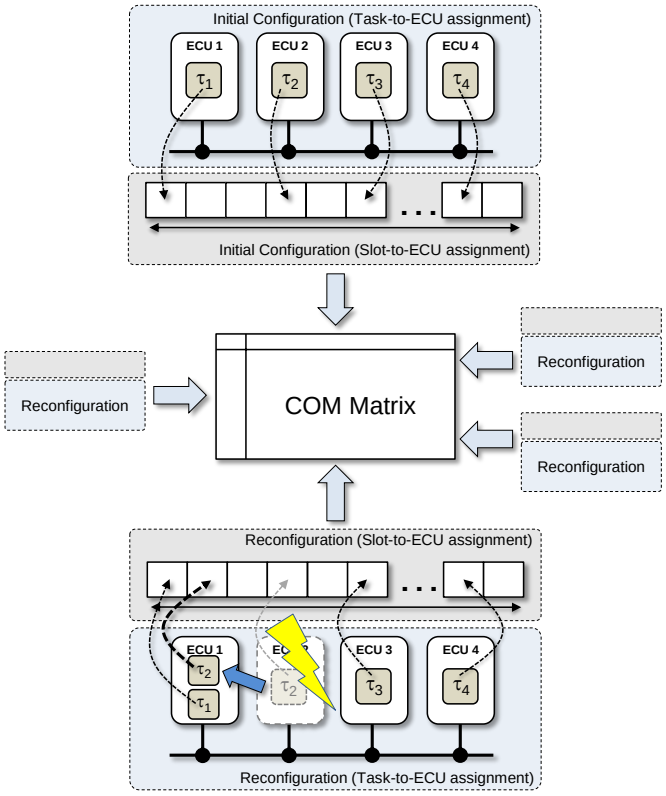


Figure 4.4: Example for COM matrix determination.

the initial configuration one task is assigned to each node. Thus, one transmission slot is assigned to each ECU, too. To compensate the failure of ECU_2, task τ_2 is resumed on ECU_1. This implies the reservation of an additional transmission slot for ECU_1 in the communication matrix. Finally, the complete communication matrix is determined considering all necessary additional transmission slots for each reconfiguration.

Based on the determined communication matrix and the task lists each task of the DCC performs the appropriate configuration for its hosting node. Figure 4.5 shows the flowchart of self-reconfiguration activities by means of the DCC in case of an error detection and fault localization. As mentioned in Section 4.2 our approach is supposed to compensate different kinds of faults which may either result in a fail-stop failure or an arbitrary Byzantine failure of an ECU.

Obviously to compensate a ECU fault by means of performing the self-reconfiguration, the DCC has to detect an error and localize the corresponding ECU failure. Therefore, the distributed coordinator tasks are configured to receive and monitor the messages of all relevant slots.[15] In case of a fail-

[15] Since there are no restrictions in the FlexRay protocol regarding the message reception of an ECU, theoretically each distributed coordinator task could receive messages from every slot.

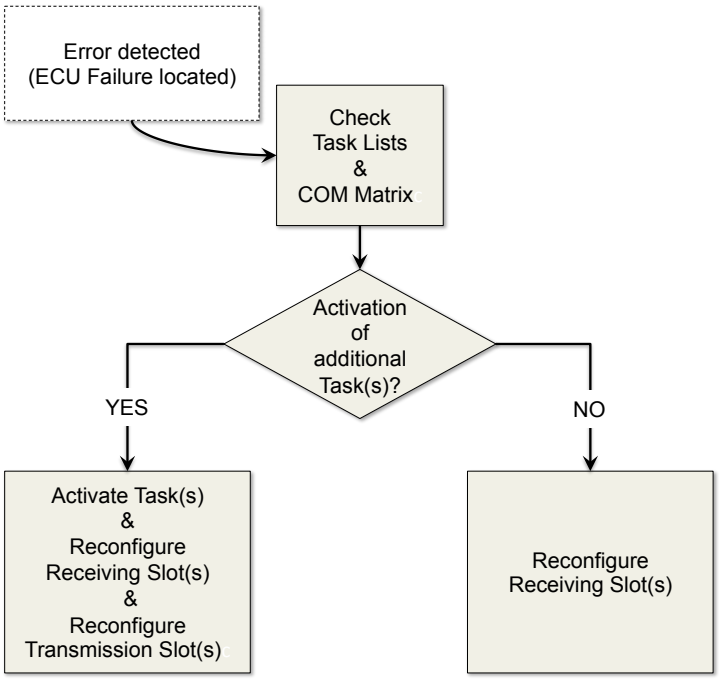


Figure 4.5: Self-reconfiguration activities of a distributed coordinator.

stop failure of an ECU, the payload segments of the FlexRay frames respectively slots assigned to this ECU are solely filled with padding bytes (cf. Section 2.2.3). This means that fail-stop failures are easily identifiable by the DCC monitoring the relevant slots and detecting null frames. Byzantine failures, where the faulty component continues to run but generates incorrect output, are obviously more complex and troublesome to deal with because they may result in non-predictable generation of erroneous output (cf. Section 4.2). As described in Section 2.2.3, FlexRay nodes offer optional bus guardians to protect the communication infrastructure against babbling-idiot failures, where a faulty node ignores the given slot assignments, transmits all the time and potentially consumes all slots. This mechanism ensures that a Byzantine failure will not result in a complete failure of the FlexRay network.

However, in case of a Byzantine failure, an ECU might still generate incorrect output and send it in its assigned slots. Therefore, we assume that the distributed coordinator tasks have the necessary information about value range consistency for the messages and frames transmitted in their corresponding slots. By means of these value ranges the DCC can detect strongly divergent values and identify them as erroneous output resulting from a Byzantine failure of the sender ECU. Appropriate value ranges for messages and output parameters heavily depend on the implemented functionality of the software architecture and the data exchange between its distributed tasks. However, their specification is beyond the scope of this thesis.

Based on these concepts and assumptions, the distributed coordinator tasks of the DCC are able to detect fail-stop and with high probability Byzantine failures of an ECU. Since the coordinator tasks must receive and monitor all relevant messages, they are scheduled after all functional tasks on their hosting ECU to ensure that they received all relevant messages before they are executed. Thus, the coordinator task will detect a fault within the duration of one hyperperiod H_Γ by analyzing the messages on the bus.[16] In the following Section 4.2.3 we describe that the hyperperiods in real-time systems are commonly in the milliseconds range. This enables short times and deterministic intervals needed for failure detection and recovery as required for the Cold Standby approach (cf. Section 4.2).

When a failure of an ECU is detected, each coordinator task checks via task lists and communication matrix if the necessary reconfiguration implies an activation of one or more additional tasks on its host. If no additional task executions on its hosting node are required, the coordinator task solely has to reconfigure the receiving slots for the input data of the executed tasks. By means of the task lists, each coordinator task knows for every possible reconfiguration which node(s) will resume executing the corresponding task(s) in case of a node failure. The COM matrix contains the necessary information of the resulting communication reconfigurations and the redundant slot assignments for the reconfigured task-to-ECU assignments. If the node failure implies the activation of one or more additional tasks on the host, the DCC additionally initiates the activation of the corresponding task(s) and the assignment of their transmission slots for the reconfiguration.

4.2.3 Specification of Timing Properties for the SW Architecture

To apply our design approach, we have to consider the timing properties and constraints of the intended functions of the distributed system. Therefore, the TDG representing the software architecture is augmented with the corresponding timing properties and constraints. To augment the graph, we apply *labeling functions* $f : V \to A$ and $g : E \to A$ to label the vertices $v \in V$ and the edges $e \in E$ with elements of specified attributes A, where $f(v)$ is the label that f assigns to the corresponding vertex $v \in V$ and $g(e)$ is the label that g assigns to the corresponding edge $e \in E$ [Len90].

Figure 4.6 depicts a timing-augmented TDG $G_{\text{TDG}} = (V, E) = (\Gamma, \mathcal{M})$. Each task τ_i is annotated with its WCET C_i by means of a labeling function:

$$\begin{array}{rccc} \mathfrak{c}: & \Gamma & \to & \mathbb{R}^+ \\ & \tau_i & \mapsto & C_i. \end{array} \tag{4.2}$$

[16] Section 4.2.5 describes in detail, how the FlexRay bus is synchronized to the hyperperiod.

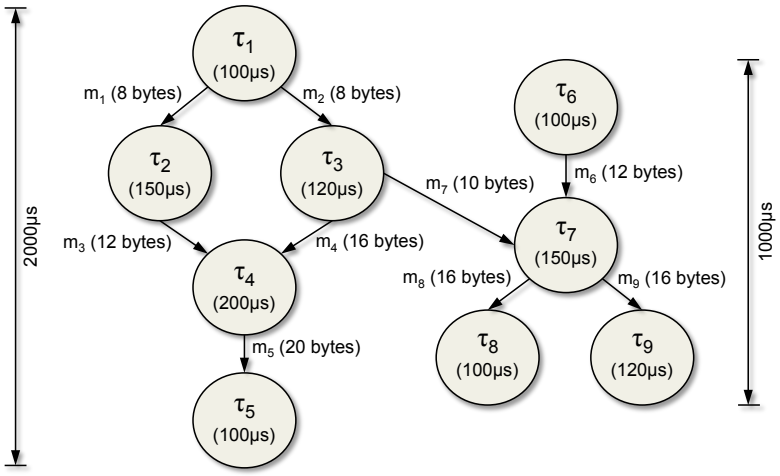


Figure 4.6: Timing-augmented TDG with split-message between two subgprahs.

As mentioned in Section 2.1.1 the software architecture of a distributed system can be composed of subgraphs and tasks with different periods. This example contains one with 1000μs and one with 2000μs. Each task of a subgraph has the same period. Like the authors in [DTT08], we define that not only the execution of each individual task but the execution of the whole TDG subgraph has to be finished in this time, i.e. it defines the maximum end-to-end delay. As described in Section 2.2.4 TADL2 allows annotating timing constraints to structural system models. Therefore, we utilize it to augment the TDG by the definition of timing constraints to guarantee a correct system behavior.

```
1  Event InputRecvByT_01 {
2  //Input received by task T_01 event
3  }
4  ...
5  Event OutputSentByT_09 {
6  //Output sent by task T_09 event
7  }
8
9  PeriodicConstraint InputT_01 {
10 //Periodic constraint for reading input of task T_01
11   event InputRecvByT_01
12   period = 2000 micros on universal_time
13   minimum = 0
14   jitter = 0
15 }
16 ...
17 PeriodicConstraint OutputT_09 {
18 //Periodic constraint for reading input of task T_09
19   event OutputSentByT_09
```

```
20   period = 1000 micros on universal_time
21   minimum = 0
22   jitter = 0
23 }
```

Listing 4.2: Excerpt of TADL2 periodic constraints for TDG in Figure 4.6.

Listing 4.2 gives an excerpt of the periodic constraints for the TDG presented in Figure 4.6. For each task the reception of input data and sending of output data is modeled as an event and the event occurrences are constrained by the corresponding periods in μs. For instance, in line 9 to 15 it defines that task τ_1 (T_01) has to read input data every 2000μs, whereas τ_9 (T_09) has to provide its output data every 1000μs. The internal reference time base universal_time was defined in Listing 2.1.

```
1  DelayConstraint E2E_T_01_T_05 {
2  //Delay constraint(max.E2Edelay) for input of T_01 to output of
       T_05
3    source InputRecvByT_01
4    target OutputSentByT_05
5    lower = 0
6    upper = 2000 micros on universal_time
7  }
8  ...
9  DelayConstraint E2E_T_01_T_09 {
10 //Delay constraint(max.E2Edelay) for input of T_01 to output of
       T_09
11   source InputRecvByT_01
12   target OutputSentByT_09
13   lower = 0
14   upper = 1000 micros on universal_time
15 }
16 ...
17 }
```

Listing 4.3: Excerpt of TADL2 delay constraints for TDG in Figure 4.6.

Listing 4.3 provides an excerpt of the delay constraints for the TDG shown in Figure 4.6. It shows the end-to-end delay constraint for two different event chains. The constraints impose the limits for the time between the input event (stimulus) of task τ_1 (T_01) and the output events (responses) of task τ_5 (T_05) in line 1 to 7 respectively τ_9 (T_09) in line 9 to 15. The limits of the end-to-end delay are set to 2000μs and 1000μs corresponding to the periods of the tasks. The event chains are composed of several included input and output events for the involved tasks (cf. Section 2.2.4).

Figure 4.6 also illustrates the *split message* m_7 representing the dependency between τ_3 and τ_7 which are elements of two subgraphs with different peri-

ods. Obviously, as shown in the Listings above, the scheduling of split messages implies specific timing constraints depending on the periods of sender task τ_{tx} and receiver task τ_{rx}:

- $T_{tx} < T_{rx}$: In this case the sender period is smaller than the receiver period. This means that it is sufficient for the sender τ_{tx} to transmit the message with the period of the receiver τ_{rx}. This avoids unnecessary messages without a receiver resulting in additional communication overhead.
- $T_{tx} > T_{rx}$: In this case the messages are still transmitted with the sender period because only after each execution of the sender new data is available. Hence, this case does not influence the message scheduling. However, it should be ensured that τ_{tx} can transmit the messages soon enough to reduce obsolete data.

In [DTT08] the authors propose to constrain the possible periods for subgraphs and tasks to facilitate the system design. Therefore, in a system with n tasks and a maximum task period $T_{max} = \max\{T_i | i = 1, \ldots, n\}$, for each task τ_i it must hold:

$$T_i \cdot 2^{x_i} = T_{max}, \text{ with } x_i \in \mathbb{N}_0, \ i = 1, \ldots, n. \tag{4.3}$$

This results in a so-called *harmonic task set* where the maximum task period equals the hyperperiod ($T_{max} = H_\Gamma$). For instance, in a system with $T_{max} = 4ms$ the other possible periods are $2ms, 1ms, 500\mu s$, and so on. In real-world applications task sets mostly have harmonic periods [Eis+10]. This results from the fact, that harmonic task sets allow higher processor utilization for static fixed-priority scheduling strategies. More precisely, every task of a harmonic task set scheduled with RM can meet its deadline for $U \leq 1.0$ [Liu00]. Even if the whole task set Γ is not harmonic, still higher utilization bounds are possible by identifying harmonic subsets in Γ [KM97]. The utilization of harmonic periods also facilitates the scheduling of split messages described above. Hence, we also consider the condition above as a timing constraint of the software architecture model for our design approach, if possible. To be more flexible in the system design our approach also supports non-harmonic task sets with $T_{max} \neq H_\Gamma$ to a certain extend. Section 4.2.5 describes the necessary configurations and the given limitations of the FlexRay communication for supporting this.

Moreover, each message m_i of the TDG is annotated with its data length L_{m_i} by a labeling function:

$$\begin{array}{rccc} \mathfrak{d}: & \mathcal{M} & \rightarrow & \mathbb{R}^+ \\ & m_i & \mapsto & L_{m_i}. \end{array} \tag{4.4}$$

Obviously, the data length of a message implies a certain time needed for its transmission. As described in Section 2.2.3 the TDMA-based FlexRay protocol offers a deterministic transmission and timing behavior by means of a static slot-to-ECU assignment and a guaranteed bus scheduling strategy. Nevertheless, in addition to the task WCETs, the transmission time of the exchanged messages on a path from one end of the TDG to the other one is part of the resulting end-to-end delay. Thus, to fulfill the defined maximum end-to-end delay constraints, the resulting communication delays for inter-node messages have to be considered and reduced if necessary (cf. Section 2.2.2).

4.2.4 Task Scheduling Strategy

Task scheduling in distributed systems has to cover the local level of each processor and the global scheduling, i.e. the mapping of tasks to ECUs (cf. Section 2.2.1). Our design approach combines the advantages of static and dynamic task allocation. It keeps the determinism of the static allocation and guarantees the timing constraints. Additionally, it increases the fault tolerance by including possible reallocations in the predetermined reconfigurations to compensate node failures. Furthermore, for a real-time system the determined task mapping must ensure that the local scheduling strategy can guarantee the fulfillment of the timing constraints of each task on each ECU.

As described in Section 2.2.2 distributed system tasks have to communicate locally via intra-ECU communication or remotely over the network via inter-ECU communication. The transmission time of a message via local communication can be considered as negligible. The transmission of messages between distant tasks may result in a communication delay, i.e. the execution of a task which receives input data from a task over the network may be delayed by the transmission time of the message. Therefore, not only the computation but also additionally the transmission has to be finished before the deadline of the task. In general, the time needed for the message transmission will be distinctly smaller than the computation time of a task. Nevertheless the remote communication may delay the task execution significantly depending on the actual scheduling. This implies, that beside task WCETs, for a proper schedulability test the resulting communication delays of a task mapping have to be considered, too. Consequently, we have to extend the schedulability tests for the scheduling strategies RM/DM and EDF* presented in Section 2.1.2 to also consider communication delays.

For EDF* scheduling we have to integrate the resulting communication delay δ_i in the processor demand criterion from Equation 2.5. Therefore, we extend

this test to the *extended processor demand criterion* PDC*, as we proposed in [KMR12]:

$$\forall L > 0 \quad \sum_{i=1}^{n} \left\lfloor \frac{L + T_i - D_i}{T_i} \right\rfloor \cdot (C_i + \delta_i) \leq L. \tag{4.5}$$

Considering the properties described in Section 2.1.2, this results in the PDC* schedulability test which must be checked at least until the largest relative deadline $D_{\max}$:

$$\begin{aligned} &\forall L \in \mathcal{D} \quad \sum_{i=1}^{n} \left\lfloor \tfrac{L+T_i-D_i}{T_i} \right\rfloor \cdot (C_i + \delta_i) \leq L, \\ \text{with} \quad &\mathcal{D} = \{d_k | d_k \leq \min[H, \max(D_{\max}, L^*)]\}. \end{aligned} \tag{4.6}$$

For DM scheduling we have to integrate the resulting communication delay δ_i in the response time analysis from Equation 2.4. This results in the *extended response time analysis* RTA*, as we proposed in [Klo+13]:

$$R_i = (C_i + \delta_i) + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j. \tag{4.7}$$

As mentioned above, for intra-ECU communication the delay can be considered as $\delta_i = 0$. For inter-ECU messages the resulting communication delay depends on the given schedule and its parameters. Therefore, the task τ_{last} scheduled last before τ_i on the same ECU and the set of sender tasks Γ_{tx} sending inter-ECU messages to τ_i must be considered. Before τ_i can be executed, the transmission time $\psi_{m_{(\tau_j,\tau_i)}}$ of the latest input message from n senders has to be completed. Thus, τ_i may start at the maximum of its release time r_i, the finishing time f_{last} of τ_{last} and the finishing time of the latest input message transmitted by $\tau_j \in \Gamma_{\text{tx}}$. Considering these properties, this results in:

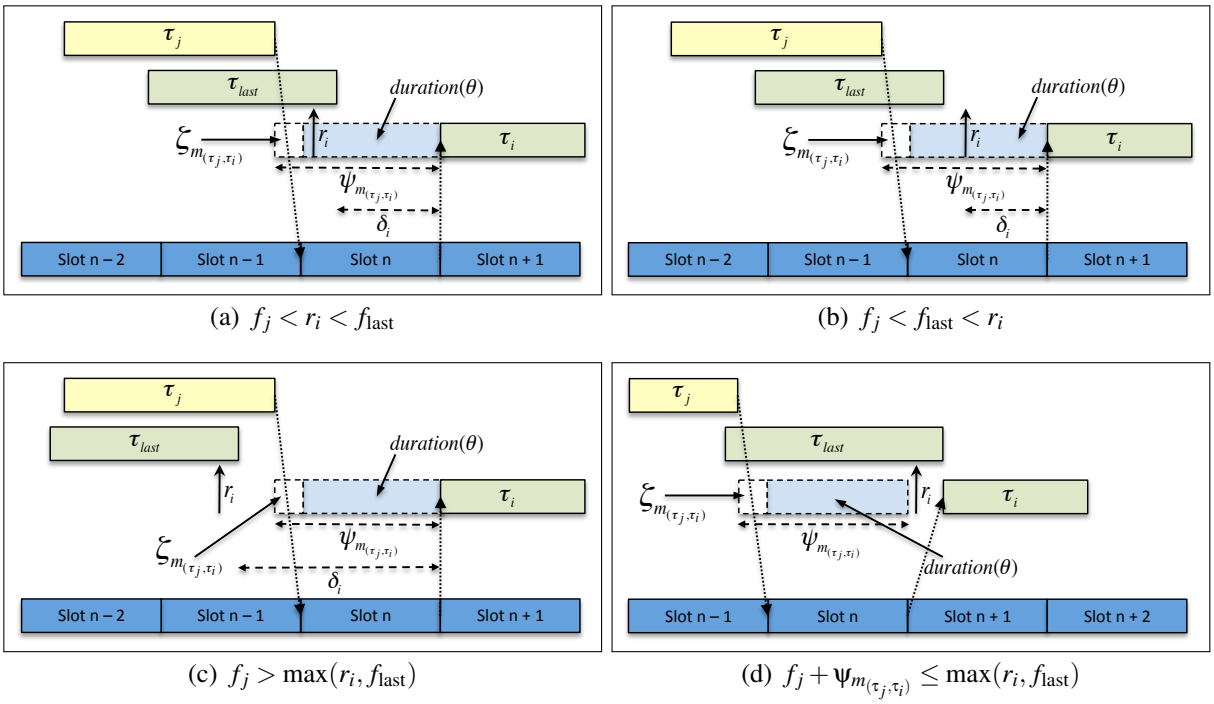
$$\delta_i = \max\left(0, \max\{f_j + \psi_{m_{(\tau_j,\tau_i)}} | \tau_j \in \Gamma_{\text{tx}}, j = 1, ..., n\} - \max\{r_i, f_{\text{last}}\}\right). \tag{4.8}$$

The FlexRay protocol applies TDMA-based bus write accesses, i.e write accesses are only allowed in assigned slots (cf. Section 2.2.3). This implies a transmission start delay $\zeta_{m_{(\tau_j,\tau_i)}}$ for a message $m_{(\tau_j,\tau_i)}$. The transmission start delay represents the time interval between the finishing time of the sender task τ_j and the start time of the assigned time slot $\theta_{m_{(\tau_j,\tau_i)}}$ in the FlexRay cycle:

$$\zeta_{m_{(\tau_j,\tau_i)}} = s_{\theta_{m_{(\tau_j,\tau_i)}}} - f_j.$$

Since the message data is not available for the receiving task τ_i until the end of the assigned slot the transmission time also contains the duration of the assigned slot $\theta_{m_{(\tau_j,\tau_i)}}$ which is equal for each slot. Summarized $\psi_{m_{(\tau_j,\tau_i)}}$ is defined as:

$$\psi_{m_{(\tau_j,\tau_i)}} = \zeta_{m_{(\tau_j,\tau_i)}} + \text{duration}(\theta). \tag{4.9}$$



(a) $f_j < r_i < f_{\text{last}}$

(b) $f_j < f_{\text{last}} < r_i$

(c) $f_j > \max(r_i, f_{\text{last}})$

(d) $f_j + \psi_{m_{(\tau_j,\tau_i)}} \leq \max(r_i, f_{\text{last}})$

Figure 4.7: Calculation of δ_i for different scheduling scenarios.

Figure 4.7 illustrates different examples for the determination of δ_i based on the parameters defined in Equation 4.8 and 4.9. It shows how different scheduling scenarios may influence the additional communication delay and result in an increased response time R_i of the receiver task τ_i. In the example of Figure 4.7(a) the condition $f_j < r_i < f_{\text{last}}$ holds, i.e. the execution of τ_i is already delayed by τ_{last}. This means that the additional communication delay of s_i depends on f_{last}. Thus, the communication delay is calculated as:

$$\delta_i = f_j + \psi_{m_{(\tau_j,\tau_i)}} - f_{\text{last}}.$$

In Figure 4.7(b) the order of the parameters is $f_j < f_{\text{last}} < r_i$. This means that the execution of τ_i is not delayed by τ_{last} which results in a total delay of s_i relative to r_i. Here, the communication delay is calculated as:

$$\delta_i = f_j + \psi_{m_{(\tau_j,\tau_i)}} - r_i.$$

Since $f_j < \max(r_i, f_{\text{last}})$ holds, $\delta_i < \psi_{m_{(\tau_j,\tau_i)}}$ also holds in both cases. This implies that the message transmission is partly performed during the execution of τ_{last} and/or before r_i. If $f_j > \max(r_i, f_{\text{last}})$ holds, as in the example shown in Figure 4.7(c), the communication delay may further increase, even to a value of $\delta_i > \psi_{m_{(\tau_j,\tau_i)}}$. In this case $m_{(\tau_j,\tau_i)}$ cannot be transmitted during the execution of τ_{last} or before r_i and the value of δ_i gets maximized. Figure 4.7(d) provides an example for the other extreme case where $f_j + \psi_{m_{(\tau_j,\tau_i)}} \leq \max(r_i, f_{\text{last}})$ holds. In this case the inter-ECU communication does not imply any delay, i.e. $\delta_i = 0$. As shown in the figure the complete message transmission is performed during the execution of τ_j and completed before r_i and f_{last} and the message is buffered until τ_i starts.

All examples in Figure 4.7 also illustrate that the overall transmission time is composed of the transmission start delay and the slot duration as defined in Equation 4.9. In all examples $\zeta_{m_{(\tau_j,\tau_i)}} < \text{duration}(\theta)$ holds because $m_{(\tau_j,\tau_i)}$ is transmitted in the next FlexRay slot θ after f_j. If the next slot cannot be assigned to $m_{(\tau_j,\tau_i)}$, the transmission start delay increases by the number of not assignable slots to:

$$\zeta_{m_{(\tau_j,\tau_i)}} \geq k \cdot \text{duration}(\theta),$$

where k is the number of not assignable slots after f_j.

Depending on the applied local task scheduling strategy the corresponding schedulability test, as described above, has to be fulfilled to guarantee a feasible schedule for each ECU resulting in a valid task scheduling for the complete distributed system. Therefore, the PDC* respectively RTA* calculation based on the determination of the resulting values for the transmission time $\psi_{m_{(\tau_j,\tau_i)}}$ and the communication delay δ_i by means of the given parameters is the basis for our task mapping approach described in Section 4.4.1.

In Section 2.1.2 we mentioned that EDF has some significant advantages over RM/DM scheduling: EDF allows higher processor utilization and has lower runtime overhead. A common argument contra EDF is that it is more complicated to implement because of its dynamical priority assignment during runtime. Therefore, in many important real-time domains fixed-priority scheduling algorithms are still most frequently applied. For instance in the automotive domain RM/DM scheduling is the most widely utilized scheduling approach [FFR12]. Hence, our approach supports EDF as well as RM/DM with their corresponding schedulability tests as described above.

4.2.5 Configuration of Hardware Architecture

The design of a distributed real-time system also implies the necessity for an appropriate configuration of the hardware architecture on which the software is executed. It must enable an execution of the system functions which guarantees a correct system behavior including the fulfillment of the given timing constraints. Essentially, this means that the number and performance of computational node resources must be sufficient to execute the given set of functional tasks with respect to the given real-time constraints. Furthermore, the communication infrastructure must offer the required rates and capacities to ensure a reliable, deterministic, and timely data transmission. Therefore, our design approach considers the given functional and timing-related properties of the software architecture to derive an appropriate configuration of the hardware topology including a node setup and a FlexRay bus setup.

Node Setup

A distributed real-time system is composed of a set of networked computational nodes. Thus, a set $\mathcal{E}$ of n ECUs can be defined as:

$$\mathcal{E} = \{\text{ECU}_j | \, j = 1, \ldots, n\}. \tag{4.10}$$

As input for our design approach we model a *hardware architecture graph* (HAG) representing the available ECUs in the network and their interconnection. Figure 4.8 illustrates an exemplary HAG $G_{\text{HAG}} = (V, E) = (\mathcal{E}, E)$. The vertices V represent the set of ECUs $\mathcal{E}$. The edges E represent the communication connection between these nodes. Since we consider bus communication, all ECUs are connected with each other by an undirected edge.

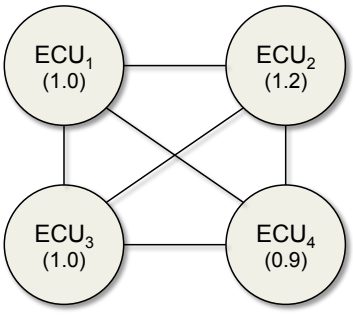


Figure 4.8: Exemplary hardware architecture graph.

The WCET of a task τ_i depends on the resources of ECU_j it is executed on and can be formalized as:

$$C_{\text{ECU}_j, \tau_i}. \tag{4.11}$$

In a homogeneous distributed system all nodes are of the same type, i.e. they have equal capacities (cf. Section 2.2). Thus, in a homogeneous system the condition:

$$C_{\mathrm{ECU}_j,\tau_i} = C_{\mathrm{ECU}_j,\tau_i}, \forall \tau_i \in \Gamma,\ \mathrm{ECU}_j, \mathrm{ECU}_k \in \mathcal{E}, \tag{4.12}$$

holds. Equation 4.12 states, that in a homogeneous system a specified task τ_i has the same WCET $C_{\mathrm{ECU}_j,\tau_i} = C_i$ on each ECU_j. Obviously, the nodes of a distributed real-time system must be configured such that each task of the given software architecture can fulfill its functional and its timing constraints. In particular, this means that each task meets its deadline by means of an appropriate scheduling. As described in Section 2.1.2 utilization-based schedulability tests are sufficient and guarantee the feasibility of a task set on an ECU. The value for U_{lub} depends on the scheduling strategy and on the given task set. It is at least $U_{\mathrm{lub}} = 0.69$ for arbitrary task sets and up to $U_{\mathrm{lub}} = 1$ for harmonic task sets when utilizing RM or DM. For EDF it is also $U_{\mathrm{lub}} = 1$. In any case, the maximum possible utilization of one ECU is $U_{\mathrm{lub}} = 1$.

Based on that, we can determine the lower limit of ECUs required to execute n tasks in a distributed system. Considering a homogeneous system, the WCET $C_{\mathrm{ECU}_j,\tau_i} = C_i$ of a task τ_i is identical on each ECU. Moreover, the period T_i of a task τ_i is a timing constraint which is independent from the ECU. Thus, the lower limit m of required ECUs can be calculated as:

$$\left\lceil \sum_{i=1}^{n} \frac{C_{\mathrm{ECU}_j,\tau_i}}{T_i} \right\rceil \leq m \cdot U_{\mathrm{lub}}. \tag{4.13}$$

For instance, we consider a maximum utilization of $U_{\mathrm{lub}} = 1$ and five tasks with a common period of $T_i = 1000\mu s$. Moreover, we assume the following WCETs for the tasks: $C_1 = 600\mu s$, $C_2 = 300\mu s$, $C_3 = 500\mu s$, $C_4 = 400\mu s$, $C_5 = 400\mu s$. For this example, the summed utilization on the left side of Equation 4.13 is $U = 2.2$. This results in $m = 3$ for the lower limit of required ECUs. However, our fault-tolerant design approach utilizes dynamic redundancy to compensate the failure of an arbitrary ECU. Therefore we have to add an additional ECU resulting in:

$$\left\lceil \sum_{i=1}^{n} \frac{C_{\mathrm{ECU}_j,\tau_i}}{T_i} \right\rceil \leq (m-1) \cdot U_{\mathrm{lub}}. \tag{4.14}$$

This means that for the example described above we consider $m = 4$ for the lower limit of required ECUs in the initial configuration to provide sufficient ECU resources required for the reconfigurations. Even if this additional ECU

implies a hardware overhead, this dynamic redundancy concept significantly reduces it compared to static hardware redundancy, particularly in huge systems with many ECUs (cf. Section 2.3.3). As described in Section 4.2.4, task dependencies and inter-ECU communication may result in delays of task executions. Thus, the determined number of ECUs may not be sufficient to fulfill the timing constraints of a given task set. In this case, our approach returns this result to the system designer, who can adjust the hardware topology for a further iteration with additional or more powerful ECUs. Nevertheless, since we want to reduce the resulting hardware overhead for the fault tolerance, the determination of this lower bound for required ECUs is an adequate support for the system designer to perform an initial hardware configuration.

Although we focus on single ECU failures in this thesis, for the sake of completeness it should be mentioned that for the compensation of multiple faulty ECUs, the right side of Equation 4.14 can be extended to $m - x$ with $x > 1$. Obviously, this results in additional hardware overhead and unused resources in configurations with less than x ECU failures.

Our approach also supports heterogeneous systems to a certain extend. Therefore, each ECU in the HAG is annotated with an *execution time factor* by a labeling function:

$$\begin{array}{rccc} \mathfrak{f}: & \mathcal{E} & \rightarrow & \mathbb{R}^+ \\ & \mathrm{ECU}_j & \mapsto & \mathfrak{f}(\mathrm{ECU}_j). \end{array} \tag{4.15}$$

This factor represents the performance relation between the underlying reference ECU considered for the TDG and an ECU_j. To calculate the WCET of a task τ_i on an ECU_j, our approach takes the WCET C_i from the TDG as reference value and divides it by the corresponding execution time factor:

$$C_{\mathrm{ECU}_j,\tau_i} = \frac{C_i}{\mathfrak{f}(\mathrm{ECU}_j)}.$$

For instance, if ECU_1 is the underlying reference ECU of the TDG and ECU_2 is 20% more performant, then the execution time factor of ECU_2 is $\mathfrak{f}(\mathrm{ECU}_2) = 1.2$. In this example, for task τ_3 ($C_3 = 120\mu s$) from the TDG in Figure 4.6, the WCET on ECU_2 is $C_{\mathrm{ECU}_2,\tau_3} = 100\mu s$. Figure 4.8 shows the execution time factors for an exemplary HAG including the reference value $\mathfrak{f}(\mathrm{ECU}_1) = 1.0$.

It is important to remind that our fault-tolerant design approach supports heterogenous systems solely to a certain extend. Indeed we can also determine the lower limit of required ECUs with Equation 4.14 utilizing the execution time factors. But if the deviation of capabilities for one or more of the ECUs is to too high, the design will get complicated and inefficient. For instance, the remaining ECUs of a reconfiguration will probably not be able to compensate the failure of an ECU with multiple capacities. Thus, this setup will

result in a SPOF or a much higher need for redundancy capacities and hardware overhead. Therefore, the level of heterogeneity in the system should not be too high.

Besides computational resources each node also has to provide sufficient communication capabilities for the required data transmissions. As described in Section 2.2.3 in a FlexRay network a message buffer on each ECU implements an application-decoupled and flexible message transmission and reception with respect to timing. To set up a message buffer scheme for the network, i.e. message buffers for each node, the following steps are necessary [RS06]:

- Consider the communication needs of the system, i.e. type and number of messages that have to be transmitted between the nodes.
- Determine the characteristics of the messages, particularly the required payload sizes.
- Configure the message buffers to offer the required functionality, i.e. setting up the message buffer registers and allocating the required shared memory for the message data.

Parts of the message buffer configuration can already be done at the hardware configuration stage: For instance, the maximum number of message buffers to be implemented and the maximum payload size to be supported. However, most of the message buffer configuration, such as assigning message buffers to slots and channels, selecting transmit or receive, and setting up cycle count filtering is done during the configuration stage of the FlexRay protocol at system design [RS06]. Consequently, for our approach we consider a message buffer setup which allows storing messages with the maximum payload size defined for the slots in each message buffer. Obviously, the number and size of messages respectively slots depends on the FlexRay bus setup that we describe in the next section.

FlexRay Bus Setup

In [ASH06] the authors summarize that the FlexRay protocol specification lists 74 parameters and their possible settings span a space of more than 10^{48} (theoretical) configurations. This results in an increased configuration complexity for the system design. Setting up all of these parameters and configuring the communication on FlexRay is a burdensome task for a system designer and therefore addressed by numerous publications like [ASH06], [MN08], [PS11], and [Jan+11]. Thus, in this thesis we focus on the configuration of the most important parameters for the FlexRay schedule to determine a fault-tolerant system by means of our design approach. However,

many of the protocol parameters influence each other and can be derived directly or indirectly based on these basic parameters.

Bandwidth (Transmission Rate) For the bandwidth of the FlexRay bus, we configure a transmission rate of 10 Mbit/s. Even if the current version of the FlexRay protocol also supports rates of 5 Mbit/s and 2.5 Mbit/s, we set up the maximum available transmission rate to reduce the resulting transmission delays for inter-ECU communication. As mentioned in Section 2.2.3 the transmission of 1 byte of data needs $1\mu s$ at a transmission rate of 10 Mbit/s.

Communication Cycle In this thesis we consider a pure static FlexRay cycle without dynamic segment and without symbol window. Furthermore, we consider a NIT with a length of just a few microseconds. Thus, the communication cycle Θ_{cycle} can be defined as a set of n static slots:

$$\Theta_{\text{cycle}} = \{\theta_i | \, i = 1, \ldots, n\}. \tag{4.16}$$

As input for our design approach we model a *communication graph* (COMG) representing the available static slots in a communication cycle. Figure 4.9 illustrates an exemplary COMG $G_{\text{COMG}} = (V, E) = (\Theta, \emptyset)$. The vertices represent the set of available slots Θ with $\Theta = \Theta_{\text{cycle}}$. Since there is no functional relation between the slots, we model G_{COMG} as a graph without edges, i.e. $E = \emptyset$.

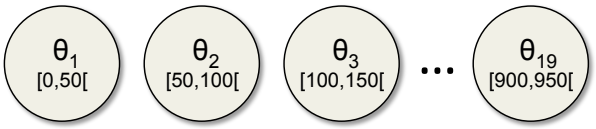


Figure 4.9: Exemplary communication graph.

Each of the n slots θ_i is annotated with its *slot time interval* by means of a labeling function $\mathfrak{s}$ defined as:

$$\begin{array}{rccc} \mathfrak{s}: & \Theta & \rightarrow & \{[s_{\theta_i}, f_{\theta_i}[\, | \, 1, \ldots, n\} \\ & \theta_i & \mapsto & [s_{\theta_i}, f_{\theta_i}[. \end{array} \tag{4.17}$$

To facilitate the synchronization of task executions and message transmissions and enable efficient corresponding schedulings, we set the length of the FlexRay communication cycle to the hyperperiod H_Γ of the given task set:

$$\text{length}(\Theta_{\text{cycle}}) = H_\Gamma.$$

As defined in Equation 4.3, if Γ is a harmonic task set, $T_{\max} = H_\Gamma$ holds. Thus, we set the cycle length for a harmonic task set to:

$$\text{length}(\Theta_{\text{cycle}}) = T_{\max}.$$

For instance, for the TDG depicted in Figure 4.6 the cycle length would be set to $\text{length}(\Theta_{\text{cycle}}) = T_{\max} = 2000\mu s$. But to be more flexible, we do not limit our approach to these harmonic task sets. Thus, we also support non-harmonic task sets with $T_{\max} \neq H_\Gamma$. In this case we set the communication cycle length to the hyperperiod H_Γ of the given task set which is the least common multiple of the task periods (cf. Section 2.1.2). Nevertheless, there are some major constraints, caused by the FlexRay specification, limiting the support of such task sets to a certain extend. As mentioned in Section 2.2.3 the maximum length of a FlexRay communication cycle is $16ms$. This means that for the support of longer periods cycle multiplexing must be utilized. However, for cycle multiplexing FlexRay solely offers cycle repetitions of $Rep = \{5, 10, 20, 40, 50\}$ as well as $Rep \in \{2^n | n = 0, \ldots, 6\}$ (cf. Section 2.2.3). Consequently, theoretically hyperperiods up to $H_\Gamma = 2^6 \cdot 16ms = 1024ms$ are possible. Even for shorter hyperperiods and for harmonic task sets cycle multiplexing can be applied. For instance, for the TDG in Figure 4.6 the cycle length can be configured as $\text{length}(\Theta_{\text{cycle}}) = 1000\mu s$ with a cycle repetition of $Rep = 2$. Generalized, our approach sets the communication cycle length to the task set hyperperiod, where the possible values for H_Γ are defined as:

$$\begin{aligned} & H_\Gamma = \text{length}(\Theta_{\text{cycle}}) \cdot Rep \leq 1024ms, \\ \text{with} \quad & \\ & \text{length}(\Theta_{\text{cycle}}) \leq 16ms, \\ & Rep \in \{5, 10, 20, 40, 50, 2^n | n = 0, \ldots, 6\}. \end{aligned} \tag{4.18}$$

As described in Section 2.2.3 a network idle time for the node clock synchronization is mandatory in the communication cycle. The NIT is a communication-free period at the end of the cycle and cannot be used for message transmission. This means, the duration of the NIT reduces the length of the static segment within the communication cycle. However, a NIT duration of a few microseconds is sufficient and the slot duration must be larger than $13\mu s$ to transmit any payload data. Therefore, we consider a deliberately kept short NIT duration. This results in the loss of just the last or at most a few slots for the static segment at the end of the communication cycle.

Slots The size of the static slots within the communication cycle is configurable by the system designer and allows a payload in each slot between 0 and 254 bytes. As mentioned above at a transmission rate of 10 Mbit/s we

can define that the time duration of a slot in microseconds equals the slot size in bytes:

$$\text{duration}(\theta) = \text{size}(\theta).$$

Based on that, the number of slots within the communication cycle can be calculated as:

$$\left|\Theta_{\text{cycle}}\right| = \frac{\text{length}(\Theta_{\text{cycle}})}{\text{duration}(\theta)} - \text{NIT}. \tag{4.19}$$

As mentioned above the available time for the static slots is reduced by the NIT which we configure as long as one slot or as integer multiple of the configured slot duration. Obviously, the configured slot size must also be an integer divisor of the remaining time in the communication cycle length. Although the slot size is freely configurable under the above-mentioned constraints, the system designer must consider the number and size of messages in the given software architecture. To avoid unnecessary transmission delays, the slot size should be sufficient to transmit even the largest message within the payload of one slot.

With Equation 4.19 the number of available slots within the COMG is calculated as input for our design approach. By means of the labeling function in Equation 4.17 each slot is annotated with its corresponding slot time interval $[s_{\theta_i}, f_{\theta_i}[$. Obviously, the size of the slot time interval is the slot duration. Thus, the start time of slot θ_i is calculated as:

$$s_{\theta_i} = (i-1) \cdot \text{duration}(\theta), \tag{4.20}$$

and the finishing time of slot θ_i is calculated as:

$$f_{\theta_i} = i \cdot \text{duration}(\theta). \tag{4.21}$$

Figure 4.9 depitcs the slots and slot time intervals for an exemplary COMG resulting from the following FlexRay bus configurations:

- Communication cycle length: $\text{length}(\Theta_{\text{cycle}}) = 1000\mu s$,
- Slot duration: $\text{duration}(\theta) = 50\mu s$,
- Network Idle Time: NIT = $\text{duration}(\theta) = 50\mu s$.

The configured cycle length would allow 20 slots with slot time interval of $50\mu s$. Since the NIT reduces the available time by one slot duration, the COMG contains 19 slots as shown in Figure 4.9. For instance, the slot time interval for slot θ_2 is $[50\mu s, 100\mu s[$.

As described in Section 2.2.3 the payload data in a FlexRay frame is stored and addressed in two-byte words, i.e. each message needs multiples of 2 bytes in the payload segment of a FlexRay frame respectively slot. Hence, the required payload length of a message m_i with an original data length L_{m_i} is defined as:

$$L_{m_i} \leq \text{payload}(m_i) = n \cdot 2\,\text{bytes} \leq \text{maxPayload}(\theta),\ n \in \mathbb{N}_0.$$

For instance, the largest message of the TDG in Figure 4.6 is 20 bytes. Thus, due to the FlexRay protocol overhead, we propose to set the slot size to at least $\text{size}(\theta) = 20\,\text{bytes} + 13\,\text{bytes} = 33\,\text{bytes}$ (cf. Section 2.2.3).[17] Even if this results in a larger slot size, it is often reasonable to transmit multiple messages from one sender ECU in one slot because this can further decrease transmission delays and communication overhead. Section 4.2.6 provides details on the concepts supported by our approach to enable combined message transmission in one slot.

4.2.6 Protocol Data Unit Concept & Frame Packing

As described in Section 2.2.3 FlexRay utilizes padding bytes to fill the remaining bytes if the message to transmit is smaller than the configured payload size. Depending on the given message sizes and the defined slot size, this may result in unnecessary communication overhead for messages from the same sender ECU transmitted in different slots. Therefore, several Flexray communication stacks, like the one presented in [GK07], support the *protocol data unit* (PDU) concept.

As shown in Figure 4.10 several PDUs can be combined within the payload segment of a FlexRay frame [Rau08]. PDUs may have different sizes, i.e. contain a variable number of two-byte data words. This implies that it is possible to encapsulate the data exchanged between ECUs in messages respectively PDUs [Par12]. The FlexRay communication stack gets the message data from the host via CHI and processes the PDUs into frames and vice versa [GK07].

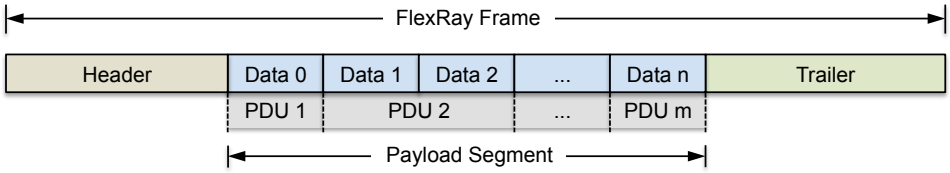


Figure 4.10: Example for PDU concept in a FlexRay frame.

[17] Theoretically, it is also possible to split up a large message and transmit it in several slots. But for a higher practicability we assume at least slot sizes sufficient to transmit the largest message.

Since FlexRay slots are statically assigned to one sender ECU, the PDU concept allows us to perform *frame packing* for messages sent by the set of tasks Γ_{ECU_k} hosted by ECU_k. This means tasks may send different messages combined in one frame if they are executed on the same ECU. This condition can be formalized as:

$$\mathrm{ECU}_{\tau_i} = \mathrm{ECU}_{\tau_j}, \text{ with } \tau_i \neq \tau_j \in \Gamma_{\mathrm{ECU}_k} \subset \Gamma.$$

Obviously, the summed size of the messages m_i may not be larger than the available maximum payload of the slot θ. For n different messages this condition can be defined as:

$$\sum_{i=1}^{n} \mathrm{payload}(m_i) \leq \mathrm{maxPayload}(\theta).$$

Figure 4.11 illustrates an example for frame packing. The tasks τ_1 and τ_2 are executed on the same ECU, i.e. $\mathrm{ECU}_{\tau_1} = \mathrm{ECU}_{\tau_2}$. Since the condition $\mathrm{payload}(m_1) + \mathrm{payload}(m_2) \leq \mathrm{maxPayload}(\theta)$ also holds, the messages m_1 and m_2 (PDU 1 and PDU 2) can be combined within the payload segment of the same FlexRay frame.

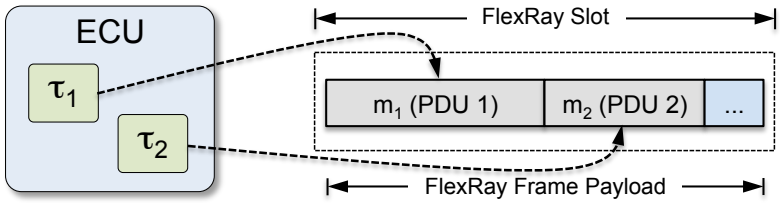


Figure 4.11: Example for frame packing.

Frame packing enables a more efficient and flexible bus scheduling of messages. Since multiple messages from one sender ECU can be combined in one frame, less slots must be assigned to this ECU. This allows a better utilization of individual payload of frames and slots which results in a decreased communication overhead in these slots. Furthermore, it also facilitates earlier transmission of messages if a message is scheduled combined with other messages in one slot instead of being transmitted in the subsequent slots. Consequently, this may result in shorter communication delays and response times for the receiver tasks of messages. The reduced number of required slots also increases the flexibility of the slot assignments. This enables our approach to assign slots which are not already assigned to an ECU to a different ECU in another configuration.

4.3 Modeling of System Properties

These sections describe the graph-based modeling of the system properties as input for our design approach. This includes the modeling of the software architecture and the hardware topology in Section 4.3.1 as well as the determination of the resulting timing constraints and communication properties derived from the given inputs and presented in Section 4.3.2. For a better traceability, all model elements and performed steps of the design approach are illustrated based on an example we published in [KMR12].

4.3.1 Software Architecture and Hardware Topology

The software architecture of a distributed real-time system offers and controls one or more functions implemented with a number of dependent tasks. A task set Γ of such a system with n tasks can be modeled as:

$$\Gamma = \{\tau_i = (T_i, C_i, r_i, d_i, s_i, f_i) | \, i = 1, \ldots, n\}. \tag{4.22}$$

As described in Section 2.1.1 each task τ_i defined in Equation 4.22 is characterized by several parameters. This includes the period T_i and the WCET C_i, a release time r_i at which the task can be started and a deadline d_i at which the execution of τ_i must be finished. These parameters are globally defined and constrain the possible scheduling of the given task set to guarantee a correct system behavior. Additionally, a task is described by a start time s_i and a finishing time f_i. The actual execution, i.e. start and finish of a task, depends on the performed scheduling of the task set. The applied scheduling strategy has to guarantee that start and finishing time of each task lie within its available execution time interval:

$$\Psi_i = [r_i, d_i[\text{ with } r_i \leq s_i < f_i \leq d_i.$$

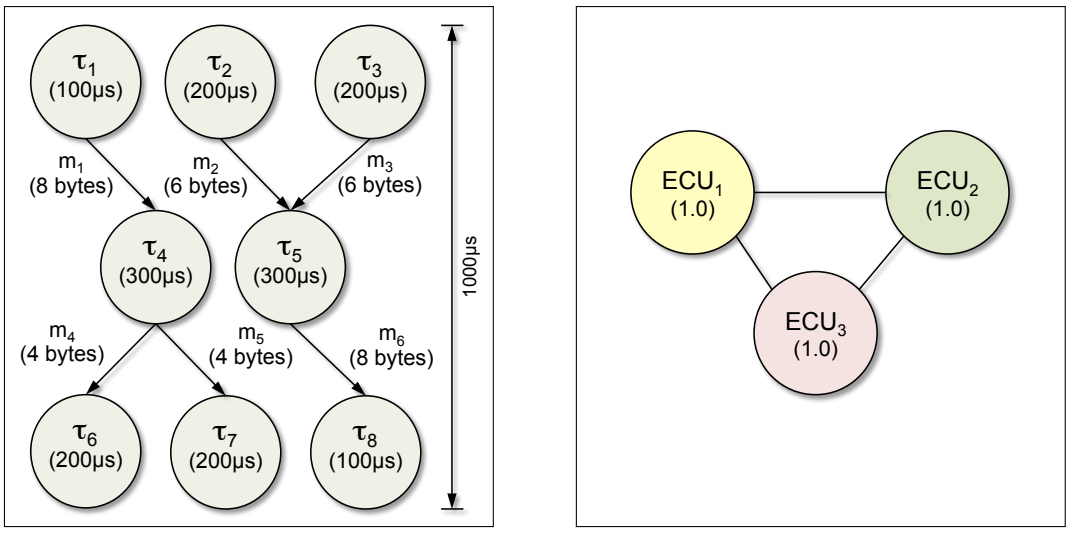
Distributed tasks in a real-time network exchange data by means of real-time message communication. Similar to the task set a message set $\mathcal{M}$ of n real-time messages can be modeled as:

$$\mathcal{M} = \{m_i = (T_{m_i}, L_{m_i}, r_{m_i}, d_{m_i}, s_{m_i}, f_{m_i}) | \, i = 1, \ldots, n\}. \tag{4.23}$$

Equation 4.23 defines that each message m_i is characterized by a period T_{m_i} which directly depends on the period T_{tx} of the sender task τ_{tx} and the period T_{rx} of the receiver task τ_{rx} (cf. Section 4.2.3). Moreover, each message m_i has a data length L_{m_i} to transmit, a release time r_{m_i}, and a deadline d_{m_i}. As described in Section 2.2.2 we define that r_{m_i} and d_{m_i} limit the available

transmission time interval $\Upsilon_{m_i} = [r_{m_i}, d_{m_i}[= [f_{\text{tx}}, s_{\text{rx}}[$. This means that the transmission of message m_i can be started earliest when τ_{tx} has finished its computation and must be finished latest when τ_{rx} is started. Therefore, the start time s_{m_i} and finishing time f_{m_i} of m_i must lie within this interval:

$$f_{\text{tx}} = r_{m_i} \leq s_{m_i} < f_{m_i} \leq d_{m_i} = s_{\text{rx}}.$$



(a) Software architecture modeled with timing-augmented TDG.

(b) Hardware topology modeled with HAG.

Figure 4.12: Graph-based modeling of system properties as input for design approach (example from [KMR12]).

To represent the task dependencies and specify the timing properties of the given software architecture, we model this input for our approach as a timing-augmented task dependency graph $G_{\text{TDG}} = (\Gamma, \mathcal{M})$ as described in Section 4.2.3. Figure 4.12(a) depicts an example TDG from [KMR12] composed of eight tasks $\Gamma = \{\tau_1, \ldots, \tau_8\}$ and six messages $\mathcal{M} = \{m_1, \ldots, m_6\}$. Each task τ_i is annotated with its WCET C_i and each message m_i with its data length L_{m_i} by means of the labels $\mathfrak{c}(\tau_i)$ and $\mathfrak{d}(m_i)$ defined in Section 4.2.3.

According to the definition from [DTT08], we distinguish between three different task subsets in the TDG to reference them in the following descriptions:

- $\Gamma_{\text{in}} \subset \Gamma$: The subset of tasks (vertices) in the TDG which have no predecessors and incoming messages (edges).
- $\Gamma_{\text{out}} \subset \Gamma$: The subset of tasks (vertices) in the TDG which have no successors and outgoing messages (edges).

- $\Gamma_{mid} \subset \Gamma$: Contains all tasks (vertices) in the TDG which are not element of Γ_{in} or Γ_{out}. This means, these tasks have predecessors and successors as well as incoming and outgoing messages (edges).

Moreover, there are different approaches described in literature to define and model the messages and data exchange in a TDG. For example, on the one hand, the authors in [DTT08] define that each message sent by a task is different and must therefore be scheduled separately on the bus. This corresponds to the case illustrated in the example TDG of Figure 4.12(a) where task τ_4 sends two different messages m_4 and m_5. On the other hand, the authors in [KHM03] assume that one task only sends one specific message instance which can be received by one or multiple tasks and is identical for all receivers. For the example of Figure 4.12(a) this would mean that $m_4 = m_5$ would hold. To allow and support more flexible system designs, we principally consider and utilize the modeling with differently defined and separately scheduled messages. However, our approach also supports the concept of identical message instances.

Listing 4.4 provides the XML-based input for our design approach representing the TDG in Figure 4.12(a). In the lines 2 to 11 it shows how the tasks are defined and annotated with their corresponding WCETs. Furthermore, in the lines 12 to 31 it represents the communication dependencies between the tasks via defining senders and receivers for each individual message and annotates each message with its corresponding data length. For instance line 13 to 15 define message m_1 (M_01) with a data length of 8 bytes to be transmitted from sender task τ_1 (T_01) to receiver task τ_4 (T_04). This allows our approach to directly derive the precedence constraints of the given software architecture.

```
1  <TDG>
2    <tasks>
3      <task ID="T_01" WCET="100" />
4      <task ID="T_02" WCET="200" />
5      <task ID="T_03" WCET="200" />
6      <task ID="T_04" WCET="300" />
7      <task ID="T_05" WCET="300" />
8      <task ID="T_06" WCET="200" />
9      <task ID="T_07" WCET="200" />
10     <task ID="T_08" WCET="100" />
11   </tasks>
12   <messages>
13     <message ID="M_01" Payload="8" SenderID="T_01">
14       <ReceiverID>T_04</ReceiverID>
15     </message>
16     <message ID="M_02" Payload="6" SenderID="T_02">
17       <ReceiverID>T_05</ReceiverID>
```

```
18      </message>
19      <message ID="M_03" Payload="6" SenderID="T_03">
20        <ReceiverID>T_05</ReceiverID>
21      </message>
22      <message ID="M_04" Payload="4" SenderID="T_04">
23        <ReceiverID>T_06</ReceiverID>
24      </message>
25      <message ID="M_05" Payload="4" SenderID="T_04">
26        <ReceiverID>T_07</ReceiverID>
27      </message>
28      <message ID="M_06" Payload="8" SenderID="T_05">
29        <ReceiverID>T_08</ReceiverID>
30      </message>
31    </messages>
32  </TDG>
```

Listing 4.4: TDG input for example in Figure 4.12(a).

The overview in Figure 4.1 shows that besides the software architecture our design approach also needs the given hardware topology as input. Therefore, we model it as a hardware architecture graph $G_{\text{HAG}} = (\mathcal{E}, E)$ introduced in Section 4.2.5. In the example from [KMR12] illustrated in Figure 4.12(b), we assume a distributed system topology with three functional nodes respectively ECUs, i.e. the HAG has three vertices. These ECUs communicate over the FlexRay bus represented by the undirected edges between the ECUs. As described in Section 4.2.5, each ECU_j is annotated with an execution time factor by means of the label $\mathfrak{f}(\text{ECU}_j)$. However, in this example we consider a homogeneous system topology resulting in an execution time factor of 1.0 for each ECU_j. Listing 4.5 gives the XML-based input for our design approach representing the HAG in Figure 4.12(b) and shows how the ECUs annotated with their corresponding execution time factor are defined. Moreover, for a better traceability and identification in the following concepts and design steps, the ECUs in the example HAG are marked with different colors.

```
1  <HAG>
2    <ecus>
3      <ecu ID="ECU_01" ETFactor="1.0" />
4      <ecu ID="ECU_02" ETFactor="1.0" />
5      <ecu ID="ECU_03" ETFactor="1.0" />
6    </ecus>
7  </HAG>
```

Listing 4.5: HAG input for example in Figure 4.12(b).

In Section 4.2.2 we introduced our distributed coordinator concept to enable a self-reconfiguration of the remaining ECUs in case of a node failure. For this concept we have to add distributed coordinator tasks to every node of the sys-

tem. Although, the DCC is implemented by lightweight tasks not consuming any significant computational resources they still have to be scheduled and therefore be considered for the schedulability test of the applied scheduling strategy (cf. Section 4.2.4). Hence, our design approach takes the input data from the TDG and HAG to derive an extended TDG which adds one coordinator task per ECU to the given application tasks of the software architecture.

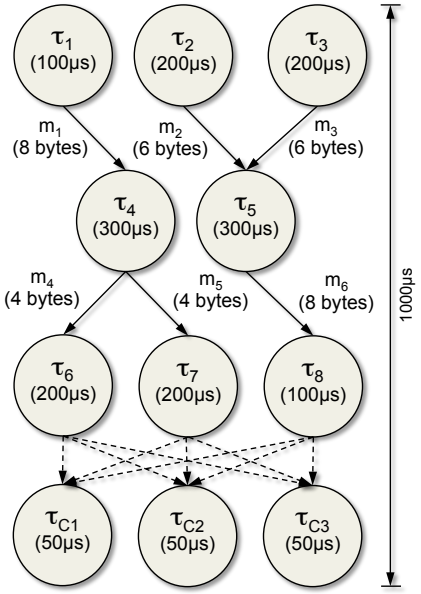


Figure 4.13: Extended TDG based on input graphs in Figure 4.12.

Figure 4.13 depicts the resulting extended TDG for the input graphs in Figure 4.12. Since the given hardware topology is composed of three ECUs, the extended TDG also contains the three coordinator tasks τ_{c_1}, τ_{c_2}, and τ_{c_3}. Each coordinator task is attached to all tasks in Γ_{out} with dashed directed edges, to specify that they will be scheduled after the last application task on their corresponding hosting ECU. Obviously, this implies that the coordinator task must also be considered on each ECU for the schedulability tests we presented in Section 4.2.4. As depicted in Figure 4.13, we assume that all distributed coordinator tasks have the same WCET, in this example $50\mu s$. Therefore, it is sufficient to add one universal generic definition of the coordinator task to TDG input as shown in line 5 of Listing 4.6.

```
1 <TDG>
2   <tasks>
3     ...
4     <task ID="T_08" WCET="100" />
5     <task ID="T_c" WCET="50" />
6   </tasks>
7   ...
8 </TDG>
```

Listing 4.6: Coordinator task definition added to TDG input in Listing 4.4.

This section presented our graph-based modeling of the software architecture and the hardware topology as input for our design approach. In the following section we describe how extend these input models with the additional information about timing constraints and communication properties required for an appropriate real-time system design.

4.3.2 Timing Constraints and Communication Properties

In Section 4.2.3 we described how we utilize TADL2 to augment a TDG by the definition of timing constraints as input for our approach to guarantee a correct system design and behavior. According to the exemplary TADL2 constraints shown in Listing 4.2 and 4.3 we define periodic constraints and end-to-end delay constraints for the task execution and message transmissions in the given software architecture represented by the timing-augmented TDG. As shown in Figure 4.13 the period constraints respectively end-to-end delay constraints for this example are defined as $1000\mu s$.

Due to these timing and precedence constraints, the individual release time r_i and deadline d_i of each task τ_i are successively calculated by means of the corresponding parameters of the predecessors and successors, as we initially described in [KMR12]. The individual release times r_i are calculated as:

$$r_i = \begin{cases} 0 & ,\text{if } \tau_i \in \Gamma_{\text{in}} \\ \max\{r_j + C_j \,|\, \tau_j \in \Gamma_{\text{directPre}_i}\} & ,\text{else.} \end{cases} \tag{4.24}$$

If a task τ_i has no predecessors, i.e. $\tau_i \in \Gamma_{\text{in}}$, its available execution time interval $\Psi_i = [r_i, d_i[$ starts at $r_i = 0$. For tasks with one or more predecessors, Equation 4.24 calculates the value for r_i by means of the parameters r_j and C_j of the direct predecessors of τ_i, i.e. $\tau_j \in \Gamma_{\text{directPre}_i} \subset \Gamma$. This means, it successively calculates the sum(s) of the computation times of the paths from τ_i backwards to its source(s) in the TDG and assigns the maximum value of this sum(s) to r_i.

In a similar manner the individual deadlines d_i are calculated as:

$$d_i = \begin{cases} \text{max end-to-end delay} & ,\text{if } \tau_i \in \Gamma_{\text{out}} \\ \min\{d_k - C_k \,|\, \tau_k \in \Gamma_{\text{directSucc}_i}\} & ,\text{else.} \end{cases} \tag{4.25}$$

If a task τ_i has no successors, i.e. $\tau_i \in \Gamma_{\text{out}}$, the available execution interval of τ_i ends at $d_i =$ max end-to-end delay as specified in the given timing constraints (cf. Section 4.2.3). For tasks with one or more successors, Equation 4.25 determines the value for d_i based on parameters d_k and C_k of the direct successors of τ_i, i.e. $\tau_k \in \Gamma_{\text{directSucc}_i} \subset \Gamma$. It successively combines the computation times along the paths from τ_i to their sink vertices and subtracts the

sum from the constrained maximum end-to-end delay. The minimum value of all paths is assigned to d_i.

As described above, we extend the given TDG by adding distributed coordinator tasks for the DCC. This means that the available execution time intervals for the application tasks within the constrained maximum end-to-end delay are reduced by the WCET of the coordinator task. Therefore, the WCET C_c of the coordinator task τ_c has to be integrated in Equation 4.25 resulting in:

$$d_i = \begin{cases} \text{max end-to-end delay} - C_c & \text{, if } \tau_i \in \Gamma_{\text{out}} \\ \min\{d_k - C_k \,|\, \tau_k \in \Gamma_{\text{directSucc}_i}\} & \text{, else.} \end{cases} \tag{4.26}$$

To enable a correct design and behavior of a distributed real-time system, the determination of these boundaries is required for all tasks by means of Equation 4.24 and 4.26. Table 4.2 provides the resulting values for the example TDG in Figure 4.13.

Task τ_i	Release Time r_i $[\mu s]$	Deadline d_i $[\mu s]$
τ_1	0	450
τ_2	0	550
τ_3	0	550
τ_4	100	750
τ_5	200	850
τ_6	400	950
τ_7	400	950
τ_8	500	950
τ_c	600	1000

Table 4.2: Available execution time intervals Ψ_i for extended TDG in Figure 4.13.

Depending on the individual scheduling and communication of tasks the start time s_i is delayed and not the whole calculated interval is available to meet the given deadline d_i. For example, a task may start its execution at its release time, i.e. $s_i = r_i$, if just the tasks contained in the determined predecessor path with the maximum computation time are scheduled to the same ECU as τ_i. If other additional tasks are scheduled before τ_i its execution is delayed, i.e. $s_i > r_i$. The same holds for predecessors which are executed on other ECUs because the necessary communication may also delay the start of the task execution. Therefore, these constraints have to be considered by the global scheduling strategy. Obviously, in a heterogeneous system these values must be calculated individually for each ECU by means of the specific WCET C_{ECU_j,τ_i} depending on its execution time factor as described in Section 4.2.5.

Beside task execution also the necessary message transmissions have to be performed within the defined timing boundaries. Consequently, the given communication properties have to be considered for a proper system design, too. This includes the message properties and the bus configuration. The data lengths are already annotated to the messages in the timing-augmented TDG and therefore available as input for our design approach (cf. Figure 4.13). As described in Section 4.2.5, we also model a communication graph $G_{\text{COMG}} = (\Theta, \emptyset)$ as input for our approach, which represents the bus configuration. Figure 4.14 shows the COMG for the example FlexRay network from [KMR12] with a transmission rate of 10 Mbit/s and a NIT duration of one slot. It represents a configured cycle length of $\text{length}(\Theta_{\text{cycle}}) = T_{\max} = 1000\mu s$ and slot duration of $\text{duration}(\theta) = 25\mu s$, derived from the properties and constraints of the TDG in Figure 4.13. The resulting slot time interval is annotated to each slot θ_i of the COMG by means of the label $\mathfrak{s}(\theta_i)$.

At a transmission rate of 10 Mbit/s the duration of a slot in microseconds equates to the slot size in byte. This means that in this configuration each slot can transmit a maximum payload data of 12 bytes (cf. Section 2.2.3). Consequently, frame packing could be utilized for several of the messages with sizes from 4 to 8 bytes depending on the task-to-ECU mapping performed by our design approach. Listing 4.5 gives the XML-based input for our design approach representing the COMG in Figure 4.14 providing the necessary communication properties defined above.

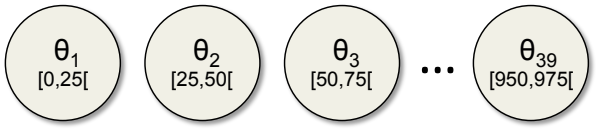


Figure 4.14: Communication graph for example system from [KMR12].

```
<COMG>
  <cycleLength> 1000 </cycleLength>
  <slotLength> 25 </slotLength>
  <slotsNIT> 1 <slotsNIT>
</COMG>
```

Listing 4.7: COMG input for example in Figure 4.14.

As described in Section 2.2.2 the properties of the message transmission also depend on the global scheduling. After the task-to-ECU mapping, the resulting available transmission time interval Υ_{m_i} of a message m_i is limited by the finishing time of the sender task τ_{tx} and the start time of the receiver task τ_{rx}.[18] For intra-node messages the communication is realized by means of shared data structures. For inter-node communication the message has to be

[18] Considering the message model with multiple receivers from [KHM03], the available transmission time interval is limited by the receiver with the earliest start time.

mapped to one of the FlexRay slots within the available transmission time interval. Consequently, the available transmission time interval must be at least one slot duration, i.e. $\Upsilon_{m_i} \geq \text{duration}(\theta)$, to enable a feasible message scheduling. Therefore, the set of available slots Θ_{m_i}, which is a subset of the complete set of FlexRay slots Θ_{cycle} modeled in the COMG, has to be determined. Our design approach calculates the set of available slots Θ_{m_i} for each inter-node message m_i by means of Υ_{m_i} and $\text{duration}(\theta)$ as:

$$\begin{aligned} \Theta_{m_i} &= \left\{ \theta_j \,\middle|\, \left\lceil \frac{r_{m_i}}{\text{duration}(\theta)} \right\rceil + 1 \leq j \leq \left\lfloor \frac{d_{m_i}}{\text{duration}(\theta)} \right\rfloor \right\} \\ &= \left\{ \theta_j \,\middle|\, \left\lceil \frac{f_{\text{tx}}}{\text{duration}(\theta)} \right\rceil + 1 \leq j \leq \left\lfloor \frac{s_{\text{rx}}}{\text{duration}(\theta)} \right\rfloor \right\}. \end{aligned} \tag{4.27}$$

Obviously, the available slots within an available transmission time interval are a connected subset. Figure 4.15 illustrates the relation between Υ_{m_1} and Θ_{m_1} for an example inter-node message $m_1 = m_{(\tau_1,\tau_2)}$. In this example the message m_1 is released at the finishing time of task τ_1, during the duration of slot 2 (θ_2) and has its deadline at the start time of task τ_2, after the duration of slot 4 (θ_4). With Equation 4.27 the available slots for m_1 are determined as $\Theta_{m_1} = \{\theta_3, \theta_4\}$.

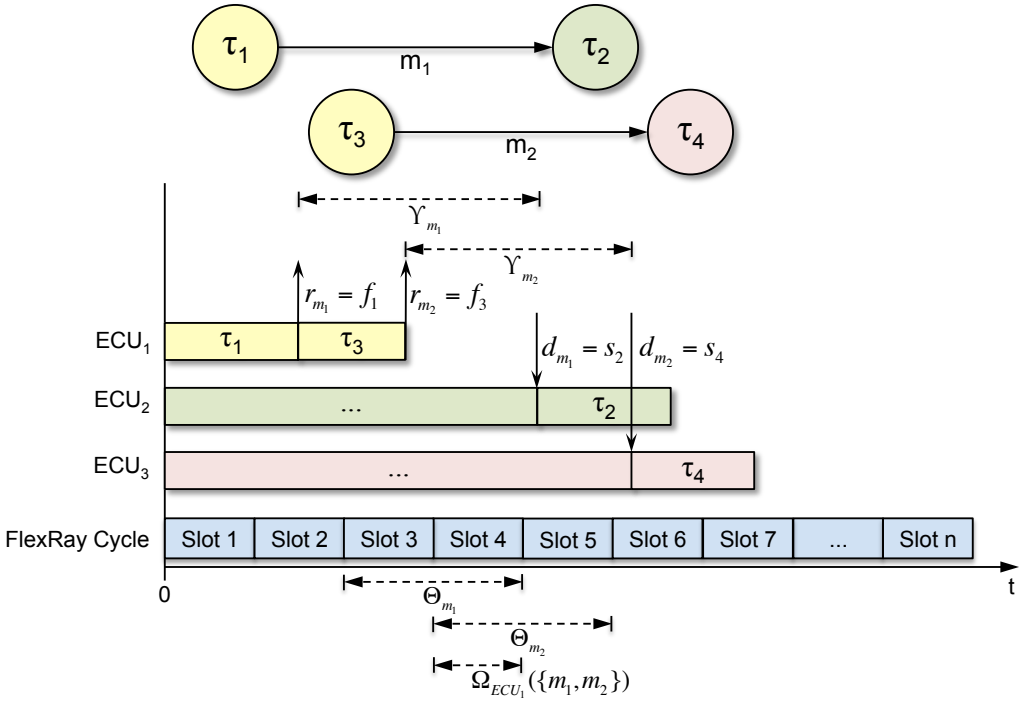


Figure 4.15: Overlapping available slots in available transmission time intervals.

Slot Reduction and Multirate Systems

The determination of the individual available slots for inter-node messages is very useful as input information for the bus mapping described in Section 4.4.2. Especially in huge and complex distributed systems with many ECUs, probably many messages will be transmitted via inter-ECU communication. However, as described in Section 2.2.3 a FlexRay slot can only be assigned statically to one specific sender ECU. In particular, to increase the flexibility for a feasible and correct fault-tolerant system design including all necessary reconfigurations, our approach is supposed to reduce the number of required slots per sender ECU. Therefore, it analyses the available slots for message subsets $\mathcal{M}$ of the inter-ECU messages $\mathcal{M}^{\text{bus}}_{\text{ECU}_{\text{tx}}}$ transmitted by a sender ECU_{tx} to determine overlapping available slots $\Omega_{\text{ECU}_{\text{tx}}}(\mathcal{M})$. Depending on the communication properties, i.e. message data length and slot sizes, two or even more of the messages sent by ECU_{tx} can be combined in one of the overlapping slots to perform an efficient bus mapping.

Figure 4.15 gives an example for overlapping available slots. The tasks τ_1 and τ_3 are mapped to the same sender ECU_1. The inter-node messages m_1 and m_2 have the available transmission time intervals Υ_{m_1} and Υ_{m_2} resulting in the available slot sets $\Theta_{m_1} = \{\theta_3, \theta_4\}$ and $\Theta_{m_2} = \{\theta_4, \theta_5\}$. Our design approach can identify the overlapping slot $\Omega_{\text{ECU}_1}(\{m_1, m_2\}) = \theta_4$ for the sender ECU_1. This means, that the bus mapping may combine the messages m_1 and m_2 in slot 4 if frame packing is possible, i.e. $\text{payload}(m_1) + \text{payload}(m_2) \leq \text{maxPayload}(\theta)$ (cf. Section 4.2.6).

In Section 4.2.3 we described that our approach supports the design of multirate systems with different periods for tasks and subgraphs. This includes harmonic task sets with $T_{\max} = H_\Gamma$, which are most common in real-world applications [Eis+10]. Additionally, it also supports non-harmonic task sets with $T_{\max} \neq H_\Gamma$ to a certain extend. For this purpose, we configure the length of the FlexRay communication cycle to the hyperperiod of the given task set, i.e. $\text{length}(\Theta_{\text{cycle}}) = H_\Gamma$ (cf. Section 4.2.5). As described in Section 2.1.2 and 2.2.2, a hyperperiod contains multiple instances of task τ_i generating multiple instances of a message m_i, i.e. $\tau_{i,j}, m_{i,j}$ with $1 \leq j \leq n, n = H_\Gamma / T_i$. Figure 4.16 depicts an exemplary multirate TDG composed of two subgraphs with different periods from [KHM03]. Similar to the TDG in Figure 4.13, it is extended by the coordinator tasks considering a hardware topology with three ECUs. The left subgraph has a period of $1500\mu s$ and the right one of $3000\mu s$. This means, that during the hyperperiod of $H_\Gamma = 3000\mu s$, $n = 3000\mu s / 1500\mu s = 2$ instances of tasks and messages for the left subgraph are executed and transmitted.

Figure 4.16 illustrates how our approach internally models and handles such multirate TDGs. The n instances of subgraphs respectively tasks with $T_i < H_\Gamma$

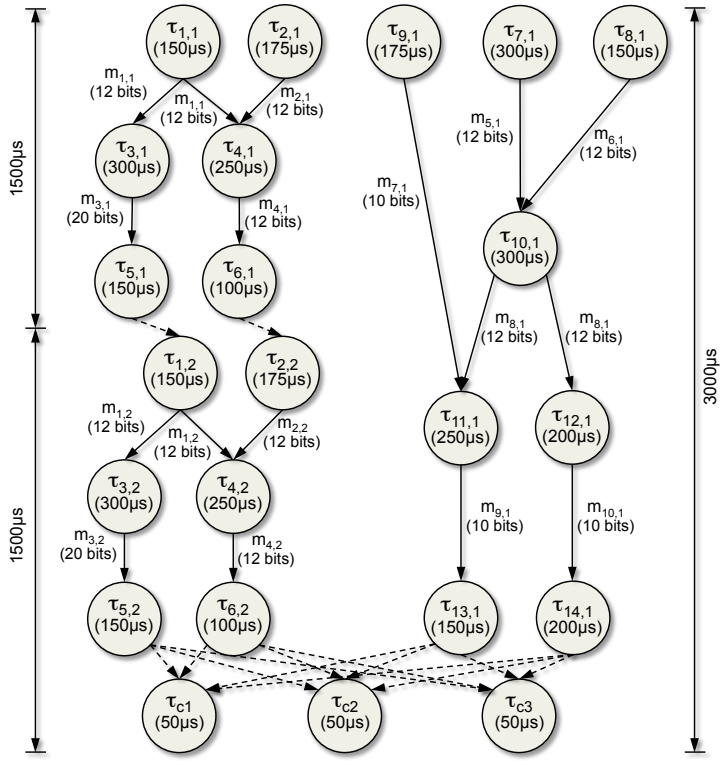


Figure 4.16: Example of an extended multirate TDG with multiple task and message instances based on TDGs from [KHM03].

are consecutively attached. The dashed arrows between the two instances of the left subgraph indicate their precedence relation but the corresponding tasks are still in Γ_{in} respectively Γ_{out}. Furthermore, the necessary coordinator tasks with the WCET $C_c = 50\mu s$ are attached to the last, i.e. n-th, instance of each subgraph. Thus, the coordinator tasks are scheduled once per hyperperiod of the given task set, i.e. $T_c = H_\Gamma$, which enables the DCC to detect a failure latest after the duration of one hyperperiod as mentioned in Section 4.2.2. Obviously, the available execution intervals $\Psi_{i,j} = [r_{i,j}, d_{i,j}[$ for a multirate system cannot be directly determined with Equation 4.24 and 4.26 because they do not distinguish between different task instances. Therefore, we extend these equations to calculate the individual instances release times $r_{i,j}$ as:

$$r_{i,j} = \begin{cases} (j-1) \cdot T_i & \text{, if } \tau_i \in \Gamma_{\text{in}} \\ \max\{r_{k,j} + C_k \,|\, \tau_k \in \Gamma_{\text{directPre}_i}\} & \text{, else,} \end{cases} \tag{4.28}$$

and the individual instances deadlines $d_{i,j}$ as:

$$d_{i,j} = \begin{cases} j \cdot \text{max end-to-end delay} - C_c & \text{, if } \tau_i \in \Gamma_{\text{out}} \\ \min\{d_{l,j} - C_l \,|\, \tau_l \in \Gamma_{\text{directSucc}_i}\} & \text{, else.} \end{cases} \tag{4.29}$$

Equation 4.28 and 4.29 calculate $r_{i,j}$ and $d_{i,j}$ for each instance j of a task τ_i by means of the given task periods respectively maximum end-to-end delay constraints. According to the definitions in Section 2.1, the release time of the j-th instance of a task $\tau_i \in \Gamma_{\text{in}}$ is $r_{i,1} = 0$ for the first instance, and $(j-1)$-times the task period T_i for further instances, i.e. $j > 1$. For the example in Figure 4.16 this results in $r_{1,1} = 0\mu s$ and $r_{1,2} = 1500\mu s$. Similar to Equation 4.28 for tasks with one or more predecessors, $r_{i,j}$ is calculated by means of the parameters $r_{k,j}$ and C_k of the j-th instances of the direct predecessors $\tau_k \in \Gamma_{\text{directPre}_i}$. Obviously, the deadline of the j-th instance of a task $\tau_i \in \Gamma_{\text{out}}$ is j-times the maximum allowed end-to-end value which corresponds to the task period T_i in our system architecture model specification (cf. Section 4.2.3). For tasks with one or more predecessors, $d_{i,j}$ is calculated by means of the parameters $d_{l,j}$ and C_l of the j-th instances of the direct successors $\tau_l \in \Gamma_{\text{directSucc}_i}$.

Table 4.3 summarizes the values of the available execution timer interval $\Psi_{i,j} = [r_{i,j}, d_{i,j}[$ for each task instance of the example multirate TDG in Figure 4.16. It shows how the individual release times of different task instances are calculated by means of their periods and precedence constraints. Moreover it shows, that even though the coordinator tasks are scheduled only once per hyperperiod, i.e. $T_c = H_\Gamma$, the deadlines $d_{i,j}$ of all task instances are shortened by the WCET $C_c = 50\mu s$ of the coordinator tasks as defined in Equation 4.29. This ensures that all different instances j and k of a task τ_i have the same relative deadline $D_i = d_{i,j} - r_{i,j} = d_{i,k} - r_{i,k}$ with $1 \leq j,k \leq n$, $n = H_\Gamma / T_i$ as defined in Section 2.1. For instance, in Table 4.3 both instances of task τ_1 have the same relative deadline $D_1 = 1000\mu s$. This equality of relative deadlines for all task instances is also necessary to perform the PDC respectively PDC* schedulability test that utilizes this parameter. Additionally, the consideration of multiple task and message instances may have an impact on the resulting communication delay δ_i we integrated in both schedulability tests. As described in Section 4.2.4 one part of the communication delay for an inter-ECU message $m_{(\tau_j,\tau_i)}$ is the transmission start delay $\zeta_{m_{(\tau_j,\tau_i)}}$ which grows if the next or more FlexRay slots after f_j are not available for the message transmission. Especially in a large system with many inter-ECU messages this may result in different transmission start delays for k different message instances from the sender task τ_j. In this case we have to consider the worst case transmission start delay for all message instances:

$$\zeta_{m_{(\tau_j,\tau_i)}} = \max\left\{\zeta_{m_{(\tau_j,\tau_i),k}} | 1 \leq k \leq n\right\},$$

to ensure a correct schedulability test for all instances of the receiver task τ_i.

Moreover, it is important to note that not all instances of different messages in a multirate system can have overlapping slots. For instance, in the multirate system from Figure 4.16 with $T_{m_1} = 1500\mu s$ and $T_{m_6} = 3000\mu s$ where the

Task (instance) $\tau_{i,j}$	Release Time $r_{i,j}[\mu s]$	Deadline $d_{i,j}[\mu s]$
$\tau_{1,1}$	0	1000
$\tau_{2,1}$	0	1100
$\tau_{3,1}$	150	1300
$\tau_{4,1}$	175	1350
$\tau_{5,1}$	450	1450
$\tau_{6,1}$	425	1450
$\tau_{7,1}$	0	2250
$\tau_{8,1}$	0	2250
$\tau_{9,1}$	0	2550
$\tau_{10,1}$	300	2550
$\tau_{11,1}$	600	2800
$\tau_{12,1}$	600	2750
$\tau_{13,1}$	850	2950
$\tau_{14,1}$	800	2950
$\tau_{1,2}$	1500	2500
$\tau_{2,2}$	1500	2600
$\tau_{3,2}$	1650	2800
$\tau_{4,2}$	1675	2850
$\tau_{5,2}$	1950	2950
$\tau_{6,2}$	1925	2950
τ_c	2100	3000

Table 4.3: Available execution time intervals $\Psi_{i,j}$ for multirate TDG in Figure 4.16.

available slots $\Theta_{m_{1,1}}$ and $\Theta_{m_{6,1}}$ may overlap, the available slots $\Theta_{m_{1,2}}$ cannot overlap with $\Theta_{m_{6,1}}$. Considering the set of all inter-node message instances $m_{i,j} \in \mathcal{M}^{\text{bus}}_{\text{ECU}_{\text{tx}}}$ which are sent by ECU_{tx} within the hyperperiod H_Γ, the determination of overlapping slots for messages can be formalized as:

$$\begin{aligned} \Omega_{\text{ECU}_{\text{tx}}} : \mathcal{P}\left(\mathcal{M}^{\text{bus}}_{\text{ECU}_{\text{tx}}}\right) \setminus \emptyset &\rightarrow \mathcal{P}(\Theta_{\text{cycle}}), \\ p &\mapsto \bigcap_{m_{i,j} \in p} \Theta_{m_{i,j}}. \end{aligned} \tag{4.30}$$

Equation 4.30 defines the function $\Omega_{\text{ECU}_{\text{tx}}}$ which assigns the power set[19] of all inter-node messages $\mathcal{M}^{\text{bus}}_{\text{ECU}_{\text{tx}}}$ sent by ECU_{tx} in one configuration to the power set of all slots in the FlexRay communication cycle. It maps each element p from the power set of message instances to the intersection of available slots for all message instances $m_{i,j} \in p$. By means of this function the overlapping slots for all possible message instance combinations in one configuration are determined. This information can be used to reduce the

[19] Without the empty set $\emptyset$.

number of slots assigned to ECU_{tx} by utilizing frame packing in overlapping available slot intervals.

Communication with Environment

The resulting reduction of required slots increases the flexibility for the determination of feasible slot assignments in large and complex systems with many necessary reconfigurations. Moreover, it helps to keep slots free for the additional bus communication which is required by our proposed reconfigurable distributed system topology presented. As mentioned in Section 4.2.1, the data required and provided by the set of functions of the distributed system has to be exchanged with the environment via sensors and actuators which are hardwired to dedicated peripheral interface nodes. Even though the actual setup and configuration of I/O and peripheral interfaces is beyond the scope of this thesis, here we briefly describe how to consider this communication in the system design approach.

Obviously, the tasks in Γ_{in} request input data and the tasks in Γ_{out} provide output data with their corresponding periods. On the one hand, the j-th instance of a task $\tau_{i,j} \in \Gamma_{\text{in}}$ needs the input at the beginning of its j-th period to start its execution. To allow a start time of $s_{i,j} = r_{i,j} = (j-1) \cdot T_i$ as derived in Equation 4.28, we consider a transmission of the necessary input data by the responsible peripheral interface node before the end of period $j-1$. On the other hand, the j-th instance of a task $\tau_{i,j} \in \Gamma_{\text{out}}$ may provide its output data until its deadline $d_{i,j} = j \cdot \max \text{ end-to-end delay} - C_c$ which is derived from the maximum end-to-end delay constraint, i.e. task period T_i, by Equation 4.29. Thus, it is reasonable to transmit the output data $\tau_{i,j} \in \Gamma_{\text{out}}$ to the peripheral interface nodes in the beginning of the following period $j+1$. Therefore, depending on the number of peripheral interface nodes and the resulting number and size of messages for I/O data exchange, the required number of slots at the beginning and the end of the periods can be blocked to assign them to these messages and their sender nodes.

In the example of Figure 4.16 the tasks $\tau_1, \tau_2 \in \Gamma_{\text{in}}$ require new input data from the I/O interfaces every $1500\mu s$ and the tasks $\tau_5, \tau_6 \in \Gamma_{\text{out}}$ provide output data for the peripheral nodes with the same rate. The tasks $\tau_7, \tau_8, \tau_9 \in \Gamma_{\text{in}}$, as well as $\tau_{13}, \tau_{14} \in \Gamma_{\text{out}}$ must exchange I/O data once per hyperperiod of the TDG, i.e. $H_\Gamma = T_{\max} = 3000\mu s$. As described in Section 4.2.5 we set the FlexRay communication cycle length for this example to $\text{length}(\Theta_{\text{cycle}}) = T_{\max} = 3000\mu s$. Thus, the necessary slots at the beginning of the FlexRay cycle are blocked for the messages from all tasks in Γ_{out} respectively their hosting ECUs. Beside slot reuse also frame packing can be utilized for messages from tasks executed on the same ECU to reduce the number of blocked slots. However, at the beginning of the FlexRay cycle the tasks in Γ_{in} will

be executed. Therefore, the transmission of data to the peripheral interfaces can be performed at least partly during the execution of these tasks in slots which are not applicable for functional node communication. Similar conditions hold for the end of the FlexRay cycle. Here, the necessary slots are blocked for the messages from the peripheral interfaces to the tasks in Γ_{in}. Slot reuse and frame packing are also applicable for these messages. For instance, if several tasks need the same input data or one peripheral interface node is hardwired to different sensors. As mentioned above, the message transmission must be performed at the end of the FlexRay cycle where the tasks in Γ_{out} and the coordinator tasks or even no more tasks are executed. Thus, message transmissions from the peripheral interfaces can be performed at least partly during or after the execution of these tasks without blocking slots which are required for functional node communication.

Obviously, subgraphs respectively their tasks τ_i in Γ_{in} and Γ_{out} with a period of $T_i < H_\Gamma$ must exchange data with the peripheral interfaces multiple times within the hyperperiod of the TDG. For our example, additional slots for the data exchange between the left subgragh and the peripheral interfaces have to be blocked around its period of $1500\mu s$. Consequently, the slots which are assigned to the interface nodes have to be blocked and are not available for the functional node communication. The slots assigned to the ECUs hosting the tasks $\tau_5, \tau_6 \in \Gamma_{\text{out}}$ may only be partly used for functional node communication if frame packing is possible. Since the resulting pre-assignment of slots to ECUs has to be considered by our design approach, the COMG input is extended by this required information. Listing 4.8 depicts as an example how the COMG input from Listing 4.7 is extended by information about blocked slots. In this example the first three and the last three slots are pre-assigned for communication with peripheral interfaces.

```
1  <COMG>
2    <cycleLength> 1000 </cycleLength>
3    <slotLength> 25 </slotLength>
4    <slotsNIT> 1 <slotsNIT>
5    <blockedSlotsIN>
6      <slot ID="S_37" />
7      <slot ID="S_38" />
8      <slot ID="S_39" />
9    </blockedSlotsIN>
10   <blockedSlotsOUT>
11     <slot ID="S_01" />
12     <slot ID="S_02" />
13     <slot ID="S_03" />
14   </blockedSlotsOUT>
15 </COMG>
```

Listing 4.8: COMG input from Listing 4.7 extended by information about pre-assigned slots.

In Section 4.2.3 and 4.3.1 we described the graph and XML-based representation of the TDG as input for our design approach. The TDG defines precedence constraints and is annotated with timing properties, i.e. WCETs, of the given tasks. Furthermore, in Section 4.2.3 we also presented TADL2 timing constraints for a given TDG. These information are also part of the TDG description as input for our approach. The resulting properties of the individual tasks in the TDG, i.e. realease time and deadline, to fulfill the given precedence and timing constraints are calculated by means of Equation 4.24 and 4.25. Obviously, the timing constraints as well as the task and communication properties have to be considered for the actual system deployment we present in the next section.

4.4 Deployment

This section describes the system deployment for fault-tolerant distributed real-time systems as essential part of our design approach. As illustrated in the approach overview in Figure 4.1, an appropriate deployment comprises the determination of feasible task and bus mappings for the initial configuration and based on that for all necessary reconfigurations to design a fault-tolerant system. Therefore, the following sections 4.4.1 and 4.4.2 describe our task and bus mapping concepts and algorithms. Additionally, for a better traceability all deployment steps are applied to the example from [KMR12] that we introduced above.

4.4.1 Task Mapping

To determine feasible overall task mappings, we have to ensure that all functions in the distributed real-time system fulfill their timing constraints and all tasks meet their deadlines. In Section 4.2.4 we introduced the PDC* and RTA* schedulability tests, which combine tasks' WCETs and possible delays resulting from inter-ECU communication. Depending on the applied local task scheduling strategy the corresponding schedulability test has to be fulfilled to guarantee a feasible schedule for each ECU resulting in a valid task scheduling for the complete distributed system. Therefore, the check of the schedulability test is the necessary condition for each task assignment. Before a task can be mapped to an ECU it has to be guaranteed that the new task and all other tasks on this ECU will still meet their deadlines after the assignment. Hence, to guarantee a feasible schedule the scheduling analysis is performed for each task of the TDG before mapping, beginning with the tasks in Γ_{in}. By means of this analysis the suitability of an ECU is checked. Based on that test, our approach analyzes and reduces the resulting execution delay for each task-to-ECU mapping to finally ensure minimized end-to-end delays for all event chains from Γ_{in} to Γ_{out}.

As mentioned in Section 4.2 our design approach it is not intended to optimize one specific single attribute of the system, e.g. the task response time. However, it is reasonable to enable parallel execution of tasks with similar available execution time intervals to reduce their response times. As described in Section 4.3.2 the dependencies between the tasks in a given TDG result in an available execution time interval $\Psi_i = [r_i, d_i[$ for each task τ_i. To increase the flexibility within the scheduling and the efficiency of the utilization of the available resources, it is useful to start the execution of a task τ_i as soon as possible within its available execution time interval. Therefore, we propose the mapping of tasks with similar available execution time intervals to different ECUs if possible to enable an at least partly parallel execution of these tasks. In the following we describe in detail how the task mappings within our design approach are performed to realize this.

In the example of Figure 4.13 and Table 4.2 the tasks τ_4 and τ_5 have the similar available execution time intervals $\Psi_4 = [100\mu s, 750\mu s[$ and $\Psi_5 = [200\mu s, 850\mu s[$. This means that τ_4 may start its execution earliest at $s_4 = r_4 = 100\mu s$, depending on the mapping of its predecessors and on the other tasks executed on ECU_{τ_4}. If τ_5 is also mapped to ECU_{τ_4} it may start earliest at $s_5 = f_4 = s_4 + C_4 = 100\mu s + 300\mu s = 400\mu s$ instead of $r_5 = 200\mu s$. The earliest possible finishing time of τ_5 would be $f_5 = s_5 + C_5 = 400\mu s + 300\mu s = 700\mu s$. If the tasks would be mapped to different ECUs, i.e. $\text{ECU}_{\tau_4} \neq \text{ECU}_{\tau_5}$, τ_5 may start earliest at $s_5 = r_5 = 200\mu s$ and therefore be partly executed parallel to τ_4 to be finished earliest at $f_5 = s_5 + C_5 = 200\mu s + 300\mu s = 500\mu s$. Summarized, by means of this approach the response time can be reduced for at least partly in parallel executed tasks. This implies that each task waiting for input from a task τ_i or other tasks scheduled after τ_i may start earlier. As mentioned above, our approach performs the mapping which minimizes the value for the resulting maximum of all end-to-end delays for all event chains from Γ_{in} to Γ_{out}.

In Section 4.2.4 we described that the performed task mapping may have a great impact on the resulting communication overhead. Therefore, our approach is also intended to reduce this overhead regarding different aspects. Especially in complex distributed systems with a high number of connected tasks exchanging messages it is desirable to reduce the number of assigned FlexRay slots per ECU to increase the flexibility for the determination of a feasible system design and scheduling. Therefore, our mapping approach aims to enable message transmission via intra-ECU communication instead of inter-ECU communication if possible. For example, if several tasks executed on different ECUs finishing at similar points in time would need access to the FlexRay bus it could become quite complex or in worst case impossible to determine a feasible bus scheduling for the messages. This risk can be avoided by reducing the number of needed slots.[20] Thus, our approach

[20] Necessarily, the reduction of inter-ECU communication implies an increased number of intra-ECU messages.

firstly considers to map a task τ_i to the same ECU as one or more of its direct predecessors in $\Gamma_{\text{directPre}_i}$ to enable intra-ECU communication. If τ_i has more than one direct predecessors, our approach analyses their hosting ECUs to determine the best candidate for the mapping of τ_i.

However, our approach also aims to reduce communication delays and response times. Hence, the at least partly parallel execution of tasks with similar available execution time intervals is used to increase the flexibility of the scheduling. Our approach implicitly checks if one or more of these ECUs are not assigned to any task with a similar available execution time interval. In this case, if the schedulability test is positive, τ_i can be mapped to ECU_{τ_j} with $\tau_j \in \Gamma_{\text{directPre}_i}$ resulting in a reduced communication delay and response time for τ_i. For this purpose, our approach identifies the last scheduled tasks τ_{last} on each ECU_{τ_j} hosting direct predecessors and compares their finishing times f_{last}. If one or more of the direct predecessors of τ_i are last tasks, the approach maps τ_i to the same ECU as the predecessor with the latest finishing time. This results in the shortest possible delay, because it avoids additional inter-ECU communication for the latest message input of τ_i. If no direct predecessor is a last scheduled task, the approach maps τ_i to the ECU with the earliest finishing time and shortest communication delay. This results in a reduction or total avoidance of the communication delay δ_i because the inter-ECU message transmission can at least partly be performed during the execution of the tasks scheduled after the direct predecessors (cf. Section 4.2.4). Summarized, this mapping approach which results in a reduced communication delay is not conflicting with the aim to reduce inter-ECU messages because it avoids unnecessary inter-ECU messages.

Our approach starts the task mapping on Γ_{in} with the assignment to different ECUs, if possible. Hence, the assignment of tasks in Γ_{mid} and Γ_{out} to the same ECU as their direct predecessors implicitly realizes the mapping of tasks with similar available execution time intervals to different ECUs resulting in an at least partly parallel execution of these tasks and reduced inter-ECU communication as mentioned above.

Inital Mapping

The task mapping approach is implemented with several algorithms. It starts with the mapping for the initial configuration which is represented by Algorithm 1 (INITIALTASKMAPPING) in pseudocode. The algorithm gets the TDG $G_{\text{TDG}} = (\Gamma, \mathcal{M})$, HAG $G_{\text{HAG}} = (\mathcal{E}, E)$, and COMG $G_{\text{COMG}} = (\Theta, \emptyset)$ as input. The COMG has to be considered because it contains information about the available slots in Θ which the algorithm has to determine for the transmission start delays of input messages (cf. Section 4.3.2).

Algorithm 1 INITIALTASKMAPPING

Input: TDG $G_{\text{TDG}} = (\Gamma, \mathcal{M})$, $G_{\text{HAG}} = (\mathcal{E}, E)$, and $G_{\text{COMG}} = (\Theta, \emptyset)$.
Output: TMG $G_{\text{TMG}} = (\mathcal{E}, \mathcal{M})$ with TaskMapping $\text{M}^{\tau}_{\text{init}} : \Gamma \mapsto \mathcal{E}$.

```
1: SORTBYDEADLINEANDRELEASETIME(Γ)
2: for all τ_i ∈ Γ_in do
3:    E_tmp ← GETFEASIBLEECUS(τ_i, E)
4:    ECU_tmp ← GETEARLIESTFINISH(E_tmp)
5:    MAPTASK(τ_i, ECU_tmp)
6:    UPDATESTARTANDFINISHTIMES(Γ)
7: end for
8: for all τ_i ∈ Γ \ Γ_in do
9:    E_tmp ← GETFEASIBLEECUS(τ_i, E, Θ)
10:   ECU_tmp ← GETMINIMUMTASKDELAY(τ_i, E_tmp, Θ)
11:   MAPTASK(τ_i, ECU_tmp)
12:   UPDATESTARTANDFINISHTIMES(Γ)
13:   UPDATEAVAILABLESLOTS(Θ)
14: end for
15: return TMG G_TMG = (E, M) with TaskMapping M^τ_init : Γ ↦ E
```

Before the actual mapping is performed, it defines the mapping order of the given task set Γ via sorting its tasks by deadline and release time. The algorithm starts the mapping with the tasks in Γ_{in}. In the beginning of each iteration it checks the given ECU set $\mathcal{E}$ for feasible candidates and determines the candidate set $\mathcal{E}_{\text{tmp}}$. This means, that the schedulability test PDC* respectively RTA* is perfomed for each ECU including the additional task τ_i to map.[21] To increase the execution parallelism of tasks, in each iteration the algorithm identifies the $\text{ECU}_{\text{tmp}} \in \mathcal{E}_{\text{tmp}}$ whose last executed task τ_{last} has the earliest finishing time f_{last} and maps τ_i to this ECU_{tmp}. By doing so, our approach ensures that for n available ECUs the first n tasks in Γ_{in} are mapped to empty ECUs with $f_{\text{last}} = 0$. Since, the tasks are ordered by deadlines, this guarantees that the n tasks with the earliest deadlines are mapped to different ECUs and executed at least partly in parallel. By selecting the ECU in each iteration as described above, the algorithm ensures that tasks with earlier deadlines are mapped to ECUs hosting last tasks with earliest finishing times. This improves the mapping of each task $\tau_i \in \Gamma_{\text{in}}$ and reduces its response time R_i. After each mapping, the start time s_i and finishing time f_i of each task are updated according to the resulting current schedule. Since the tasks in Γ_{in} do not have input messages implying possible communication delays the algorithm does not need to consider communication delays for these tasks.

[21] To handle multirate systems and because of the fact that there are commonly more tasks in Γ_{in} than ECUs available, the algorithm has to perform the schedulability test already for the tasks in Γ_{in} Moreover, both schedulability tests assume the critical instant of the task set Γ, i.e. synchronous activation of tasks (cf. Section 2.1.2). Therefore, the tests consider the first instance of all tasks in Γ to be activated at $r_{i,1} = 0$ instead of the release times calculated by Equation 4.28.

When all tasks in Γ_{in} are mapped the algorithm continues with the tasks in Γ_{mid} and Γ_{out}, i.e. $\Gamma \setminus \Gamma_{in}$, which have predecessors. Here, in the beginning of each iteration it also checks the given ECU set $\mathcal{E}$ for feasible candidates and determines the candidate set $\mathcal{E}_{tmp}$. Now, for each task τ_i the communication delay δ_i has to be integrated in the schedulability test and the appropriate task-to-ECU mapping is more complex. Therefore, Algorithm 2 (GETMINIMUMTASKDELAY) is utilized to determine the $ECU_{tmp} \in \mathcal{E}_{tmp}$ with the minimum execution delay respectively response time R_i. Finally, Algorithm 1 iteratively maps each task τ_i to the corresponding ECU_{tmp} and updates the start times s_i and finishing times f_i, as well as the available slots in Θ for all tasks according to the resulting current schedule.

As mentioned above the determination of the best assignable ECU for tasks with predecessors is performed by Algorithm 2. It gets the task τ_i to map, the set $\mathcal{E}_{tmp}$ of feasible candidate ECUs, and the set Θ of available slots as input. The algorithm determines the set $\mathcal{E}_{pre} \subseteq \mathcal{E}_{tmp}$ hosting the direct predecessors $\Gamma_{directPre}$ of τ_i. The set Γ_{last} of tasks τ_{last} last executed on these ECUs is identified. Hence, the intersection $\Gamma_{cap} = \Gamma_{last} \cap \Gamma_{directPre}$ represents the direct predecessors of τ_i which are scheduled and executed at last on their corresponding hosting ECUs. This means, that if one or more of the direct predecessors of τ_i are last tasks, the set Γ_{cap} will also contain one or more elements, i.e. $\Gamma_{cap} \neq \emptyset$. If this is the case, Algorithm 2 returns the ECU hosting the direct predecessor with the latest finishing time for the mapping of τ_i. Since τ_i can be executed earliest after the reception of the last input message, this mapping results in the shortest possible execution delay for τ_i because it avoids additional inter-ECU communication delay for the latest input message.

If there are tasks mapped to all ECUs which are executed after the direct predecessors, the necessary inter-ECU communication for the input messages of τ_i can be performed at least partly during their execution. In this case the candidate ECUs are compared regarding their finishing time and the resulting communication delay δ_i for task τ_i. Out of the set $\mathcal{E}_{pre}$ the algorithm determines the ECU_{preMin} which is hosting one or more direct predecessors with the earliest finishing time and shortest communication delay. Obviously, the resulting communication delay of an ECU depends on the properties and availability of the FlexRay slots for the necessary inter-ECU messages and our approach considers the first available slot in Θ. If there are no ECUs in the candidate set $\mathcal{E}_{tmp}$ which are not hosting any direct predecessor, i.e. $\mathcal{E}_{tmp} \setminus \mathcal{E}_{pre} = \emptyset$, Algorithm 2 returns ECU_{preMin} to Algorithm 1. If there are ECUs in $\mathcal{E}$ which do not host any direct predecessor, the algorithm also considers these candidates for the mapping of τ_i. This is reasonable because, even though a mapping to an ECU in $\mathcal{E}_{pre}$ reduces the inter-ECU communication overhead, it is possible that a mapping to an ECU in $\mathcal{E} \setminus \mathcal{E}_{pre}$ results in a lower execution delay respectively response time of τ_i and our approach is

Algorithm 2 GETMINIMUMTASKDELAY

Input: Task τ_i, set of candidate ECUs $\mathcal{E}_{\text{tmp}}$, and available slots Θ.
Output: ECU with minimum execution delay.

```
1: Γ_directPre ← GETDIRECTPREDECESSORS(τ_i)
2: E_pre ← GETHOSTECUS(Γ_directPre, E_tmp)
3: Γ_last ← GETLASTTASKS(E_pre)
4: Γ_cap ← Γ_last ∩ Γ_directPre
5: if Γ_cap ≠ ∅ then
6:    ECU ← GETHOSTECU(GETLATESTFINISH(Γ_cap, E_tmp))
7: else
8:    ECU_preMin ← GETEARLIESTFINISHANDSHORTESTCOMDELAY(E_pre, Θ)
9:    if E_tmp \ E_pre ≠ ∅ then
10:      ECU_nonPreMin ← GETEARLIESTFINISHANDSHORTESTCOMDELAY(E_tmp \ E_pre, Θ)
11:      Δ ← DIFF⟨GETDELAY(ECU_preMin, Θ), GETDELAY(ECU_nonPreMin, Θ)⟩
12:      if Δ > 0 then
13:         ECU ← ECU_nonPreMin
14:      else
15:         ECU ← ECU_preMin
16:      end if
17:   else
18:      ECU ← ECU_preMin
19:   end if
20: end if
21: return ECU
```

designed to reduce the resulting execution delay for each task-to-ECU mapping. Therefore, the algorithm determines the $\text{ECU}_{\text{nonPreMin}}$ in $\mathcal{E}_{\text{tmp}} \setminus \mathcal{E}_{\text{pre}}$ with the earliest finishing time and shortest communication delay, too. Consequently, $\text{ECU}_{\text{preMin}}$ and $\text{ECU}_{\text{nonPreMin}}$ are the two remaining candidates for the mapping of τ_i.

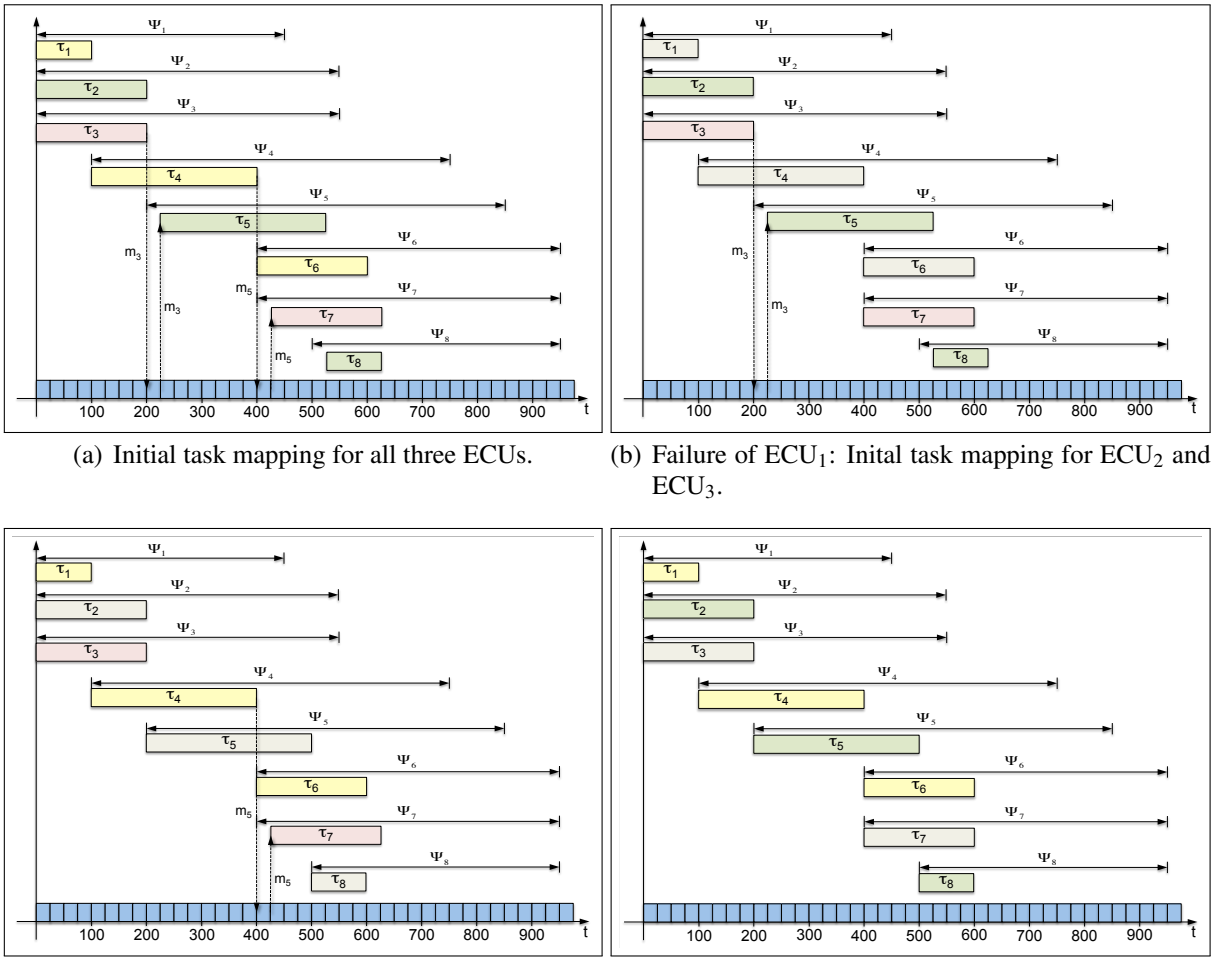
These ECUs are compared by calculating the difference Δ of the resulting execution delays. If the delay is longer for $\text{ECU}_{\text{preMin}}$, i.e. $\Delta > 0$, the algorithm returns $\text{ECU}_{\text{nonPreMin}}$. If the delay is equal or shorter for $\text{ECU}_{\text{preMin}}$ Algorithm 2 returns this ECU to map τ_i to a ECU hosting one or more direct predecessors avoiding additional inter-ECU communication overhead.[22] Furthermore, it is also possible that none of the ECUs hosting the direct predecessors of τ_i fulfills the schedulability test. In this case, the input set $\mathcal{E}_{\text{tmp}}$ of candidate ECUs passed to Algorithm 2 does not contain any ECU hosting a direct predecessor and $\mathcal{E}_{\text{pre}}$ becomes the empty set. Thus, Γ_{last} and Γ_{cap} become empty sets, too. Since $\mathcal{E}_{\text{pre}}$ does not contain any element,

[22] Here it is possible to further prioritize the avoidance of inter-ECU communication overhead by setting the corresponding threshold for the difference Δ. For instance, a high threshold, i.e. $\Delta >> 0$ implies that a task τ_i is only mapped to an ECU in $\mathcal{E} \setminus \mathcal{E}_{\text{pre}}$ if this would result in a much shorter execution delay.

ECU_{preMin} is also not set. The set $\mathcal{E}_{tmp} \setminus \mathcal{E}_{pre}$ contains all possible candidate ECUs for the mapping of τ_i. Out of this set, the algorithm determines the $ECU_{nonPreMin}$ with the earliest finishing time and shortest communication delay in $\mathcal{E}_{tmp} \setminus \mathcal{E}_{pre}$, too. The determination of the resulting execution delay for the not set element ECU_{preMin} returns the value infinity which ensures that $\Delta > 0$ always holds. Consequently, Algorithm 2 finally returns $ECU_{nonPreMin}$.

By means of Algorithm 1 and 2 our approach tries to determine a feasible initial task mapping with shortened execution delays respectively response times considering timing, order, and precedence constraints. Obviously, it is also possible that no feasible solution can be found for the given inputs. However, the algorithms always terminate, either returning a feasible initial mapping after iterating over all tasks or with a negative result in M^{τ}_{init}. The worst case time complexity, i.e. the worst case running time of an algorithm, can be expressed by the O-notation. Thus, the O-notation is used as an asymptotic upper bound for the time complexity of an algorithm on arbitrary inputs [CLR90]. Algorithm 1 iterates over all tasks in Γ. In each iteration it analyzes all ECUs in $\mathcal{E}$ updates the properties of all tasks and the available slots. This results in a time complexity of $O(|\Gamma| \cdot (|\mathcal{E}| + |\Gamma| + |\Theta|))$. However, in each iteration also Algorithm 2 is executed (cf. line 10). This algorithm analyzes a set of ECUs and determines the intersection of two task sets in line 3 and 4 which results in a worst case running time of $O(|\Gamma|^2 + |\mathcal{E}|)$. Furthermore, it analyzes the given set of slots for each ECU regarding their resulting finishing times and communication delays in the lines 8 and 10. This operation has a worst case running time of $O(|\Theta| \cdot |\mathcal{E}|)$. Hence, each invocation of Algorithm 2 has an entire running time of $O(|\Gamma|^2 + |\mathcal{E}| \cdot |\Theta|)$. Considering this, the resulting upper bound of the time complexity for Algorithm 1 is $O(|\Gamma|^3 + |\Gamma| \cdot |\mathcal{E}| \cdot |\Theta|)$.

Figure 4.17 illustrates Gantt Charts which represent the local scheduling and task executions on the ECUs resulting from initial task mapping performed on input TDG shown in Figure 4.13 and HAG from Figure 4.12(b). For a better traceability, all tasks are colored as their hosting ECU in the HAG. All Gantt Charts depict the available execution time interval Ψ_i for each task τ_i and the FlexRay cycle with the configured size and number of slots. Figure 4.17(a) shows the resulting schedule for the initial mapping on all three available ECUs, i.e. the task mapping for the initial configuration. It illustrates that the three tasks $\tau_1, \tau_2, \tau_3 \in \Gamma_{in}$ can be started parallel without any execution delay. The task τ_5 which has the two direct predecessors τ_2 mapped to ECU_2 and τ_3 mapped to ECU_3 is also mapped to ECU_2. Thus, its execution is delayed by a communication delay of $\delta_5 = 25\mu s$ considering the first available slot for the transmission of the inter-ECU message m_3. The Figure also shows, that τ_7 is not mapped to ECU_1 as its direct predecessor τ_4 to reduce the resulting response time R_7. Since Algorithm 1 already mapped τ_6 to ECU_1, this would result in an execution delay of $s_7 = f_6 = f_4 + C_6 = f_4 + 200\mu s$.



(a) Initial task mapping for all three ECUs.

(b) Failure of ECU_1: Inital task mapping for ECU_2 and ECU_3.

(c) Failure of ECU_2: Inital task mapping for ECU_1 and ECU_3.

(d) Failure of ECU_3: Inital task mapping for ECU_1 and ECU_2.

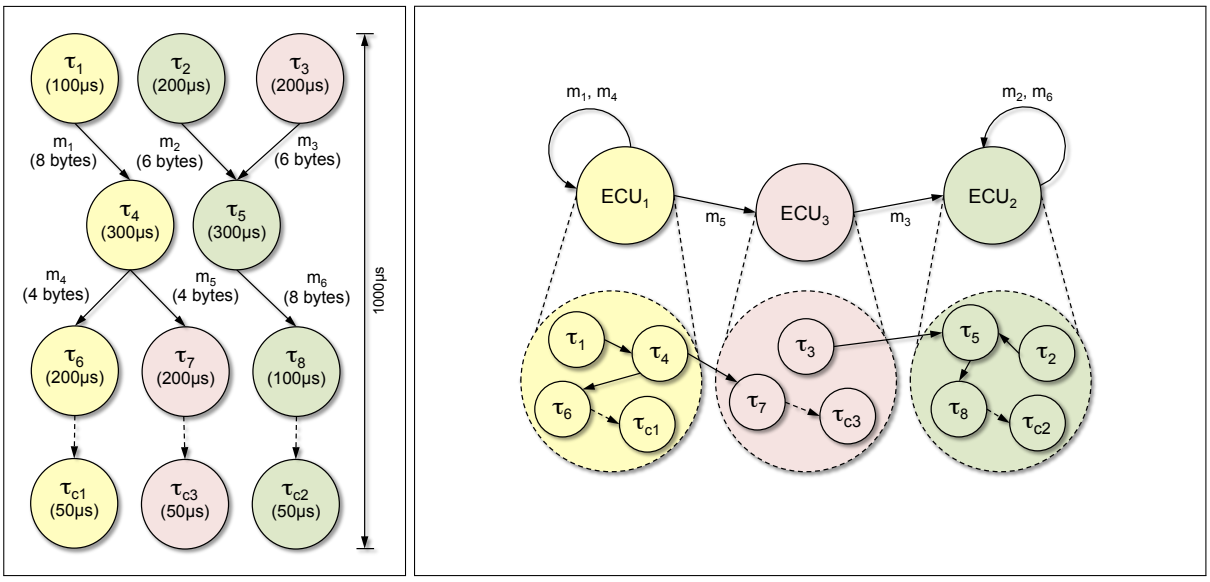
Figure 4.17: Gantt Charts for local schedulings and task executions on ECUs resulting from initial task mapping performed on input graphs shown in Figure 4.12(b) and 4.13.

Therefore, Algorithm 2 returns ECU_3 with the earliest finishing time and shortest communication delay for the inter-ECU message m_5, resulting in a much shorter execution delay with $s_7 = f_4 + \delta_7 = f_4 + 25\mu s$.

The considered example solely contains a TDG and tasks with the same period, i.e. it is not representing a multirate system. Nevertheless, the task mapping algorithm is also applicable for multirate systems. As described in Section 4.3.2, n instances of a task τ_i with a period $T_i = H_\Gamma/n$ are executed within the hyperperiod of the given multirate task set. This means that for a mapping of a task in a multirate system our approach has to analyze and reduce the resulting execution delays for multiple task instances to meet their deadlines. The determination of feasible candidate ECUs based on the schedulability tests PDC* and RTA* already considers multiple instances with different periods and communication delays to guarantee the fulfillment

of the timing constraints. However, the execution delays respectively response times of multiple task instances may vary. Especially, for the first instances of the tasks in Γ_{in} with $r_{i,1} = 0$, the critical instant criterion holds if more than one task is scheduled on the same ECU, i.e. a task instance has the longest response time if it is released simultaneously with all higher priority tasks (cf. Section 2.1.1).

Table 4.3 provides the corresponding values for the multirate TDG with two subgraphs in Figure 4.16. It shows that the first instances of all tasks in Γ_{in} are released at $r_{i,1} = 0$, i.e. at the beginning of the hyperperiod. Considering less than five ECUs at least one of these tasks will be delayed. Since the successors of delayed tasks are also delayed this results in an increased end-to-end delay. Therefore, the algorithm identifies the critical task instances producing the largest response times and maps the task to the corresponding ECU to reduce their execution delays and the resulting end-to-end delays. Consequently, if the algorithm reduces the end-to end delays of these critical task instances the end-to-end delays of the other instances are even shorter. The critical task instance of each task τ_i is determined and considered in the schedulability tests for feasible ECUs in the lines 3 and 9 of Algorithm 1 and passed to Algorithm 2 in line 10.



(a) Task mapping of initial configuration.

(b) TMG for initial configuration.

Figure 4.18: Task mapping (a) and resulting TMG (b) for input graphs shown in Figure 4.12(b) and 4.13.

As a final result, after all task mappings are performed, Algorithm 1 returns the task-to-ECU mapping M^{τ}_{init} of the initial configuration as a *task mapping graph* (TMG). A TMG $G_{\text{TMG}} = (V, E) = (\mathcal{E}, \mathcal{M})$ ia an abstracted and combined representation of the task mappings and directly provides the resulting

inter-ECU communication. Therefore, it is used as internal representation for our approach and as input for the bus mapping described in Section 4.4.2. Figure 4.18 illustrates the performed task mapping and the resulting TMG of the initial configuration for the input graphs shown in Figure 4.12(b) and 4.13. For a better traceability, in Figure 4.18(a), all tasks of the input TDG are colored as their hosting ECU and Figure 4.18(b) depicts the output TMG of Algorithm 1. The TMG shows that τ_1, τ_4, τ_6 are initially mapped to ECU_1 while ECU_2 executes τ_2, τ_5, τ_8 and τ_3, τ_7 are assigned to ECU_3. This implies the intra-ECU messages m_1, m_4 on ECU_1 and m_2, m_6 on ECU_3. The messages m_3 and m_5 are inter-ECU messages and have to be mapped and scheduled on the bus according to their timing constraints.

Redundancy Mapping

As described in Section 4.2 the main objective of our approach is the determination of a fault-tolerant system design with the compensation of one arbitrary component fault by means of efficient redundancy. To reduce the required hardware redundancy it utilizes the Cold Standby approach and maps inactive task replicas to the ECUs of the reconfigurable distributed system topology. Here, each reconfiguration respectively redundancy mapping is based on the initial mapping performed by Algorithm 1. This means that the initial mappings for the remaining ECUs are kept unchanged and just the replicas for the task set initially mapped to the ECU considered to be failed are mapped to the other nodes. However, for a network topology with n ECUs also n redundancy task mappings have to be determined to be able to compensate each possible node failure. As well as the initial mapping the required redundancy mappings are determined already at the design phase. Hence, during runtime the self-reconfiguration of the system activates the corresponding pre-determined reconfiguration to compensate a detected ECU failure. As described in Section 4.2.2 this ensures a short and deterministic time interval required for a self-reconfiguration.

Algorithm 3 (REDUNDANCYTASKMAPPING) represents in pseudocode how our design approach performs the corresponding redundancy mappings. For each mapping the algorithm gets the set of remaining ECUs $\mathcal{E}_{\text{rem}}$, the TMG $G_{\text{TMG}} = (\mathcal{E}, \mathcal{M})$ of the inital mapping M^{τ}_{init}, and the COMG $G_{\text{COMG}} = (\Theta, \emptyset)$ as input. Before the redundancy mapping is performed by means of the inital mapping M^{τ}_{init} and the remaining ECUs, the algorithm determines the task set Γ_{fail} which was executed on the failed ECU. Afterwards, it initializes the redundancy mapping M^{τ}_{red} with the task mapping of the remaining ECUs kept from M^{τ}_{init}. Figure 4.17(b) to 4.17(d) depict the resulting Gantt Charts of the task mappings for the remaining ECUs based on the initial mapping in Figure 4.17(a). For instance, Figure 4.17(b) illustrates the initial task mapping after a failure of ECU_1. It shows, that in this case the grey colored tasks $\tau_1, \tau_4, \tau_6 \in$

Algorithm 3 REDUNDANCYTASKMAPPING

Input: Remaining ECUs $\mathcal{E}_{\text{rem}}$, TMG $G_{\text{TMG}} = (\mathcal{E}, \mathcal{M})$ of $\text{M}^{\tau}_{\text{init}}$, and $G_{\text{COMG}} = (\Theta, \emptyset)$.
Output: TMG $G_{\text{TMG}} = (\mathcal{E}, \mathcal{M})$ with RedundancyTaskMapping $\text{M}^{\tau}_{\text{red}} : \Gamma \mapsto \mathcal{E}$.

1: $\Gamma_{\text{fail}} \leftarrow$ GETFAILEDECUTASKSET($\text{M}^{\tau}_{\text{init}}, \mathcal{E}_{\text{rem}}$)
2: $\text{M}^{\tau}_{\text{red}} \leftarrow$ GETMAPPINGFORREMAININGECUS($\text{M}^{\tau}_{\text{init}}, \mathcal{E}_{\text{rem}}$)
3: **for all** $\tau_i \in \Gamma_{\text{fail}}$ **do**
4: $\quad \mathcal{E}_{\text{tmp}} \leftarrow$ GETFEASIBLEECUS($\tau_i, \mathcal{E}_{\text{rem}}$)
5: $\quad \text{ECU}_{\text{tmp}} \leftarrow$ GETECUMINOVERALLE2E(τ_i, $\mathcal{E}_{\text{tmp}}, \text{M}^{\tau}_{\text{red}}$)
6: $\quad \text{M}^{\tau}_{\text{red}} \leftarrow \text{M}^{\tau}_{\text{red}} \cup$ MAPTASK(τ_i, ECU_{tmp})
7: $\quad$ UPDATESTARTANDFINISHTIMES(Γ)
8: $\quad$ UPDATEAVAILABLESLOTS(Θ)
9: **end for**
10: **return** TMG $G_{\text{TMG}} = (\mathcal{E}, \mathcal{M})$ with RedundancyTaskMapping $\text{M}^{\tau}_{\text{red}} : \Gamma \mapsto \mathcal{E}$

Γ_{fail} have to be mapped to the remaining ECUs. It also shows that τ_2, τ_5, τ_8 remain mapped on ECU_2 and τ_3, τ_7 on ECU_3. This implies that m_3 remains an inter-ECU message from ECU_3 to ECU_2 as shown in the inital mapping TMG in Figure 4.18(b). Similar conditions hold for the other possible ECU failures shown in Figure 4.17.

Based on the corresponding input, Algorithm 3 iteratively maps the tasks in Γ_{fail} to the remaining ECUs. Hence, in each mapping step the redundancy mapping $\text{M}^{\tau}_{\text{red}}$ is complemented by the currently performed task mapping. In the beginning of each iteration it checks the given set $\mathcal{E}_{\text{rem}}$ for feasible candidates $\mathcal{E}_{\text{tmp}}$ by means of the corresponding schedulability test. Out of these candidates, Algorithm 4 (GETECUMINOVERALLE2E) determines and returns the ECU resulting in the minimum overall end-to-end delay for a assignment of task τ_i. Finally, Algorithm 3 iteratively complements $\text{M}^{\tau}_{\text{red}}$ by mapping τ_i to the corresponding ECU_{tmp} and updates the start times s_i and finishing times f_i, as well as the available slots in Θ for all tasks according to the resulting current schedule.

Algorithm 4 GETECUMINOVERALLE2E

Input: Task τ_i, set of candidate ECUs $\mathcal{E}_{\text{tmp}}$, available slots Θ, and current mapping $\text{M}^{\tau}_{\text{cur}}$.
Output: ECU resulting in minimum overall end-to-end (E2E) delay.

1: **for all** $\text{ECU}_i \in \mathcal{E}_{\text{tmp}}$ **do**
2: $\quad \text{M}^{\tau}_i \leftarrow \text{M}^{\tau}_{\text{cur}} \cup$ MAPTASK(τ_i, ECU_i)
3: $\quad \text{E2E}_{\text{ECU}_i} \leftarrow$ GETOVERALLE2EDELAYANDCOMOVERHEAD($\text{M}^{\tau}_i, \Theta$)
4: $\quad \text{E2E} \leftarrow \text{E2E} \cup \text{E2E}_{\text{ECU}_i}$
5: **end for**
6: ECU $\leftarrow$ ECUMINE2EANDCOMOVERHEAD(E2E)
7: **return** ECU

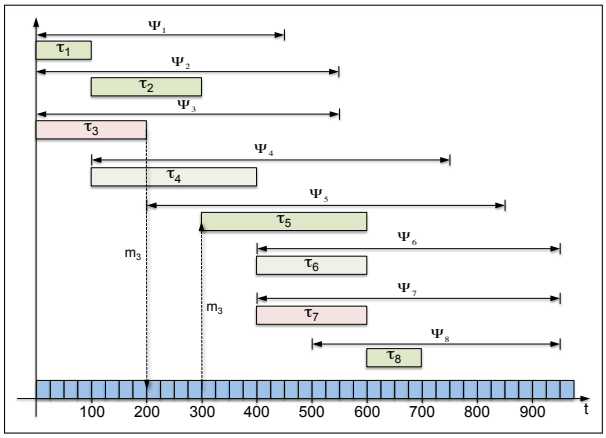
As mentioned above, Algorithm 4 serves to determine the ECU resulting in the minimum overall end-to-end delay. For this purpose, it gets the task τ_i to map, the set $\mathcal{E}_{\text{tmp}}$ of feasible candidate ECUs, the set Θ of available FlexRay slots and the current status of the redundancy mapping M^{τ}_{cur} as input. It analyzes each $\text{ECU}_i \in \mathcal{E}_{\text{tmp}}$ based on its current mapping status. In each iteration it complements the current mapping M^{τ}_{cur} to M^{τ}_i by mapping and inserting τ_i to the current schedule of ECU_i preserving order and precedence constraints by means of the given timing constraints.

The performed mapping and insertion may result in task shiftings and growing execution delays due to the constraints on one or more of the ECUs. Therefore, the algorithm calculates the overall end-to-end delay for all event chains implied by M^{τ}_i and stores it in $\text{E2E}_{\text{ECU}_i}$ referencing to ECU_i. By doing so, it implicitly determines the resulting communication overhead, i.e. the inter-ECU messages, for each mapping. Here, similar to Algorithm 1 it also considers the first available slot in Θ for resulting inter-ECU message transmissions. Iterating over each $\text{ECU}_i \in \mathcal{E}_{\text{tmp}}$ this results in a complete set E2E of overall end-to-end delays, i.e. one for each possible task-to-ECU mapping. Finally, Algorithm 4 compares the overall end-to-end delays in E2E and returns the ECU with the minimum value. If there is more than one ECU with the minimum value it returns the one with the lowest communication overhead to avoid additional inter-ECU messages.[23]

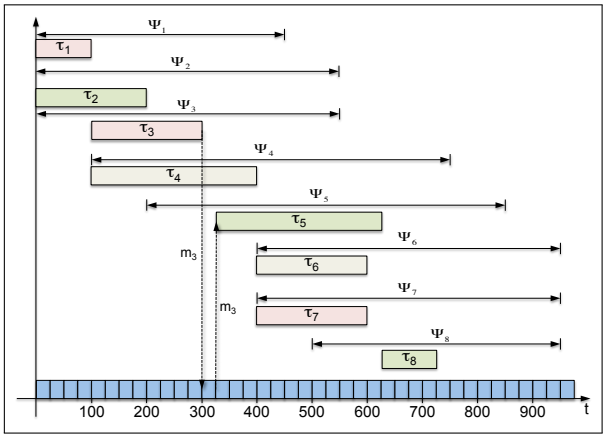
By means of Algorithm 3 and 4 our deployment approach tries to determine a feasible redundancy mapping for a considered ECU failure resulting in the shortest overall end-to-end delay for each task in Γ_{fail}. Similar to the initial task mapping it is also possible that no feasible solution can be found for the given inputs. However, the algorithms always terminates, either returning a feasible redundancy mapping after iterating over all tasks or with a negative result in M^{τ}_{red}. For the determination of the ECU with the shortest overall end-to-end delay Algorithm 4 analyzes the assigned tasks and all slots iterating over each given ECU (cf. line 3). This results in a worst case running time of $O(|\mathcal{E}| \cdot |\Gamma| \cdot |\Theta|)$. Algorithm 3 invokes Algorithm 4 for each $\tau_i \in \Gamma$ to calculate the resulting end-to-end delay for the mapped task sets on all ECUs (cf. line 5). Hence, the time complexity of Algorithm 3 is $O(|\Gamma|^2 \cdot |\mathcal{E}| \cdot |\Theta|)$.

Figure 4.19 depicts the Gantt Charts for the local schedulings and the overall end-to-end delays resulting from the iterative redundancy task mappings to the different remaining ECUs in case of a failure of ECU_1. Considering the given network topology with three ECUs, the tasks $\tau_1, \tau_4, \tau_6 \in \Gamma_{\text{fail}}$ can be mapped to ECU_2 or ECU_3. Figure 4.19(a) and 4.19(b) illustrate the two different options for task τ_1 with the resulting local schedules and overall end-to-end delays. A comparison of these figures shows that a mapping to

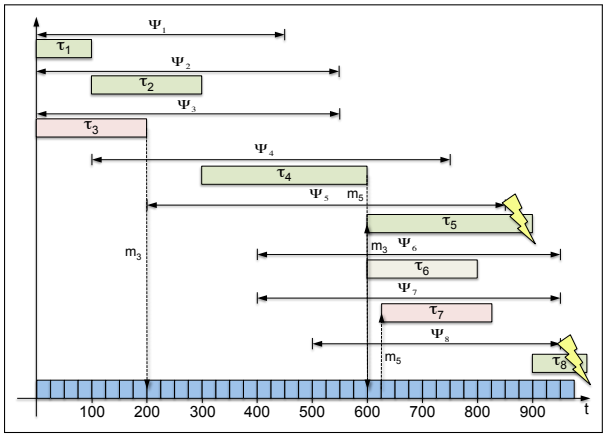
[23] Here it is also possible to further prioritize the avoidance of inter-ECU communication overhead by allowing to return ECUs with longer overall end-to-end delays and setting the corresponding threshold.



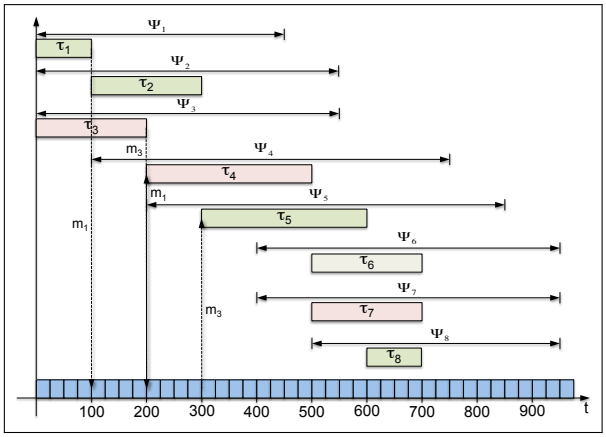
(a) Redundancy mapping of τ_1 to ECU_2.



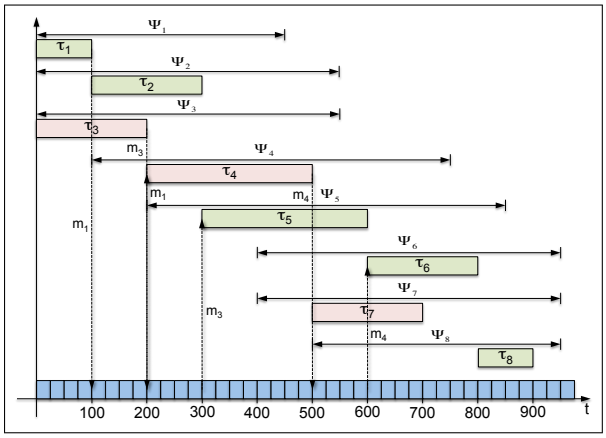
(b) Redundancy mapping of τ_1 to ECU_3.



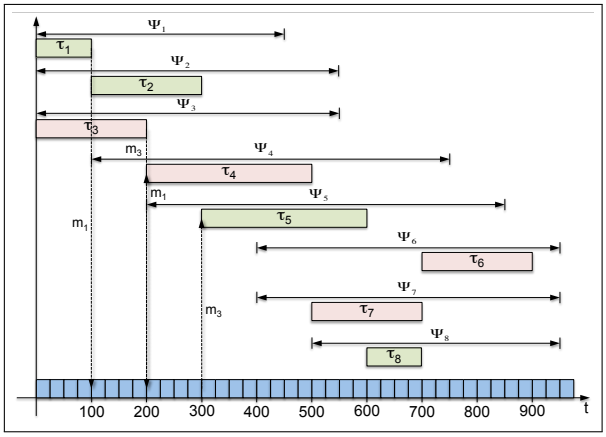
(c) Redundancy mapping of τ_4 to ECU_2.



(d) Redundancy mapping of τ_4 to ECU_3.



(e) Redundancy mapping of τ_6 to ECU_2.



(f) Redundancy mapping of τ_6 to ECU_3.

Figure 4.19: Gantt Charts for local schedulings and overall end-to-end delays resulting from redundancy task mappings to different remaining ECUs (Failure of ECU_1).

ECU_2 results in an overall end-to-end delay of $700\mu s$ and to ECU_3 in $725\mu s$. Hence, Algorithm 4 returns ECU_2 for the mapping of τ_1. Based on the re-

sulting schedules and the updated start and finishing times of the tasks the approach determines the best candidate ECU for a redundancy mapping of τ_4 in the next iteration step. Figure 4.19(c) shows that a mapping of τ_4 to ECU_2 results in a deadline miss of τ_5 and its direct successor τ_8. Consequently, the algorithm returns ECU_3 for the mapping of τ_4, which also results in an overall end-to-end delay of $700\mu s$. Furthermore, this mapping implies the additional inter-ECU message m_1 from ECU_2 to ECU_3. The last task in Γ_{fail} to map is τ_6. Figure 4.19(e) and 4.19(f) show that both options result in an overall end-to-end delay of $900\mu s$. However, the mapping of τ_6 to ECU_2 requires the additional inter-ECU message m_4 from ECU_3 to ECU_2. Thus, Algorithm 4 returns ECU_3 for this redundancy mapping to avoid this additional communication overhead.

Summarized, Figure 4.19 illustrates how our approach determines a feasible reconfiguration for a failure of ECU_1. All deadlines, timing, order and precedence constraints are fulfilled. Furthermore, unnecessary communication overhead is avoided by adding only one more inter-ECU message. Figure A.1 and A.2 in Appendix A depict the Gantt Charts for the corresponding iterative redundancy task mappings to the different remaining ECUs in case of a failure of ECU_2 and ECU_3 based on the inital mapping. These figures show that our approach determines feasible redundancy mappings for all possible reconfigurations of the given example. Obviously, similar to the initial mapping, in case of an input TDG representing a multirate system, our approach has to analyze and compare the overall end-to-end delays of all task instances respectively their corresponding event chains.

Finally, each redundancy mapping returns a TMG representing M_{red}^{τ} for the corresponding reconfiguration. Figure A.3 and A.4 in Appendix A illustrate the performed task mappings and the resulting TMGs of the three reconfigurations for the input graphs shown in 4.12(b) and Figure 4.13 . For instance, Figure A.4(a) depicts the TMG for the reconfiguration in case of a failure of ECU_1. Compared to the TMG for the initial configuration in Figure 4.18(b), it shows that, as described above, τ_1 is mapped to ECU_2 while τ_4 and τ_6 are mapped to ECU_3. Furthermore, the comparison shows that resulting from this mapping m_1 from ECU_2 to ECU_3 becomes an additional inter-ECU message while m_5 becomes an intra-ECU message on ECU_3.

4.4.2 Bus Mapping

The performed task mappings described above result in the necessity of inter-ECU communication over the bus for several messages. Consequently, a corresponding bus mapping and scheduling for these messages has to be performed. Especially in huge distributed systems with many ECUs communicating over the bus, an efficient and flexible bus mapping is required to reduce the resulting communication delays and determine a feasible schedule.

Therefore, our approach analyzes the inter-ECU communication of the initial configuration and all reconfigurations to perform a combined bus mapping that utilizes message respectively slot reuse and frame packing for inter-ECU messages with the same sender ECU.

Algorithm 5 (BUSMAPPING) represents in pseudocode how our approach realizes the combined bus mapping. As input the algorithm gets the set $\mathfrak{M}^{\tau}$ of task mappings, i.e. TMGs of the initial mapping and all redundancy mappings, HAG G_{HAG}, and COMG G_{COMG}. It starts with the bus mapping for the initial configuration. The algorithm determines the inter-ECU messages $\mathcal{M}^{\text{bus}}_{\text{M}^{\tau}_{\text{init}}}$ of the initial task mapping $\text{M}^{\tau}_{\text{init}} \in \mathfrak{M}^{\tau}$ and the set $\mathcal{E}_{\text{tx}}$ of their corresponding sender ECUs. For each sender $\text{ECU}_i \in \mathcal{E}_{\text{tx}}$ it identifies the transmitted inter-ECU messages $m_j \in \mathcal{M}^{\text{bus}}_{\text{ECU}_i}$. To perform the bus mapping, the algorithm iterates over all messages $m_j \in \mathcal{M}^{\text{bus}}_{\text{ECU}_i}$ and determines the available slots Θ_{m_j} in their available transmission time intervals Υ_{m_j}.

As described in Section 4.4.1, the initial task mapping always utilizes the first available slot for an inter-ECU message transmission. Moreover, this mapping to the first available slot implicitly applies frame packing if possible. Obviously, in the initial configuration frame packing is only applicable for different inter-ECU messages sent by the same task.[24] In this case, several messages can be transmitted in the same frame respectively slot θ_{map} if their summed size fits in the defined available maximum slot payload (cf. Section 4.2.6). After the actual mapping the start times s_{m_j} and finishing times f_{m_j} of all messages $m_j \in \mathcal{M}^{\text{bus}}_{\text{ECU}_i}$ are updated.

Algorithm 5 continues the bus mapping for each redundancy task mapping $\text{M}^{\tau}_i \in \mathfrak{M}^{\tau} \setminus \{\text{M}^{\tau}_{\text{init}}\}$. Here it also determines the inter-ECU messages $\mathcal{M}^{\text{bus}}_{\text{M}^{\tau}_i}$ and the set $\mathcal{E}_{\text{tx}}$ of their corresponding sender ECUs to identify the transmitted inter-ECU messages $m_k \in \mathcal{M}^{\text{bus}}_{\text{ECU}_j}$ from sender ECU_j in each task mapping M^{τ}_i. Analogues to the bus mapping for the initial configuration, the algorithm iterates over all messages $m_k \in \mathcal{M}^{\text{bus}}_{\text{ECU}_j}$ and determines the available slots Θ_{m_k} in their available transmission time intervals Υ_{m_k}. However, to make use slot reuse if possible it checks if m_k from ECU_j was initially mapped to a slot $\theta \in \Theta_{m_k}$. If this is the case, Θ_{m_k} is set to this assigned slot for reusing the message. In each iteration the determined set of available slots Θ_{m_k} complements the set Ξ_{ECU_j} of available slots sets for all inter-ECU messages sent by ECU_j.

Then the mapping for the determined messages is performed. Therefore, the algorithm determines the overlapping slots Ω_{ECU_j} for the sets of available slots in Ξ_{ECU_j} as described in Section 4.3.2 and maps each message to the first available overlapping slot. This implicitly realizes slot reuse and frame packing if possible. Here, the iteration over the sender ECUs ensures that

[24] Considering the fact that in general the specified slot size will be smaller than the computation time of single tasks (cf. Section 4.2.4).

Algorithm 5 BUSMAPPING

Input: Set of TaskMappings (TMGs) $\mathfrak{M}^{\tau}$, $G_{\text{HAG}} = (\mathcal{E}, E)$, and $G_{\text{COMG}} = (\Theta, \emptyset)$.
Output: BusMapping $\text{M}^{\text{bus}} : (\mathcal{M}^{\text{bus}}, \mathcal{E}) \mapsto \Theta$

1: $\text{M}^{\tau}_{\text{init}} \leftarrow$ GETINITIALMAPPING($\mathfrak{M}^{\tau}$)
2: $\mathcal{M}^{\text{bus}}_{\text{M}^{\tau}_{\text{init}}} \leftarrow$ GETINTERECUMSGS($\text{M}^{\tau}_{\text{init}}$)
3: $\mathcal{E}_{\text{tx}} \leftarrow$ GETSENDERECUS($\mathcal{M}^{\text{bus}}_{\text{M}^{\tau}_{\text{init}}}, \mathcal{E}$)
4: **for all** $\text{ECU}_i \in \mathcal{E}_{\text{tx}}$ **do**
5: $\quad \mathcal{M}^{\text{bus}}_{\text{ECU}_i} \leftarrow$ GETTXMESSAGESFROMECU($\text{M}^{\tau}_{\text{init}}, \text{ECU}_i$)
6: $\quad$ **for all** $m_j \in \mathcal{M}^{\text{bus}}_{\text{ECU}_i}$ **do**
7: $\quad\quad \Theta_{m_j} \leftarrow$ GETSLOTSINTXINTERVAL(m_j)
8: $\quad\quad \theta_{\text{map}} \leftarrow$ GETFIRSTAVAILABLESLOT(Θ_{m_j})
9: $\quad\quad$ MAPTOSLOT(m_j, θ_{map})
10: $\quad$ **end for**
11: $\quad$ UPDATESTARTANDFINISHTIMES($\mathcal{M}^{\text{bus}}_{\text{ECU}_i}$)
12: **end for**
13: **for all** $\text{M}^{\tau}_i \in \mathfrak{M}^{\tau} \setminus \{\text{M}^{\tau}_{\text{init}}\}$ **do**
14: $\quad \mathcal{M}^{\text{bus}}_{\text{M}^{\tau}_i} \leftarrow$ GETINTERECUMSGS(M^{τ}_i)
15: $\quad \mathcal{E}_{\text{tx}} \leftarrow$ GETSENDERECUS($\mathcal{M}^{\text{bus}}_{\text{M}^{\tau}_i}, \mathcal{E}$)
16: $\quad$ **for all** $\text{ECU}_j \in \mathcal{E}_{\text{tx}}$ **do**
17: $\quad\quad \mathcal{M}^{\text{bus}}_{\text{ECU}_j} \leftarrow$ GETTXMESSAGESFROMECU($\text{M}^{\tau}_i, \text{ECU}_j$)
18: $\quad\quad$ **for all** $m_k \in \mathcal{M}^{\text{bus}}_{\text{ECU}_j}$ **do**
19: $\quad\quad\quad \Theta_{m_k} \leftarrow$ GETSLOTSINTXINTERVAL(m_k)
20: $\quad\quad\quad$ **if** CHECKFORSLOTREUSE(Θ_{m_k}) = **true then**
21: $\quad\quad\quad\quad \Theta_{m_k} \leftarrow$ GETASSIGNEDSLOT(m_k)
22: $\quad\quad\quad$ **end if**
23: $\quad\quad\quad \Xi_{\text{ECU}_j} \leftarrow \Xi_{\text{ECU}_j} \cup \Theta_{m_k}$
24: $\quad\quad$ **end for**
25: $\quad\quad$ MAPTOFIRSTAVAILABLEOVERLAPPINGSLOTS(Ξ_{ECU_j})
26: $\quad\quad$ UPDATESTARTANDFINISHTIMES($\mathcal{M}^{\text{bus}}_{\text{ECU}_j}$)
27: $\quad$ **end for**
28: **end for**
29: **return** $\text{M}^{\text{bus}} : (\mathcal{M}^{\text{bus}}, \mathcal{E}) \mapsto \Theta$

only messages from the same sender are reused or combined. This means that the same message transmitted by different sender ECUs in different configurations is mapped to different slots as required by the fixed slot-to-ECU of FlexRay message scheduling. As described above, the messages initially mapped to a specific slot can only be mapped to the same slot, i.e. the message is reused. Messages with overlapping sets of available slots are mapped to the first available overlapping slot if their summed payload allows frame packing. Otherwise, they have to be transmitted in separate slots. After the actual mapping the algorithm updates the start times s_{m_k} and finishing times

f_{m_k} of all messages $m_k \in \mathcal{M}^{\text{bus}}_{\text{ECU}_j}$. Consequently, in some cases, this update may also implicitly update the start time s_{rx} of the receiver task τ_{rx}. Finally, Algorithm 5 returns a combined bus mapping M^{bus} for all configurations. It always terminates after iterating over all determined inter-ECU messages, either with a feasible solution or with a negative result in M^{bus}. To perform the bus mapping, the algorithm has to iterate over each transmitted inter-ECU message from each sender ECU and analyze the slot set to determine available slots. This results in a worst case time complexity of $O(|\mathcal{E}| \cdot |\mathcal{M}| \cdot |\Theta|)$.

Table 4.4 summarizes the properties of the inter-ECU messages for the initial configuration and reconfigurations resulting from the task and bus mappings performed for the given example. It shows that the initial configuration requires the two inter-ECU messages m_3 from ECU_3 to ECU_2 and m_5 from ECU_1 to ECU_3. Since the initial task mapping utilizes the first available slot for an inter-ECU message transmission for the given example this results in the available transmission time intervals $\Upsilon_{m_3} = [200\mu s, 225\mu s[$ and $\Upsilon_{m_5} = [400\mu s, 425\mu s[$ (cf. Figure 4.17(a)). Consequently, Algorithm 5 determines the corresponding available slots and maps the messages to θ_9 and θ_{17}. This also updates the start times and finishing times of these messages, e.g. $s_{m_3} = s_{\theta_{m_3}} = 200\mu s$ and $f_{m_3} = f_{\theta_{m_3}} = 225\mu s$.

Initial Configuration:

m_i	$m_{(\tau_{\text{tx}},\tau_{\text{rx}})}$	$\text{ECU}_{\tau_{\text{tx}}}$	$\text{ECU}_{\tau_{\text{rx}}}$	Υ_{m_i}	Θ_{m_i}	θ_{m_i}	$s_{\theta_{m_i}}$	$f_{\theta_{m_i}}$
m_3	$m_{(\tau_3,\tau_5)}$	ECU_3	ECU_2	$[200\mu s, 225\mu s[$	$\{\theta_9\}$	θ_9	$200\mu s$	$225\mu s$
m_5	$m_{(\tau_4,\tau_7)}$	ECU_1	ECU_3	$[400\mu s, 425\mu s[$	$\{\theta_{17}\}$	θ_{17}	$400\mu s$	$425\mu s$

Reconfiguration 1 (Failure of ECU_1):

m_i	$m_{(\tau_{\text{tx}},\tau_{\text{rx}})}$	$\text{ECU}_{\tau_{\text{tx}}}$	$\text{ECU}_{\tau_{\text{rx}}}$	Υ_{m_i}	Θ_{m_i}	θ_{m_i}	$s_{\theta_{m_i}}$	$f_{\theta_{m_i}}$
m_1	$m_{(\tau_1,\tau_4)}$	ECU_2	ECU_3	$[100\mu s, 200\mu s[$	$\{\theta_5, \ldots, \theta_8\}$	θ_5	$100\mu s$	$125\mu s$
m_3	$m_{(\tau_3,\tau_5)}$	ECU_3	ECU_2	$[200\mu s, 300\mu s[$	$\{\theta_9, \ldots, \theta_{12}\}$	θ_9	$200\mu s$	$225\mu s$

Reconfiguration 2 (Failure of ECU_2):

m_i	$m_{(\tau_{\text{tx}},\tau_{\text{rx}})}$	$\text{ECU}_{\tau_{\text{tx}}}$	$\text{ECU}_{\tau_{\text{rx}}}$	Υ_{m_i}	Θ_{m_i}	θ_{m_i}	$s_{\theta_{m_i}}$	$f_{\theta_{m_i}}$
m_5	$m_{(\tau_4,\tau_7)}$	ECU_1	ECU_3	$[400\mu s, 700\mu s[$	$\{\theta_{17}, \ldots, \theta_{28}\}$	θ_{17}	$400\mu s$	$425\mu s$
m_6	$m_{(\tau_5,\tau_8)}$	ECU_3	ECU_1	$[700\mu s, 725\mu s[$	$\{\theta_{29}\}$	θ_{29}	$700\mu s$	$725\mu s$

Reconfiguration 3 (Failure of ECU_3):

No inter-ECU messages.

Table 4.4: Properties of the inter-ECU messages for initial configuration and reconfigurations resulting from task and bus mappings.

In reconfiguration 1 for the compensation of a failure of ECU_1 the redundancy mapping also results in two inter-ECU messages. The first message m_1 from ECU_2 to ECU_3 can be transmitted in $\Upsilon_{m_1} = [100\mu s, 200\mu s[$ respectively $\Theta_{m_1} = \{\theta_5, \ldots, \theta_8\}$ (cf. Figure 4.19(f)). Hence, Algorithm 5 maps the message to the first available slot θ_5. Furthermore, the inter-ECU message m_3 from ECU_3 to ECU_2 with $\Upsilon_{m_3} = [200\mu s, 300\mu s[$ respectively $\Theta_{m_3} = \{\theta_9, \ldots, \theta_{12}\}$ must be mapped to the bus. A comparison with the message mapping of the initial mapping shows that this message can be reused in reconfiguration 1. Therefore, the algorithm applies slot reuse in the slot θ_9. Here, it also updates the start times and finishing times of the messages to the corresponding values. A similar situation holds for reconfiguration 2, that compensates a failure of ECU_2. The redundancy mapping for this reconfiguration implies the two inter-ECU messages m_5 from ECU_1 to ECU_3 and m_6 from ECU_3 to ECU_1 (cf. Figure A.1(e)). The message mapping of m_5 can be kept from the initial configuration in the reused slot θ_{17}. As described above, for message m_6 the redundancy mapping algorithm utilizes the first available slot to reduce the communication delay of the receiver task τ_8. This results in the available transmission time interval $\Upsilon_{m_6} = [700\mu s, 725\mu s[$ of slot θ_{29}, to which Algorithm 5 maps message m_6. For reconfiguration 3, that compensates a failure of ECU_2, no bus mapping has to be performed because the redundancy mapping does not require any inter-ECU message (cf. Figure A.2(c)).

Table 4.4 shows that for the given example no frame packing must be performed because every ECU sends at most one message per configuration. The examples presented in Chapter 5 are more complex and contain more inter-ECU messages with overlapping slot intervals. Thus, the bus mapping also applies frame packing on these examples. Moreover, to support multirate systems, Algorithm 5 determines overlapping slots for each inter-ECU message instance to utilize frame packing for the bus mappings and reduce the number of required FlexRay slots. For this purpose, the available slots in the transmission time interval of all instances for each message are determined in line 19 of Algorithm 5 to enable frame packing by mapping to overlapping slots in line 25. Furthermore, also slot reuse is applied if possible to further decrease the number of assigned slots per ECU as described above.

By means of Algorithm 5 our design approach determines a feasible bus mapping for the initial configuration and the reconfigurations. The combined analysis and comparison of inter-ECU messages regarding their sender ECUs and their available transmission intervals enables an efficient slot assignment utilizing slot reuse and frame packing to reduce the number of required redundant slots per reconfiguration. Especially for large systems with many ECUs and inter-ECU messages, this reduction helps to determine feasible bus mappings for a high number of required reconfigurations. As mentioned in Section 4.2, the actual realization and implementation of task context mi-

gration for the Cold Standby redundancy is beyond the scope of this thesis. However, slots which are not assigned to an ECU in any of the configurations could be utilized for this purpose in future works.

4.5 Design Result and Feedback

In the previous section we described how our approach performs the necessary task and bus mappings for the design of a fault-tolerant distributed real-time system by means of the presented algorithms. As shown in Figure 4.1, the actual deployment of our design approach implies mappings for the initial configuration and all necessary reconfigurations and finally returns a design result.

Algorithm 6 DEPLOYMENT

Input: TDG $G_{\text{TDG}} = (\Gamma, \mathcal{M})$ and $G_{\text{HAG}} = (\mathcal{E}, E)$, and $G_{\text{COMG}} = (\Theta, \emptyset)$.
Output: DG $G_{\text{DG}} = (\mathcal{E} \cup \Theta, \mathcal{M})$ with TaskMapping $\Gamma \mapsto \mathcal{E}$ and BusMapping $(\mathcal{M}^{\text{bus}}, \mathcal{E}) \mapsto \Theta$.

1: $\text{M}^{\tau}_{\text{init}} \leftarrow$ INITIALTASKMAPPING$(G_{\text{TDG}}, G_{\text{HAG}}, G_{\text{COMG}})$
2: $\mathfrak{M}^{\tau} \leftarrow \text{M}^{\tau}_{\text{init}}$
3: **for all** $\text{ECU}_i \in \mathcal{E}$ **do**
4: $\quad \text{M}^{\tau}_{\text{red}} \leftarrow$ REDUNDANCYTASKMAPPING$(\mathcal{E} \setminus \{\text{ECU}_i\}, G_{\text{TMG}}(\text{M}^{\tau}_{\text{init}}), G_{\text{COMG}})$
5: $\quad \mathfrak{M}^{\tau} \leftarrow \mathfrak{M}^{\tau} \cup \text{M}^{\tau}_{\text{red}}$
6: **end for**
7: $\text{M}^{\text{bus}} \leftarrow$ BUSMAPPING$(\mathfrak{M}^{\tau}, G_{\text{HAG}}, G_{\text{COMG}})$
8: $G_{\text{DG}} \leftarrow$ CREATEDESIGNGRAPH$(\mathfrak{M}^{\tau}, \text{M}^{\text{bus}})$
9: **return** DG $G_{\text{DG}} = (\mathcal{E} \cup \Theta, \mathcal{M})$ with TaskMapping $\Gamma \mapsto \mathcal{E}$ and BusMapping $(\mathcal{M}^{\text{bus}}, \mathcal{E}) \mapsto \Theta$

Algorithm 6 (DEPLOYMENT) represents in pseudocode how our approach performs this combined deployment which includes all required mappings. As input the deployment algorithm gets the software architecture modeled as TDG $G_{\text{TDG}} = (\Gamma, \mathcal{M})$ and the network topology including the hardware and bus configuration represented by a HAG $G_{\text{HAG}} = (\mathcal{E}, E)$ and a COMG $G_{\text{COMG}} = (\Theta, \emptyset)$. By means of this input it performs the task mapping for the initial configuration calling Algorithm 1 which determines $\text{M}^{\tau}_{\text{init}}$, and adds the initial mapping to the set of task mappings $\mathfrak{M}^{\tau}$. Afterwards the algorithm determines the redundancy task mappings for all required reconfigurations. Therefore, it iterates over all $\text{ECU}_i \in \mathcal{E}$ of the HAG and calls Algorithm 3 passing the set of remaining ECUs $\mathcal{E} \setminus \{\text{ECU}_i\}$ considering a failure of ECU_i, the initial mapping $\text{M}^{\tau}_{\text{init}}$ as TMG, and the COMG. In each iteration, the set of task mappings $\mathfrak{M}^{\tau}$ is complemented by the currently performed redundancy mapping $\text{M}^{\tau}_{\text{red}}$. When all necessary task mappings are performed, the deployment algorithm executes Algorithm 5 to perform the bus mapping by means of the task mapping set $\mathfrak{M}^{\tau}$, the HAG, and the COMG. Finally, based on the determined task mappings $\mathfrak{M}^{\tau}$ and the bus mapping M^{bus}, Al-

gorithm 6 creates a *design graph* (DG) and returns this graph as design result with the corresponding deployment, i.e. task mapping $\Gamma \mapsto \mathcal{E}$ and bus mapping $(\mathcal{M}^{\text{bus}}, \mathcal{E}) \mapsto \Theta$, for the given system model input.

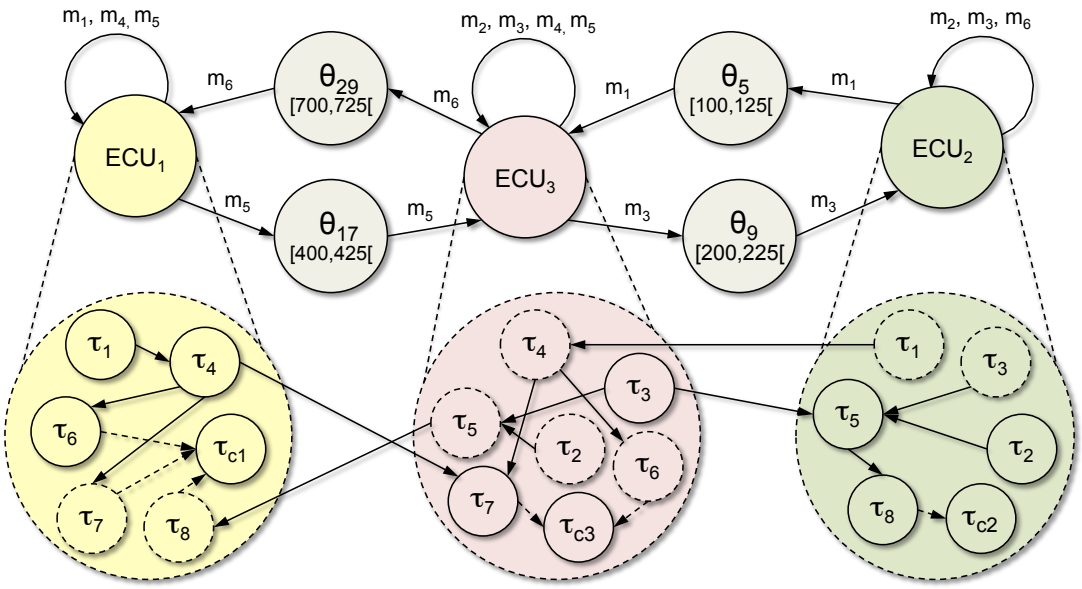


Figure 4.20: DG for input graphs shown in 4.12(b) and Figure 4.13 .

A DG $G_{\text{DG}} = (V, E) = (\mathcal{E} \cup \Theta, \mathcal{M})$ combines the given set of TMGs for the task mappings with the slots from the input COMG assigned to the required inter-ECU messages by the bus mapping for all configurations. Therefore, a DG comprises the set $\mathcal{E}$ and the assigned slots Θ as vertices and the messages $\mathcal{M}$ as edges. The corresponding annotations to the ECUs, slots, and messages are taken from the input graphs. Figure 4.20 depicts the DG for the given example and combines the TMG of the intial configuration shown in Figure 4.18(b) and the three TMGs for the reconfigurations shown in Figure A.4. Thus, beside the active tasks mapped for the initial configuration and its coordinator task each ECU also contains the determined redundant task replicas for the reconfigurations. For instance, ECU_1 hosts the tasks τ_1, τ_4, and τ_6 which are active by default and the task replicas τ_7 and τ_8 which are inactive by default and activated by the DCC for the corresponding reconfiguration in case of a ECU failure (cf. Section 4.2.2). For a better traceability, the redundant task replicas are indicated by dotted outlines.

Furthermore, a DG combines and represents the resulting messaging for each configuration. The messages exchanged between tasks which are active by default, e.g. m_1 from τ_1 to τ_4, are transmitted in the initial configuration, i.e. in a system without an ECU failure. The messages which are sent or received by a redundant task replica, e.g. m_5 from τ_4 to τ_7, are only transmitted in the corresponding reconfiguration. For the required inter-ECU messages determined by Algorithm 5, a DG also represents the corresponding information. Therefore, beside the ECUs it comprises the assigned slot from G_{COMG} for

each inter-ECU message as vertex. As summarized in Table 4.4, the combined bus mapping for the given example results in four different inter-ECU messages mapped to the slots θ_5, θ_9, θ_{17}, and θ_{29}. Each slot in the DG is connected with the sender and receiver ECUs via incoming and outgoing edges representing the mapped messages. Since a slot can only be assigned to one sender ECU, it has only one incoming edge. However, if frame packing is utilized for a slot, the incoming edge is annotated with the combined corresponding message IDs. The number of outgoing edges from a slot depend on the utilized message model. Here, we consider the message model from [DTT08] where each message sent by a task must be scheduled separately on the bus. Hence, each slot also has only one outgoing edge.[25] Inter-ECU messages exchanged between tasks which are active by default, e.g. m_3 from τ_3 on ECU_3 to τ_5 on ECU_2, are transmitted in the initial configuration and can be reused in the reconfigurations. Inter-ECU messages exchanged between redundant task replicas, e.g. m_6 from τ_5 on ECU_3 to τ_8 on ECU_1, are transmitted in the corresponding reconfiguration.

Summarized, the DG represents the whole design result containing the complete deployment for all possible configurations determined by Algorithm 6. Thus the system designer can use this feedback and perform the determined deployment. This includes the mapping of tasks to ECUs and the generation of the corresponding task lists for the distributed coordinator tasks as described in Section 4.2.2. Moreover, for the bus mapping and message scheduling the designer can directly derive the COM matrix to configure the FlexRay bus accesses. Table 4.5 shows the resulting COM matrix derived from the DG in Figure 4.20. As described in Section 4.3.2, the first three and the last three slots of the FlexRay communication cycle are pre-assigned for communication with the I/O over the peripheral interfaces. Slot θ_5 is assigned to ECU_2 for the transmission of m_1 and ECU_3 receives the message in this slot. The slots θ_9 and θ_{29} are assigned to ECU_3 for write access while ECU_1 may transmit its inter-ECU message in slot θ_{17}.

Slot	θ_1	...	θ_5	...	θ_9	...	θ_{17}	...	θ_{29}	...	θ_{39}	
ECU 1	**tx**:I/O	...	-	...	-	...	**tx**:m_5	...	rx:m_6	...	-	
ECU 2	-	...	**tx**:m_1	...	rx:m_3	...	-	...	-	...	-	
ECU 3	-	...	rx:m_1	...	**tx**:m_3	...	rx:m_5	...	**tx**:m_6	...	rx:I/O	

Table 4.5: COM matrix for input graphs shown in Figure 4.12(b) and 4.13 derived from DG in Figure 4.20.

For the given example the deployment algorithm is able to determine feasible task mappings for the initial configuration and all three required reconfigurations as well as a feasible combined bus mapping. However, it is also possible

[25] If the message model with multiple receivers from [KHM03] is applied, each slot has one outgoing edge for each receiver ECU.

that our approach cannot determine a completely feasible solution but only for a subset of all configurations. For instance, in a heterogeneous system it might happen that for the failure of a powerful ECU no appropriate redundancy mapping can be performed by Algorithm 3. In this case the designer has to decide how he reacts on this feedback. On the one hand, he can design the system according to the determined solution accepting that only a subset of possible ECU failures can be compensated. On the other hand, he can adjust the hardware topology respectively its input model, e.g. with additional or more powerful ECUs, and perform a further iteration of the design approach.

In any case, Algorithm 6 always terminates after iterating over all input graphs by executing the presented algorithms for initial task mapping, redundancy task mapping, and bus mapping. However, it depends on the results of the performed mappings if it provides a complete or partly solution. The algorithms for the initial task mapping and the bus mapping are executed once while the redundancy task mapping is invoked $|\mathcal{E}|$ times for the possible ECU failures. Summing up the running time of these algorithms the entire time complexity of Algorithm 6 is $O(|\Gamma|^3 + |\mathcal{E}|^2 \cdot |\Gamma|^2 \cdot |\Theta| + |\mathcal{E}| \cdot |\mathcal{M}| \cdot |\Theta|)$. This means that the deployment of our fault-tolerant design approach can be executed in polynomial time and its runtime is obviously depending mainly on the number of tasks and ECUs in the system. However, since our approach is performed during the design phase and can be executed on perfomant general-purpose PCs instead of embedded processors, it is easily applicable for large scaled systems. To prove this point and to determine and provide quantitative runtime values, we developed a prototypical implementation of our approach based on the pseudocode algorithms described above.

System Input:		Runtime:	
Software Architecture	Network Topology	Initial Mapping	Deployment
8 tasks, 6 messages	3 ECUs	$\sim 5ms$	$\sim 15ms$
24 tasks, 18 messages	9 ECUs	$\sim 10ms$	$\sim 170ms$
72 tasks, 54 messages	27 ECUs	$\sim 45ms$	$\sim 2{,}000ms$
216 tasks, 162 messages	81 ECUs	$\sim 400ms$	$\sim 80{,}000ms$

Table 4.6: Exemplary runtimes of inital mapping and deployment algorithm.

Table 4.6 provides some exemplary runtimes of this implementation executed on a general-purpose PC with a Intel Core i7 2.7 GHz processor and 8 GB main memory. Based on the TDG and HAG examples presented above we prepared additional contrived examples with multiples of tasks ans ECUs as system input to measure the resulting runtimes for the initial mapping and the complete fault-tolerant deployment.[26] The results in Table 4.6 show that for small systems with less than 25 tasks and 10 ECUs the complete fault-

[26]For each input we performed 20 runs and measured the approximate average runtime.

tolerant deployment is determined in less than 200*ms*. For larger systems with more tasks already the initial mapping requires more runtime. Obviously, a large number of ECUs implies the corresponding number of reconfigurations which results in a rapidly increasing runtime for the fault-tolerant deployment. For our example with 72 ECUs and 27 ECUs, it already takes approximately 2,000*ms* and for the example with 216 tasks and 81 ECUs the required runtime grows to approximately 80,000*ms*, i.e., 1 minute and 20 seconds.

As described in Chapter 1 even the largest and most complex distributed systems in modern automotive vehicles typically do not contain much more than 100 ECUs. Moreover, these systems are composed of different subnetworks and only a subset of functions and components are part of safety-critical subsystems. Thus, it can be concluded that our approach is very scalable since an iteration of the fault-tolerant deployment for a realistically scaled system can be performed by the designer in less than one minute even if we consider cross-domain dependencies and data exchange between the different subnetworks.

4.6 Summary

This chapter presented our approach for the design of fault-tolerant distributed real-time systems. It gave an overview of the approach and described how it extends and refines existing design flow steps for distributed systems. The main objectives, properties, and important constraints of our approach, which enable a fault-tolerant design to compensate one arbitrary component fault, were described. This includes the proposal of a reconfigurable distributed system topology combined with the distributed coordinator concept for fault tolerance by means of self-reconfiguration and dynamic redundancy.

Based on introduced concepts and techniques for the specification and configuration of system properties and constraints, the chapter presented our graph-based modeling concept for the given software architecture and hardware topology as input to enable a feasible, efficient, and flexible system design. As an essential part of our fault-tolerant design approach, the system deployment and its performed task and bus mappings for the inital configuration and necessary reconfigurations were described and applied to a comprehensible example. Finally, this chapter provided our graph-based representation of the whole system design result containing the complete fault-tolerant deployment for all possible configurations as feedback for the designer. Depending on this feedback the designer can react on the returned result: Either, he can design the system according to the determined solution or he can modify the input model to perform a further iteration if necessary.

Chapter 5

Application to the Design of Automotive Real-Time Systems

Today's modern automotive vehicles provide numerous complex electronic features realized by means of distributed real-time systems. Many of these systems implement and provide safety-critical functions. The fast growing number and complexity of such features results in an increasing number of ECUs. Safety-critical distributed functions have to fulfill hard real-time constraints to guarantee a dependable functionality. Furthermore, subsystems are often developed by different partners and suppliers and have to be integrated. Therefore, the design and development of these systems is a very complex task. Additionally, more and more resources have to be spent on adapting already existing solutions to different environments. As a result, some of the main drivers in automotive electronics are the reduction of hardware costs and the implementation of new innovative features [Fen+06].

In the previous chapter we introduced our design approach for fault-tolerant distributed real-time systems. In this chapter we present how to utilize our approach for the design of automotive real-time systems. Therefore, we describe the application of our approach to the modeling and deployment of automotive real-time software based on a well-defined and standardized methodology for the development of automotive systems. The concepts described in this chapter are based on work we presented in [Klo+13].

5.1 Introduction to AUTOSAR

To address the challenges mentioned above the *AUTomotive Open System ARchitecture* (AUTOSAR) development partnership was founded. AUTOSAR is a worldwide development partnership of car manufacturers, suppliers and other companies from the electronics, semiconductor, and software industry.

Since 2003 more than 160 partners organized as ”core partners”, ”premium partners”, ”associated partners”, ”development partners”, and ”attendees” are working on the development and establishment of an open standardized software architecture for automotive systems. The nine core partners which have brought together their respective areas of expertise to define an automotive open system architecture standard to support the needs of future in-car applications are: BMW, BOSCH, Continental, Daimler, Ford, GM, PSA, Toyota, and Volkswagen. The main idea of AUTOSAR is a strong global partnership that creates one common standard: ”Cooperate on standards – compete on implementation” [AUT13a]. For this purpose, AUTOSAR addresses the following issues [Rei09]:

- Standardization of important system functionality,
- scalability,
- reallocation of functionality within the vehicle network,
- integration, interchangeability, and reusability of software from different manufacturers,
- support of commercial off-the-shelf software, i.e. series software without individual adaption, and
- maintainability throughout the whole product life cycle.

Based on these issues the AUTOSAR development cooperation defines common standards to keep the growing complexity and costs of automotive system development manageable and controllable. Therefore, it offers a standardization for the software architecture of automotive ECUs and defines a methodology to support a distributed function-driven development process [SR08]. Nowadays, AUTOSAR is established and is considered as de facto standard. In this thesis we consider the currently latest AUTOSAR release 4.1. In the following sections we describe the basic concepts of AUTOSAR.

5.1.1 AUTOSAR Software Architecture

The AUTOSAR initiative defines a software architecture for automotive ECUs which decouples the software from the ECU hardware. The software is composed of several functional components called *software components* (SWCs). These SWCs can be implemented independently – even by different developers – and combined to a complete project by means of a widely automatic configuration process [ZS07]. The AUTOSAR Software Architecture is a layered architecture which consists of three layers. Figure 5.1 depicts the most abstract view on these three layers. It shows the *Application Layer*,

Runtime Environment (RTE), and *Basic Software* (BSW) which are based on the microcontroller of an ECU.

On top the Application Layer provides the actual application software structured by SWCs. The AUTOSAR Software Architecture enables the software developer to design an AUTOSAR application almost independent from the involved hardware [War09]. Since the AUTOSAR Software Architecture and especially the RTE hide the network from the application, there is no knowledge about the network required. There is also nearly no knowledge about the used ECUs required, since the software architecture abstracts from the specific ECU and its microcontroller [War09]. Section 5.1.2 describes the AUTOSAR SWCs as most important structural element of the Application Layer.

The RTE and the BSW are responsible for the abstraction between the hardware and the application software [War09]. The RTE is an ECU-specific software layer which is generated automatically at a later phase of the development. It provides a standardized *application programming interface* (API) for the application software and acts as a middleware between application layer and the BSW of the ECU [AUT13p]. The RTE manages the communication between the different elements of the Application Layer as well as between the hardware independent application software and the hardware-dependent BSW layer.

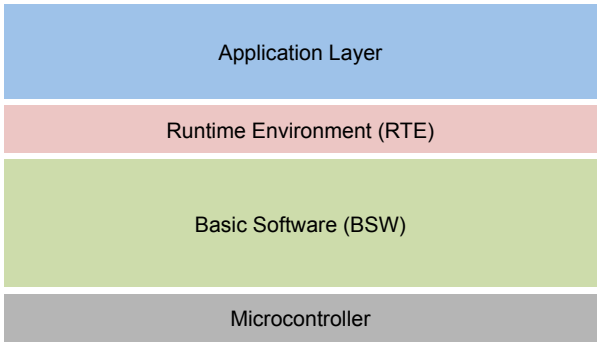
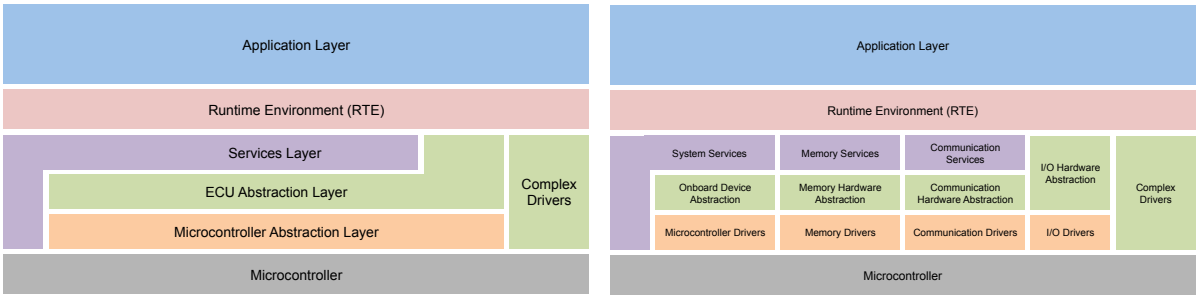


Figure 5.1: Most abstract view on layers of the AUTOSAR Software Architecture [AUT13e].

The ECU-specific BSW layer provides several interfaces called BSW modules to offer infrastructural services for the application software [AUT13m]. As depicted in Figure 5.2(a), by means of these services the BSW layer can further be divided into three different layers, namely the *Service Layer*, the *ECU Abstraction Layer*, and the *Microcontroller Abstraction Layer*. The Microcontroller Abstraction Layer offers drivers to abstract from specific controllers on an ECU. Therefore, the drivers provided by the Microcontroller Abstraction Layer offer interfaces to the ECU Abstraction Layer to enable generalized usage of different microcontrollers of the same kind. The ECU Abstraction Layer abstracts from the location of the controller. For layers

above the ECU Abstraction Layer it is not necessary to know if the controller is an on-chip or on-device controller. The Service Layer provides basic services for each AUTOSAR application, e.g. for communication. An AUTOSAR application can access these services through standardized AUTOSAR modules [War09].



(a) Different refined layers in the BSW layer.

(b) Stacks of functionality in the BSW layer.

Figure 5.2: Refined views on BSW layer of the AUTOSAR Software Architecture [AUT13e].

As shown in Figure 5.2(b), the BSW layer can also be divided into different stacks corresponding to the general functionality the basic software provides. These stacks are orthogonal to the Basic Software layers and offer the following functionality [War09]:

- The *System Stack* consisting of *Microcontroller Drivers*, *Onboard Device Abstraction*, and *System Services* provides standardized services and library functions, e.g. for timer operations or operating system functionality. It also provides ECU-specific services like ECU state management and watchdog management for hardware components.
- The *Memory Management Stack* consisting of *Memory Drivers*, *Memory Hardware Abstraction*, and *Memory Services* provides standardized access to non-volatile memory. Through this stack each software application component can allocate memory to maintain its internal state. Because of the abstraction layers of the Memory Management Stack the application component does not need to know if it is internal or external memory. The component only requests the memory via the standardized interface.
- The *Communication Stack* consisting of *Communication Drivers*, *Communication Hardware Abstraction*, and *Communication Services* provides standardized access to the vehicle network system. Through this stack software components can communicate with each other even if they are located on different ECUs. The components do not use the Communication Stack directly, but via the RTE. The RTE manages the communication between the components corresponding to their location. Components on the same ECU communicate directly while for

the communication of components on different ECUs the Communication Stack is used.

- The *I/O Stack* consisting of *I/O Drivers* and *I/O Hardware Abstraction* provides standardized access to sensors, actuators, and other peripherals. This stack does not have a service layer since there is no general interface for all possible sensors and actuators. Therefore, special software components are required to access sensors and actuators, named *Sensor/Actuator Software Components*. Considering the reconfigurable distributed system topology presented in Section 4.2.1, these components are solely required on peripheral interface nodes. They are described in more detail in Section 5.1.2

Beside the different layers and stacks described above the BSW also provides *Complex Device Drivers* [AUT13e]. The Complex Drivers Layer spans from the hardware to the RTE and enables it to bypass the BSW layer abstraction. It allows to integrate special purpose functionality, e.g. drivers for devices which are not specified within AUTOSAR, with very strict timing constrains etc., or for migration to introduce existing or new concepts into the AUTOSAR Software Architecture [AUT13c]. This can be useful for application components which need direct access to the hardware devices on the ECU, e.g. injection control or electronic valve control applications. These complex drivers are not provided by AUTOSAR but each manufacturer which needs those drivers has to implement them individually [War09].

This thesis shall not describe all BSW modules in detail. Hence, for a detailed description of the BSW modules the reader is referred to [AUT13e]. For more information on the RTE the reader is referred to [AUT13p] and [Nau09] and details about the single modules in the Communication Stack are given in [Gos09].

5.1.2 AUTOSAR Software Components

The Application Layer in AUTOSAR consists of interconnected SWCs that encapsulate a complete or at least a partial functionality of an application software. For this purpose, a SWC can basically be typed as *Atomic-*, *Parameter-*, or *Composition-SWC*. Each SWC is specified according to the AUTOSAR *Software Component Template* [AUT13f]. The specification of an Atomic-SWC contains the description of the three parts: *AtomicSoftwareComponentType*, *SwcInternalBehavior*, and *Implementation*. This enables a clear separation between

- the description of communication interfaces with other SWCs,
- the real-time behavior of a SWC, and

- the concrete implementation as source or object code.

A Composition-SWC is composed of SWCs and thus provides a hierarchical structuring of SWCs, i.e. it is a logical interconnection of other components, either atomic or again composition. The goal of a Parameter-SWC is to make calibration data visible for other SWCs within a Composition-SWC.

Beside these types, there exist several kinds of AUTOSAR SWCs for special purposes [AUT13r]. For instance, as mentioned above, Sensor/Actuator-SWCs are utilized for data exchange with the I/O. A Sensor-SWC is responsible for reading sensor data and providing its data to other components. The reading of sensor data is done through the I/O Stack of the Software Architecture. An Actuator-SWC is responsible for setting the state of an actuator on an ECU. Therefore, it can also provide an interface to other components, enabling these components to initiate the state setting of the actuator [War09]. Obviously, these components have to be deployed on an ECU with the corresponding sensors or actuators. As described in Section 4.2.1, we introduce dedicated peripheral interface nodes for data exchange with the I/O and focus on the deployment of application software to the functional nodes of the system. However, the software designer still needs to know which hardware sensors and actuators are used in the software application to design an appropriate AUTOSAR-compliant application.

AUTOSAR Ports

To connect SWCs each component has well-defined *Ports* through which the component can communicate with other SWCs or with the BSW modules. A Port belongs to exactly one SWC and represents a point of interaction between this specific component and other components in the system. The kind of interaction that a specific port provides, i.e. its communication semantics, is defined by an *Interface* that is provided or required by that port. A Port can either provide the service of an Interface, making it a *PPort*, or require the service of an Interface, making it an *RPort*. AUTOSAR provides several kinds of Interfaces for different types of data exchange [AUT13r]. However, it defines two basic communication paradigms:

- *Client-Server*: This kind of Interface is commonly used for function invocation and access to infrastructural functionality from the Application Layer. To use a specific service, the client calls a specified operation respectively function provided by the server. A call to a method of such an interface can be either blocking, which denotes the communication of client and server is synchronous, or non-blocking, which means asynchronous communication. In either case the client awaits a response from the server.

- *Sender-Receiver*: Such an Interface is used for asynchronous signal passing where a sender provides signals which can be received by one or more receivers. All calls are non-blocking calls and the sender will never get a response.

Figure 5.3 depicts SWCs interconnected by Ports with Sender-Receiver (a) and Client-Server Interfaces (b). It illustrates the AUTOSAR symbols for the different Port types [AUT13r]. The SenderSWC and the ServerSWC have PPorts, while the ReceiverSWC and the ClientSWC have RPorts all corresponding to the communication paradigm of their respective Interfaces.

(a) Sender-Receiver Interface. (b) Client-Server Interface.

Figure 5.3: Examples of SWCs interconnected by Ports with Sender-Receiver and Client-Server Interfaces.

During the application and component design, the later distribution of the components over the ECUs is not relevant. However, the communication between the components already has to be defined during this design phase. Therefore, in that phase the components communicate through the *Virtual Function Bus* (VFB). More details about the VFB are given in Section 5.1.3.

Internal Behavior

As mentioned above the specification of an Atomic-SWC contains the SwcInternalBehavior and the Implementation of its executable part. The internal behavior is represented by a set of *Runnable Entities* (Runnables). A Runnable represents a piece of code, e.g. a function or a controller, that is for example implemented in a programming language like C or as a MATLAB/Simulink model [The13]. Moreover, a Runnable is the smallest entity that can be scheduled by the RTE. Thus, Runnables permit modeling of concurrency within an Atomic-SWC.

The attribute "Atomic" refers to the fact, that this type of SWC must be completely mapped to one ECU, i.e. all Runnables of this SWC must be mapped to the same ECU.[27] Furthermore, the unit of execution in AUTOSAR is an *Operating System task* (OS task). Hence, all Runnables are mapped to such

[27] In contrast to an Atomic-SWC, the prototypes of a composite component do not need to be deployed on the same ECU but can be distributed over several ECUs.

OS tasks to be scheduled and executed. This mapping to OS tasks is performed at the corresponding configuration of an ECU which is described in more detail in [AUT13i] and [Heb09]. An OS task may contain one or more Runnables. It is not necessary to map all Runnables of one Atomic-SWC to the same OS task. In fact, it is reasonable to distribute the Runnables of one component to different OS tasks to enable concurrent execution of Runnables within one Atomic-SWC. More details about the properties of the AUTOSAR OS are given in Section 5.1.5.

5.1.3 AUTOSAR Methodology

The AUTOSAR methodology describes several fundamental activities and technical steps for the development of AUTOSAR projects [KF09]. Therefore, it specifies and describes the inputs and dependencies of the activities and the resulting work products [Heb09]. The goal of the AUTOSAR methodology is to parallelize parts of the development work and reduce the required development time. For this purpose, the development of the overall system is realized by means of different views. These different views are utilized to coordinate the work of the different involved partners and suppliers. Figure 5.4 depicts an abstract model of the different AUTOSAR methodology views and their dependencies. A detailed description of the methodology and the corresponding activities and technical steps on the particular views is given in [AUT13b].

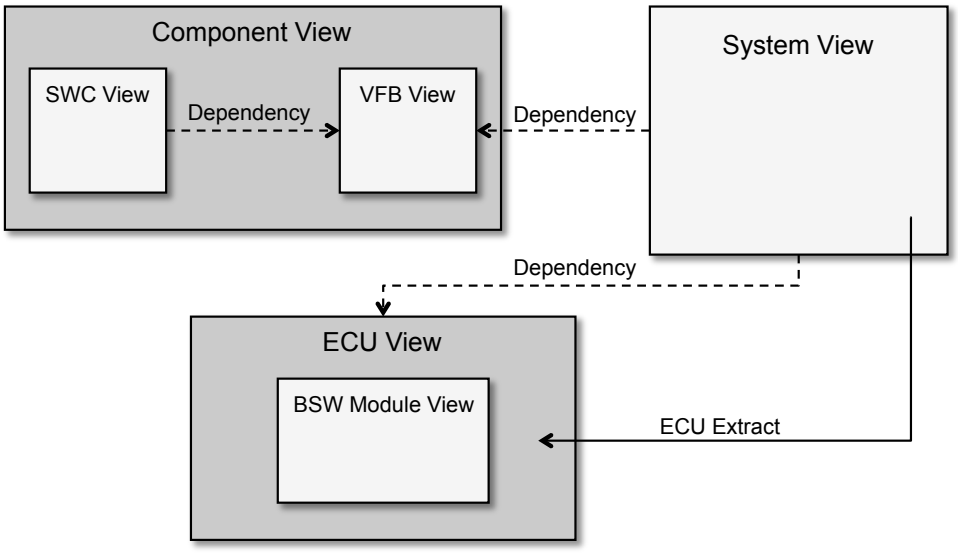


Figure 5.4: Different views of the AUTOSAR methodology based on [KF09].

Each view represents a part of the overall system at a certain time of the development process with a particular granularity. The dark grey boxes in Figure 5.4 define the application software on *Component View* including *SWC*

View and *VFB View*, as well as the hardware-dependent *ECU View* including the *BSW Module View*. They illustrate that there is no direct dependency between these views. This modularity enables an independent specification of the application software and the hardware-dependent software as parts of the overall system. This allows the parallel development of different system components by different partners at various points in time. However, as Figure 5.4 shows, there are several dependencies between different views which result from the exchange of work products for certain activities. In the following we briefly describe the individual views.

Component View

The Component View defines the application software by means of two views. The SWC View is used to describe each individual SWC and its internal behavior, whereas the VFB View specifies the software architecture of the system as a composition of the SWCs.

SWC View The SWC View specifies the SwcInternalBehavior of a SWC. As mentioned above only Atomic-SWCs have an internal behavior. The description of the SwcInternalBehavior defines which Runnables a SWC contains, how they are interconnected, and which data signals they exchange via read and write accesses. Furthermore, the real-time behavior of the Runnables is modeled by means of so-called *RteEvents*, i.e. triggering events like defined activation rates or data receptions.

AUTOSAR defines two different modes for the data signal exchange between Runnables, namely *explicit* and *implicit*. Depending on the chosen mode the corresponding RTE interface is generated to realize the data exchange.

Explicit: The Runnables communicate via explicit RTE API-calls. The receiver has a buffer which can be configured to store one or more input messages. Depending on the buffer size, two kinds of explicit communication are distinguished. If the buffer can store only one message this results in the ”last-is-best” behavior [KF09], i.e. data elements in the buffer can be overwritten by the sender or read multiple times by the receiver. If the buffer size is larger than one, data is written and read in the FIFO principle. This avoids that data elements are overwritten or read multiple times. The RTE also specifies error messages to indicate and handle full and empty buffers. Details on the RTE API functions and calls are given in [AUT13p].

Implicit: Specified data elements are also transmitted by the RTE but in contrast to the explicit mode, the RTE transmits the specified data after the termination of the sender and provides it before the start of the receiver.

This implies that during the execution of a Runnable the data keeps unchanged.

VFB View The VFB specifies the representation of the software architecture in an overall system by means of SWCs. As described in Section 5.1.2 Ports and Interfaces are utilized to interconnect SWCs for interaction. The VFB offers standardized communication mechanisms and services to model abstract and hardware-independent data exchange between SWCs [Fen+06]. Figure 5.5 depicts an example of three SWCs on VFB View communicating over Ports with Sender-Receiver Interfaces. In this example SWC_1 provides data for SWC_2 and SWC_3.

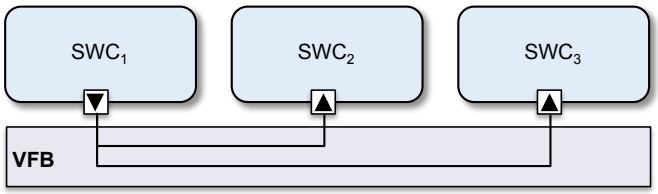


Figure 5.5: Example for communication modeling on VFB View.

As described in Section 5.1.1 the RTE manages the communication between SWCs and with the BSW. This means that in a specific implementation of a system on a hardware topology the generated RTE is responsible for the implementation of the communication defined on VFB View.

ECU View

According to the AUTOSAR methodology, the ECU View is extracted from the overall system description (cf. Figure 5.4). The configuration of an ECU starts with the splitting of the system description into several descriptions, whereas each contains all information about one single ECU. This *ECU extract* is the basis for the ECU Configuration step. Within the ECU Configuration process each single module of the AUTOSAR Architecture can be configured for the special needs of this ECU. Because of the quite complex AUTOSAR Architecture, modules, and their dependencies, this step is typically supported by tools [AUT13i]. In this step the SWCs have to be mapped to the different ECUs. This implies that the abstract communication dependencies specified on VFB View result in concrete communication links and the generation of the corresponding RTE on each ECU.

BSW Module View Since the BSW Module View is ECU-specific, it is beyond the scope of this thesis and not described in detail. However as part of the AUTOSAR methodology and for the sake of completeness we shortly describe its main properties. The individual ECU-specific BSW modules are

specified by means of the *BSWModuleDescriptionTemplate* [AUT13m]. Similar to the SWCs, here the internal behavior BswInternalBehavior including the BSW entities *BswExecutbaleEntity* is defined. An overview of the BSW modules and their individual specification options is given in [AUT13e] and [AUT13m].

System View

The System View specifies a system design with the given hardware topology and software architecture. This system design is the output of the design phase and corresponds to the AUTOSAR *System Template* [AUT13q]. The System View represents an interface to all other views (cf. Figure 5.4). It is utilized to comprise and manage the overall results of the development process. This contains the description of the software architecture, the hardware topology, and communication dependencies. Based on these descriptions, the System View depicts the mapping of SWCs defined in the VFB View to the given ECUs and the communication over the bus resulting from the performed system design.

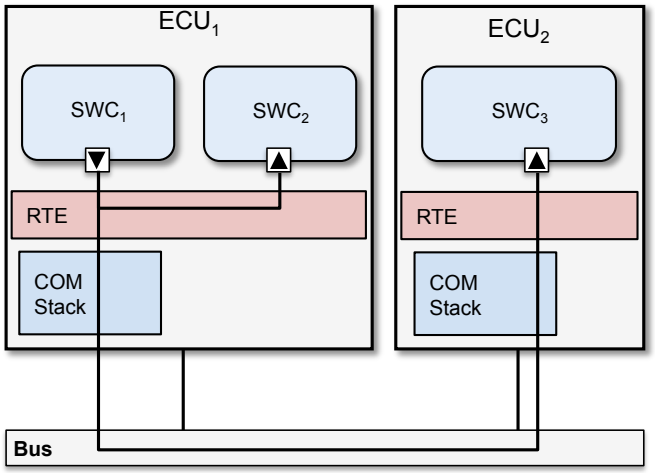


Figure 5.6: System View of an exemplary mapping of 3 SWCs to 2 ECUs.

Figure 5.6 depicts the resulting System View for an exemplary mapping of the SWCs from Figure 5.5 to a topology with 2 ECUs. In this example SWC_1 and SWC_2 are mapped to ECU_1 while SWC_3 is mapped to ECU_2. Figure 5.6 also illustrates the intra-ECU and inter-ECU communications resulting from the data dependencies between the SWCs specified on VFB View (cf. Figure 5.5). As described in Section 5.1, SWC_1 and SWC_2 are communicating directly via RTE while for the inter-ECU message transmission from SWC_1 to SWC_3 the RTE utilizes the Communication Stack. The next section provides more details on the communication in AUTOSAR.

5.1.4 AUTOSAR Communication

As described in the previous section, the data dependencies and communication between SWCs is specified on VFB View. The resulting communication on System View depends on the actual deployment of the SWCs on the ECUs of the given hardware topology. SWCs mapped to the same ECU communicate locally while SWCs on different ECUs communicate remotely over the network. In both cases, the RTE ensures the correct communication paths. As a result, the extent of utilized system components depends on the SWC distribution.

Figure 5.6 shows an example with local intra-ECU and remote inter-ECU communication. The inter-ECU data exchange between SWC_1 on ECU_1 and SWC_3 on ECU_2 goes through the communication bus which is accessed by the corresponding module of the COM respectively bus driver.[28] As shown in Figure 5.2(b), the communication drivers are provided by the AUTOSAR BSW as part of the Communication Stack. For an inter-ECU communication the RTE exchanges signals with the COM module which is part of the Communication Services. Signals are packed and unpacked by the COM module to *Interaction Layer Protocol Data Units* (I-PDUs) to be transmitted and received signals are provided to the RTE [Gos09]. The so-called *PDU Router* routes the I-PDUs between the Communication Services and the Hardware Abstraction Layer modules, i.e. the corresponding bus interface. The bus interface puts the I-PDUs into frames and prepares them for the transmission over the bus accessed via the corresponding bus driver. Further details on the COM module, the PDU Router, and the different bus drivers are given in [AUT13h] and [AUT13o].

5.1.5 AUTOSAR OS

AUTOSAR also specifies a real-time operating system called *AUTOSAR OS* [AUT13n]. AUTOSAR OS is a backwards compatible superset of the operating system *OSEK OS* [OSE05]. OSEK OS was specified within the open standard OSEK/VDX (Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen/Vehicle Distributed eXecutive) published by a consortium founded by the automotive industry [OSE13]. OSEK OS is an event-triggered real-time operating system. OSEK Time specifies a time-triggered extension for OSEK OS.

AUTOSAR OS reuses the OSEK specifications and also covers the functionality of the time-triggered OSEK Time [ZS07]. OSEK OS utilizes a pure event-driven priority-based multi-tasking concept. Time-triggering must be

[28] AUTOSAR supports several different bus protocols [AUT13h]. Here we focus on the real-time capable FlexRay bus.

implemented by alarms which often gets complex. Therefore, AUTOSAR OS introduces so-called *Schedule Tables* which are processed by means of corresponding OSEK counters. In the OS configuration it is specified which task is activated at which counter value. The process starts either automatically or is activated and stopped by means of calling the API-functions `StartScheduleTable...()` and `StopScheduleTable()`. An activation period can be set and the actual start time of a task can be delayed by an offset. Instead of a periodic repetition also a single processing of the Schedule Table is possible. Calling the API-function `NextScheduleTable()` allows switching between different Schedule Tables. Obviously, a Schedule Table has different states, e.g. NOT_STARTED and STARTED [AUT13n]. Figure 5.7 illustrates the Schedule Table concept with its properties.

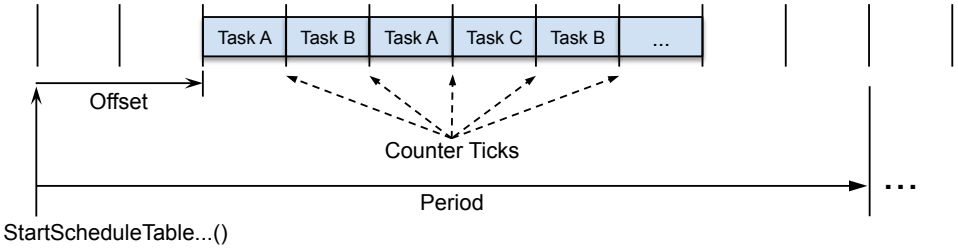


Figure 5.7: Schedule Table concept of AUTOSAR OS based on [ZS07].

The main idea of the Scheduling Table corresponds to the time-triggered scheduling concept in OSEK Time. In contrast to that in AUTOSAR OS the tasks activated by a Scheduling Table compete for the execution with other OS tasks. This means that a task is activated at the specified counter value but its actual start time still depends on its priority relative to the other activated tasks. Thus, to ensure a deterministic scheduling behavior, an appropriate priority assignment for the tasks is necessary [ZS07]. As mentioned in Section 4.2.4, Rate (Deadline) Monotonic Priority Assignment is the most widely applied approach in AUTOSAR OS systems [FFR12].

5.1.6 AUTOSAR Timing Extensions

In earlier releases of AUTOSAR timing was not properly addressed by the AUTOSAR specification. BMW Car IT developed an early prototype of an extension for the AUTOSAR model that enables the specification of timing constraints for a given system [BMW13]. Finally, various AUTOSAR partners developed and specified the so-called *AUTOSAR Timing Extensions* (ARTE) [AUT13k]. Some of the partners working on ARTE were also involved in the specification of TADL/TADL2 in the projects TIMMO and TIMMO-2-USE (cf. Section 2.2.4). Thus, the specification work on both concepts influenced each other. Since the introduction of the AUTOSAR Re-

lease 4.0.1, ARTE is part of the AUTOSAR specifications and enables one to express timing constraints in a standardized format [Per+12a].

Similar to TADL2, AUTOSAR timing models consist of timing descriptions – expressed by events and event chains – and timing constraints that are imposed on these events and event chains. Events refer to locations in AUTOSAR models at which the occurrences of events are observed [Per+12a]. ARTE defines a set of predefined event types and *Timings* related to one of the AUTOSAR methodology views described in Section 5.1.3 [AUT13k]:

- VFB Timing,
- SWC Timing,
- System Timing,
- BSW Module Timing, and
- ECU Timing.

In particular, these views allow to specify the reading and writing of data from and to specific ports of software components, calling of services, and receiving their responses on VFB Timing; the sending and receiving of data via networks and through Communication Stacks on System Timing; activating, starting, and terminating of executable entities on SWC Timing and calling of ECU-specific BSW services and receiving their responses on ECU Timing and BSW Module Timing [Per+12a].

In ARTE, similar to TADL2, event chains specify a causal relationship between events and their temporal occurrences (cf. Section 2.2.4). Event chains allow to specify the relationship between two events, for example when an event (stimulus) A occurs then the event (response) B occurs, or in other words, the event B occurs if and only if the event A occurred before. Event chains can be composed of existing event chains and decomposed into so-called event chain segments. Events also describe at which locations in this system the occurrences are observed.

ARTE offers several timing constraints. An *event triggering constraint* imposes a constraint on the occurrences of an event, which means that it specifies the way an event occurs in time. ARTE provides means to specify periodic and sporadic event occurrences, as well as event occurrences that follow a specific pattern [AUT13k]. The ARTE *latency* and *synchronization timing constraints* impose constraints on event chains. In the former case, a constraint is used to specify a reaction and age, for instance if a stimulus event occurs then the corresponding response event shall occur not later than

a given amount of time. Thus the ARTE latency respectively reaction constraint corresponds to the TADL2 delay constraint. In the latter case, the constraint is used to specify that stimuli or response events must occur within a given time interval to be said to occur simultaneous and synchronous respectively [Per+12a].

As mentioned in Section 2.2.4, we focus on the periodic constraint and particularly on the delay respectively latency constraint. Furthermore, we focus on VFB, SWC, and System Timing because the ECU-specific views are beyond the scope of this thesis (cf. Section 5.1.3).

The VFB Timing is applicable for different system granularities. The smallest granularity is the investigation of a single SWC without any contextual embedding. Here, a timing description can only refer to relations between the RPort and the PPort of the same component [AUT13k].

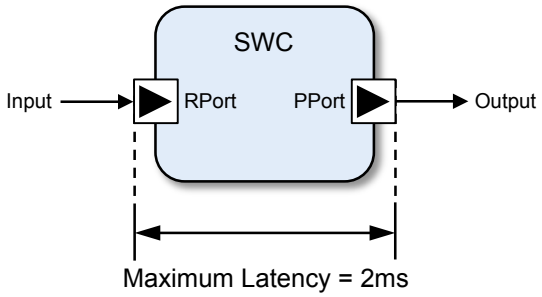


Figure 5.8: Example for a latency constraint on VFB View [AUT13k].

Figure 5.8 depicts an example latency constraint for a single SWC on VFB View [AUT13k]:

> "*From the point in time, where the value input is received by the SWC, until the point in time, where the newly calculated value output is sent, there shall be a maximum latency of 2 ms.*"

This latency constraint would be attached to the timing description that refers to the SWC.

The System Timing is used to annotate timing information to a system description. As the system description aggregates all the information about SWCs and their corresponding internal behavior, it is possible to use the same concepts that are available on VFB Timing and SWC Timing. The difference is the specific system context that defines the validity of timing information at System View. Without knowledge of the mapping of SWCs to specific ECUs, only a generic platform independent description can be provided. Moreover, System Timing refers to the concrete communication of SWCs that only was represented as abstract connectors in VFB Timing. As described in Section 5.1.4, depending on the performed deployment, commu-

nication is performed locally over the RTE or remotely over the RTE, through the Communication Stack of the BSW and a communication bus. A system-specific timing description thus can refer to signals (RTE), I-PDUs (COM), and frames (communication driver and bus) [AUT13k]. Figure 5.6 depicts possible data flows in the scope of System Timing which can be annotated with timing constraints for the corresponding events and event chains.

5.2 Fault-Tolerant Deployment of Real-Time Software in AUTOSAR Networks

In the previous sections we gave a briefly introduction to AUTOSAR. In this section we present the application of our approach to the design of automotive systems and the deployment of real-time software based on AUTOSAR. According to the design flow steps shown in Figure 1.2 the system design contains (i) the initial definition and modeling of the given software architecture and hardware topology and the corresponding interdependent (ii) Runnable respectively task mappings and (iii) bus mappings. In Section 4.1 we extended and refined these design flow steps for our fault-tolerant design approach (cf. Figure 4.1).

The inputs for our approach are AUTOSAR-based models and descriptions of the given application software architecture and hardware topology. The application software is derived from the intended function or set of functions controlled by the distributed system and modeled on VFB View. As described in Section 4.2.5, the given application software architecture implies initial constraints and requirements for the hardware topology configuration including the setup of basic bus parameters. Here, we consider the real-time capable FlexRay bus protocol supported by AUTOSAR. The corresponding timing constraints for the executed functions to guarantee a correct system behavior of each configuration with respect to the given hard real-time constraints are specified and annotated to the models by means of the AUTOSAR Timing Extensions.

Based on this input our design approach performs the actual system deployment. This comprises the determination of Runnable respectively task and bus mapping as well as corresponding schedulings for the initial configuration and based on that for all necessary reconfigurations to design a fault-tolerant system. As final design result the approach returns the determined software deployment for the automotive system design. As described in Section 5.1.3, this system design with the resulting software mappings and communication dependencies modeled on System View is the output of the design phase.

Beside the feasible AUTOSAR-compliant software deployment for a fault-tolerant system, the appropriate reconfiguration in case of an ECU-failure has to be performed. This includes the detection of the failed node and the activation of the corresponding redundant tasks within the ECU network. Thus, we also present a reconfiguration concept based on existing AUTOSAR BSW modules.

The following sections describe in detail the integration of our design approach and reconfiguration concept to the AUTOSAR Methodology and Software Architecture. For the fault-tolerant deployment we provide modified versions of our algorithms presented in Section 4.4. Since for these modified algorithms only the representation of the input changes and the performed operations are the same, their time complexity remains unchanged. As a case study, all steps are directly applied to real-world applications we initially presented in [Klo+13].

5.2.1 Modeling of Application Software

In AUTOSAR the functionality of application software is structured and modeled with interconnected SWCs. This modeling contains the specification of communication dependencies and interfaces for distributed components as well as the representation of their functional internal behavior by means of Runnables (cf. Section 5.1.2).

Similar to the task set in Equation 4.22, the set of n Runnables $\mathcal{R}$ in a AUTOSAR system can be modeled as:

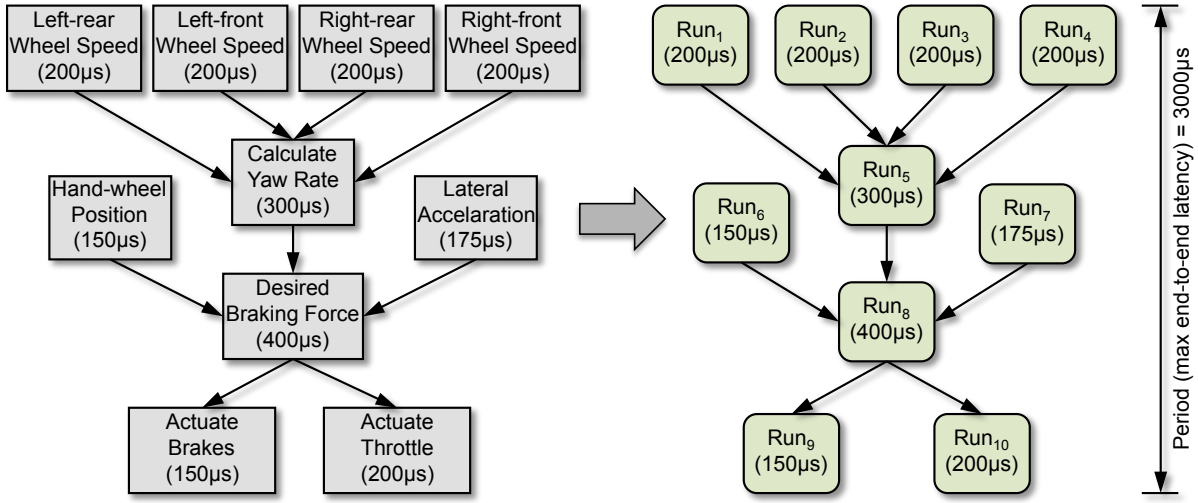
$$\mathcal{R} = \{\text{Run}_i = (T_i, C_i, r_i, d_i, s_i, f_i) \,|\, i = 1, \ldots, n\}\,. \tag{5.1}$$

Each Runnable Run_i is characterized by its period T_i, WCET C_i, release time r_i and deadline d_i. These parameters are globally defined and constrain the possible execution of the given Runnable to guarantee a correct system behavior. Consequently, each Runnable is also described by a start time s_i and a finishing time f_i which must lie within its available execution time interval:

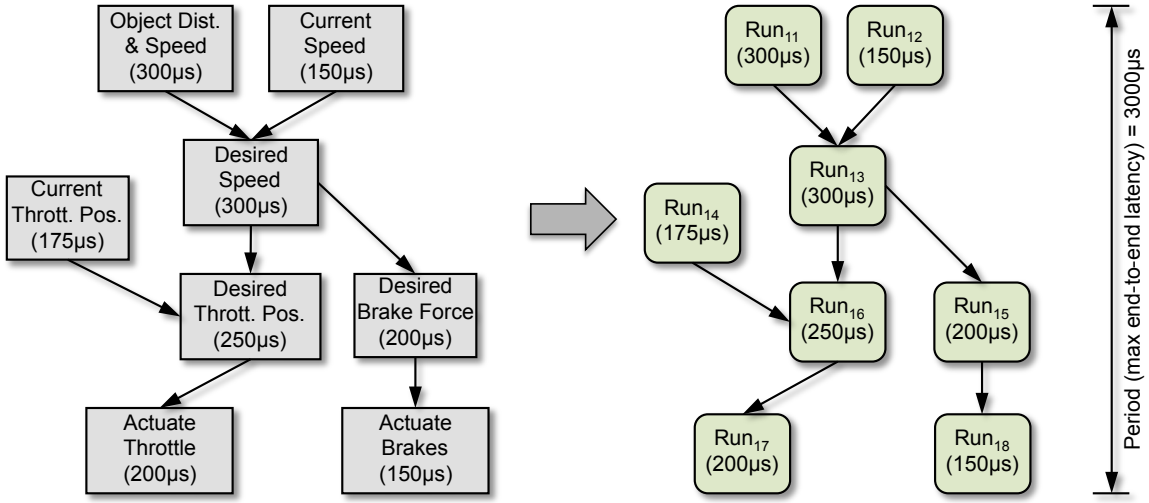
$$r_i \leq s_i < f_i \leq d_i.$$

Figure 5.9 illustrates the functional components of a Traction Control (TC) and an Adaptive Cruise Control (ACC) system model from [KHM03]. TC and ACC are typical examples for distributed real-time systems in the automotive domain. The TC system is an extension of the Antilock Braking System (ABS) [Rob06]. It actively stabilizes the vehicle to maintain its intended path even under smooth or slippery road conditions and the ACC system auto-

matically maintains a safe following distance to a detected vehicle in front.[29] These applications demand timely data exchange between distributed sensors, processors, and actuators, i.e., have specific end-to-end deadlines and therefore require a dependable distributed system [KHM03].



(a) Traction Control (TC).



(b) Adaptive Cruise Control (ACC).

Figure 5.9: Functional components and corresponding Runnables of a TC and an ACC system with their WCETs and data dependencies [KHM03].

Figure 5.9 depicts the individual components of these two applications with their data dependencies and provides information about their timing requirements and properties, i.e. the WCETs and the common period of $T_i = 3000\mu s$ they are triggered at. Furthermore, it shows the corresponding modeling of

[29] More technical details and descriptions on the TC and the ACC system can be found in [Rob06] and [Rob03].

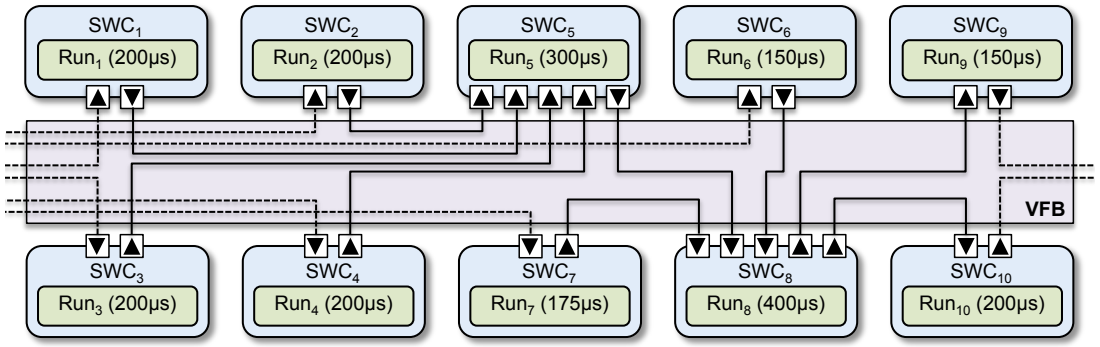
the components by means of Runnables. Each functional component is represented by a separate Runnable.

The TC system in Figure 5.9(a) is composed of ten Runnables. The four Runnables Run_1 to Run_4 represent the data acquisition and processing for the current speed of the four vehicle wheels. Based on these input data Run_5 calculates the current yaw rate of the car. Run_8 determines the desired breaking force for the individual wheel brakes by comparing the yaw rate with the current hand-wheel position and lateral acceleration data aquired and processed by Run_6 and Run_7. Finally, the calculated desired braking force is the input for Run_9 and Run_{10} to process and provide the corresponding data values for the brake actuators and the throttle actuator.

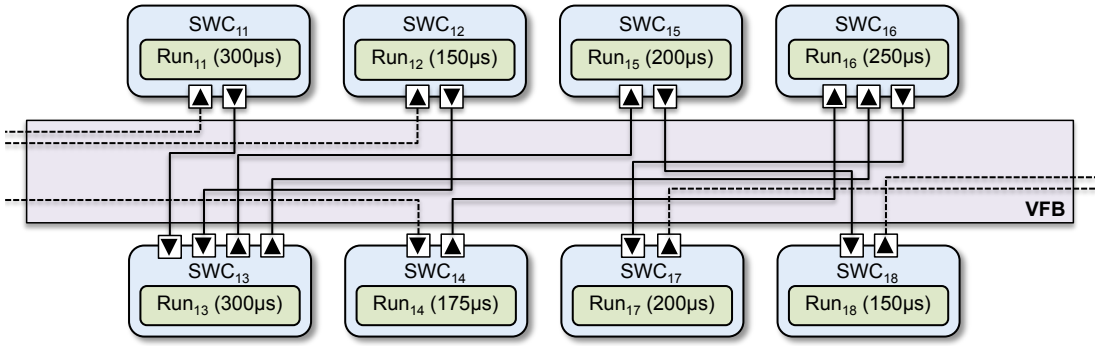
The ACC system in Figure 5.9(b) is composed of eight Runnables. Run_{11} performs the data acquisition and processing of distance and speed for the currently detected vehicle in front of the car while Run_{12} determines the current speed of the car itself. A comparison of these input values allows Run_{13} to calculate the desired speed for the car. The output of Run_{13} is used as input data by Run_{16} and Run_{15}. Run_{16} compares the desired speed value with the current throttle position acquired and provided by Run_{14} to determine the resulting desired throttle position while Run_{15} calculates the desired brake force if necessary. Finally, Run_{17} and Run_{18} process and provide the corresponding data values for the throttle actuator and brake actuators.

The structural modeling of the application software in AUTOSAR implies placement constraints because all Runnables of an Atomic-SWC must be mapped to the same ECU (cf. Section 5.1.2). As we mentioned before, we want to avoid all placement constraints to maximize the flexibility for the deployment in our fault-tolerant design approach. Thus, for an appropriate application software modeling we put each Runnable into a separate SWC to enable the required mapping of each Runnable to an arbitrary ECU. Consequently, in the following we use the terms Runnable-to-ECU and SWC-to-ECU mapping as synonyms.

Based on this design decision, Figure 5.10 shows the resulting model of the TC and ACC System on VFB View. The VFB allows to model abstract and hardware-independent data exchange between SWCs. Thus, this AUTOSAR-based model represents the given application software architecture with its communication dependencies independent from the given hardware architecture and acts as input for the deployment in our fault-tolerant design approach illustrated in Figure 4.1. The VFB View model in Figure 5.10 depicts that the SWCs of the TC and ACC system are connected by Ports with Sender-Receiver Interfaces utilizing the corresponding communication paradigm. Moreover, we utilize the implicit communication mode for the data signal exchange between the Runnables (cf. Section 5.1.3). Dashed lines represent connections to peripheral interfaces.



(a) VFB View model of TC system.



(b) VFB View model of ACC system.

Figure 5.10: VFB View models of TC and ACC system in Figure 5.9.

Obviously, the communication dependencies between the Runnables imply order and precedence constraints. However, to guarantee a correct system behavior all Runnables must fulfill their timing constraints. In Section 4.2.3 we specified, like the authors in [DTT08], that the execution period also constraints the maximum end-to-end delay respectively latency of the application software components. Since we consider a one-to-one mapping of Runnables to SWCs, the defined timing constraints are identical for a SWC and its contained Runnable. For the examples in Figure 5.9 this results in an maximum end-to-end latency of 3000μs for the Runnables respectively SWCs of the TC and ACC system.

AUTOSAR Timing Extensions are used to annotate timing constraints to the VFB View model by means of events and event chains (cf. Section 5.1.6). Figure 5.11 depicts a maximum end-to-end latency constraint for the event chain $SWC_1(Run_1) \rightarrow SWC_5(Run_5) \rightarrow SWC_8(Run_8) \rightarrow SWC_{10}(Run_{10})$ of the TC system. This timing constraint defines that the delay between the stimulus, i.e. the input on the RPort of SWC_1, and the response, i.e. the output on the PPort of SWC_{10}, must not exceed the given maximum end-to-end latency of 3000μs. Analogous, timing constraints are defined for each end-to-end

event chain of the system, i.e. from SWCs with Runnables in $\mathcal{R}_{\text{in}}$ to SWCs with Runnables in $\mathcal{R}_{\text{out}}$.

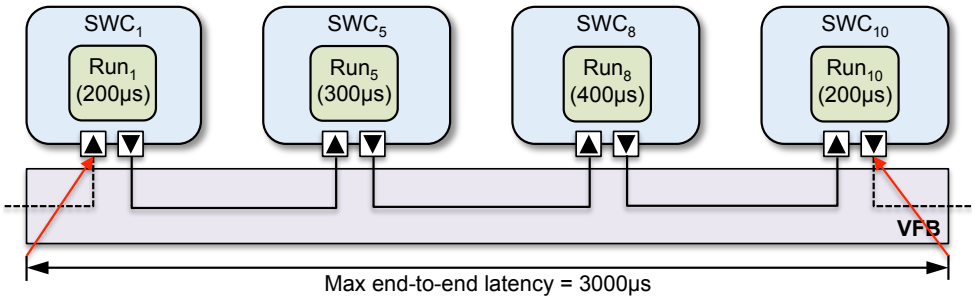


Figure 5.11: VFB Timing event chain with latency constraint for TC system.

Similar to Equation 4.24 and 4.25, based on the timing and precedence constraints the available execution time interval $\Psi_i = [r_i, d_i[$ for each Runnable Run_i can be calculated.

The corresponding release time r_i is given by:

$$r_i = \begin{cases} 0 & \text{, if } \text{Run}_i \in \mathcal{R}_{\text{in}} \\ \max\{r_j + C_j \,|\, \text{Run}_j \in \mathcal{R}_{\text{directPre}_i}\} & \text{, else.} \end{cases} \tag{5.2}$$

If a Runnable has no predecessors, i.e. $\text{Run}_i \in \mathcal{R}_{\text{in}}$, its available execution time interval of Run_i starts at $r_i = 0$. Otherwise r_i is calculated by means of r_j and C_j of the direct predecessors, i.e. $\text{Run}_j \in \mathcal{R}_{\text{directPre}_i}$. The deadline of Run_i is calculated as:

$$d_i = \begin{cases} \text{max end-to-end delay} & \text{, if } \text{Run}_i \in \mathcal{R}_{\text{out}} \\ \min\{d_k - C_k \,|\, \text{Run}_k \in \mathcal{R}_{\text{directSucc}_i}\} & \text{, else.} \end{cases} \tag{5.3}$$

If a Runnable has no successors, i.e. $\text{Run}_i \in \mathcal{R}_{\text{out}}$, the available execution time interval of Run_i ends at $d_i =$ max end-to-end latency. Otherwise, d_i depends on d_k and C_k of the direct successors of Run_i, i.e. $\text{Run}_k \in \mathcal{R}_{\text{directSucc}_i}$.[30]

Table 5.1 summarizes the calculated available execution intervals Ψ_i for the Runnables of the TC and ACC system. For instance, Run_5 has the release time $r_5 = 0\mu s + 200\mu s = 200\mu s$ and the deadline $d_5 = 3000\mu s - 200\mu s - 400\mu s = 2400\mu s$.

Beside the properties of the Runnables, also the properties of the exchanged data messages have to be considered to enable a feasible deployment and

[30] In fact, a designer could also specify earlier deadlines for runnables. However, this it not reasonable because it would result in an over-specification without benefits for the timeliness of the functionality.

TC System				ACC System			
Run_i	C_i [μs]	r_i [μs]	d_i [μs]	Run_i	C_i [μs]	r_i [μs]	d_i [μs]
Run_1	200	0	2100	Run_{11}	300	0	2350
Run_2	200	0	2100	Run_{12}	150	0	2350
Run_3	200	0	2100	Run_{13}	300	300	2650
Run_4	200	0	2100	Run_{14}	175	0	2550
Run_5	300	200	2400	Run_{15}	200	600	2850
Run_6	150	0	2400	Run_{16}	250	600	2800
Run_7	175	0	2400	Run_{17}	200	850	3000
Run_8	400	500	2800	Run_{18}	150	800	3000
Run_9	150	900	3000				
Run_{10}	200	900	3000				

Table 5.1: Runnable properties for TC and ACC systems shown in Figure 5.9.

guarantee a correct system behavior. Analogous to Equation 4.23 here also a set $\mathcal{M}$ of n real-time messages can be modeled as:

$$\mathcal{M} = \{m_i = (T_{m_i}, L_{m_i}, r_{m_i}, d_{m_i}, s_{m_i}, f_{m_i}) \,|\, i = 1, \ldots, n\}\,. \tag{5.4}$$

Equation 5.4 defines that each message m_i has a period T_{m_i} which directly depends on the period T_{tx} of the sender Runnable Run_{tx} and the period T_{rx} of the receiver Runnable Run_{rx} (cf. Section 4.2.3). The data length L_{m_i} defines how much data message m_i has to transmit. Table 5.2 summarizes the messages which are exchanged by the Runnables in the TC and ACC system and provides their individual sizes given in [KHM03].

TC System			ACC System		
m_i	$m_{(\text{Run}_{\text{tx}},\text{Run}_{\text{rx}})}$	L_{m_i} [bits]	m_i	$m_{(\text{Run}_{\text{tx}},\text{Run}_{\text{rx}})}$	L_{m_i} [bits]
m_1	$m_{(\text{Run}_1,\text{Run}_5)}$	12	m_{10}	$m_{(\text{Run}_{11},\text{Run}_{13})}$	12
m_2	$m_{(\text{Run}_2,\text{Run}_5)}$	12	m_{11}	$m_{(\text{Run}_{12},\text{Run}_{13})}$	12
m_3	$m_{(\text{Run}_3,\text{Run}_5)}$	12	m_{12}	$m_{(\text{Run}_{14},\text{Run}_{16})}$	10
m_4	$m_{(\text{Run}_4,\text{Run}_5)}$	12	m_{13}	$m_{(\text{Run}_{13},\text{Run}_{16})}$	12
m_5	$m_{(\text{Run}_6,\text{Run}_8)}$	10	m_{14}	$m_{(\text{Run}_{13},\text{Run}_{15})}$	12
m_6	$m_{(\text{Run}_5,\text{Run}_8)}$	22	m_{15}	$m_{(\text{Run}_{16},\text{Run}_{17})}$	10
m_7	$m_{(\text{Run}_7,\text{Run}_8)}$	20	m_{16}	$m_{(\text{Run}_{15},\text{Run}_{18})}$	10
m_8	$m_{(\text{Run}_8,\text{Run}_9)}$	12			
m_9	$m_{(\text{Run}_8,\text{Run}_{10})}$	12			

Table 5.2: Message properties for TC and ACC systems shown in Figure 5.9 [KHM03].

Furthermore, each real-time message m_i has a release time r_{m_i}, and a deadline d_{m_i}. Since we consider the implicit communication mode, the transmission of message m_i can be started earliest when Run_{tx} has finished its computation

and must be finished latest when Run_{rx} is started. Thus, as described in Section 2.2.2, we define that r_{m_i} and d_{m_i} limit the available transmission time interval $\Upsilon_{m_i} = [r_{m_i}, d_{m_i}[= [f_{\text{tx}}, s_{\text{rx}}[$ of a message m_i. Consequently, the start time s_{m_i} and finishing time f_{m_i} of m_i must lie within this interval:

$$f_{\text{tx}} = r_{m_i} \leq s_{m_i} < f_{m_i} \leq d_{m_i} = s_{\text{rx}}.$$

Obviously, the available transmission time interval Υ_{m_i} of each message m_i depends on the performed Runnable deployment and the resulting scheduling on the ECUs.

5.2.2 Modelling and Setup of Hardware Architecture

Beside the application software model, our AUTOSAR-based approach for the fault-tolerant deployment of real-time software also requires information about the given hardware architecture as input (cf. Figure 4.1). Basically, this contains the number and properties of the networked ECUs and the configuration of the communication bus, here the FlexRay bus.

Analogous to Equation 4.10, the set $\mathcal{E}$ of n ECUs in an automotive system can be defined as:

$$\mathcal{E} = \{\text{ECU}_j | \; j = 1, \ldots, n\}. \tag{5.5}$$

The TC and ACC system, which we consider as examples here, are realized with a total of three ECUs in a typical current vehicle network [Rob06]. The TC system is implemented with two ECUs which are directly connected to their corresponding sensors and actuators. The TC system ECU is connected to the four vehicle wheels and additional sensors to determine the current wheel speeds, yaw rate, lateral acceleration, and hand-wheel position. Based on these information, it calculates the desired braking force to process and provide the corresponding output data to the individual wheel brake actuators. Additionally, it transmits the necessary information to the engine management ECU of the vehicle which processes and provides the output data to the connected throttle actuator [Rob06]. The ACC system utilizes one ECU which is directly connected to the distance measurement sensor, e.g. radar or laser setup. This ECU calculates the desired speed by means of the detected vehicle in front and the current speed. The required information about the current speed comes from the wheel-speed sensors via TC system ECU or from specific engine sensors via engine management ECU. These ECUs get the desired speed value as feedback to calculate the resulting throttle position respectively brake force and transmit the output data to the corresponding actuators [Rob03].

The ECUs described above are hardwired to the mentioned sensors and actuators in a current vehicle network. This results in I/O-related placement

constraints for the application software because the specific Sensor/Actuator-SWCs and their Runnables must be mapped to the ECU with the corresponding sensors and actuators (cf. Section 5.1.2). Considering these placement constraints, a fault-tolerant system design for the compensation of an arbitrary ECU failure implies one redundant component per ECU. For the described system this would result in a total of six ECUs. Therefore, we utilize the reconfigurable network topology presented in Section 4.2.1 for the software deployment to avoid these placement constraints.

In Section 4.2.5 we described how we determined the lower limit of required functional nodes for the reconfigurable network topology by means of the utilization of the ECUs. As mentioned above the application software of the TC and ACC system is composed of 18 Runnables with a common period of $T_i = 3000\mu s$. Considering RM respectively DM priority assignment, this harmonic set allows a maximum utiliaztion per ECU of $U_{\text{lub}} = 1.0$. The summed utilization of these Runnables is $U = 3900/3000 = 1.3$. Inserting these values in Equation 4.14, we can determine the lower limit of required ECUs as $m = \lceil 1.3 \rceil + 1 = 3$. Here, we consider a homogeneous network structure. Hence, the Runnable WCETs provided in Table 5.1 are valid for all functional nodes. Nevertheless, our approach also supports heterogeneous ECUs as described in Section 4.2.5.

Figure 5.12 illustrates the System View model of the hardware architecture for the TC and ACC system as input for our AUTOSAR-based design approach enabling a flexible and reconfigurable deployment without placement constraints.

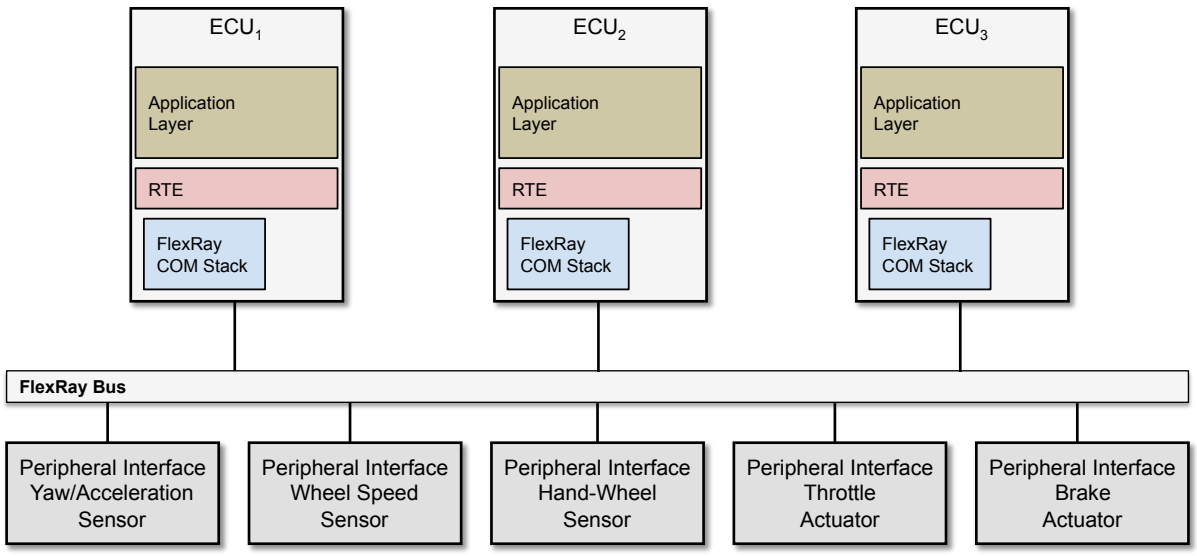


Figure 5.12: System View of reconfigurable hardware architecture for TC and ACC system.

It is composed of the required functional nodes ECU_1, ECU_2, ECU_3, and the peripheral interfaces to the involved sensors and actuators described above.

As described in Section 4.2.1, we focus on the three ECUs that are utilized to enable dynamic redundancy with reconfiguration by means of Cold Standby replicas. However, it is important to remember that the peripheral interfaces provide the required connection to the I/O of the system. These nodes convert I/O signals to FlexRay bus messages and vice versa to exchange these data with the Runnables in $\mathcal{R}_{\text{in}}$ and $\mathcal{R}_{\text{out}}$. Since peripheral interfaces do not execute any other complex function, they only require low hardware capacities which allows cost-efficient static hardware redundancy. AUTOSAR defines specific Sensor/Actuator-SWCs for the data exchange with the I/O (cf. Section 5.1.2). Thus, in an AUTOSAR-based system these SWCs are deployed on the corresponding peripheral interfaces.

FlexRay Bus Setup

The configuration and properties of the communication bus has a big impact on the performed message mapping and scheduling. Thus, as described in Section 4.2.5, we perform the configuration of the FlexRay bus and the setup of its parameters according to the properties of the given application software architecture. This means that we set the length of the communication cycle to the hyperperiod of the Runnable set:

$$\text{length}(\Theta_{\text{cycle}}) = T_{\text{max}} = 3000\mu s.$$

Furthermore, we analyze the number and size of the messages exchanged by the Runnables. Table 5.2 shows that there is a total of 16 messages exchanged in the TC and ACC system. It also shows that the individual message sizes are 10 to 22 bits. Consequently, a message requires one or two words of a FlexRay frame. For this example, we set the slot duration to $\text{duration}(\theta) = 25\mu s$ which allows a maximum payload of 12 bytes respectively six two-byte words in one slot (cf. Section 2.2.3). This allows frame packing to combine several messages if applicable. Even considering preassigned slots for the communication with the peripheral interfaces, as described in Section 4.3.2, there remain more than 100 slots in the FlexRay Cycle for the inter-ECU message transfer between the functional nodes.[31] However, our deployment approach is intended to reduce the inter-ECU messages and the resulting number of required slots by means of applying slot reuse and frame packing to determine a feasible and efficient bus mapping.

[31] According to Equation 4.19: $\left|\Theta_{\text{cycle}}\right| = \text{length}(\Theta_{\text{cycle}})/\text{duration}(\theta) = 3000\mu s/25\mu s = 120$.

5.2.3 Runnable and Task Mapping

As a first step our fault-tolerant AUTOSAR-based deployment of real-time software includes feasible mappings of SWCs respectively Runnables to the available ECUs in the network. Thus, our approach utilizes the VFB View model with the Runnables and their communication dependencies and timing constraints (cf. Figure 5.10 and 5.11) and the initial System View model of the given hardware architecture (cf. Figure 5.12) as input to perform these Runnable-to-ECU mappings. As described in Section 4.2, the objective of our approach is to determine a feasible combined solution for an initial software deployment and all necessary reconfigurations and replications for the remaining nodes of the network in case of a node failure. Thus, each configuration has to fulfill the deadlines of all Runnables and the end-to-end latency constraints for all event chains. Therefore, our approach iteratively analyzes and reduces the resulting execution delay for each Runnable-to-ECU mapping to finally ensure shortened end-to-end delays for all event chains.

Initial Runnable Mapping

It starts with the initial mapping $M_{init}^{Run} : \mathcal{R} \mapsto \mathcal{E}$ described by pseusdocode in Algorithm 7 (INITIALTASKMAPPING) which is a modified version of Algorithm 1. As described above, the AUTOSAR methodology strictly specifies the representation of the software architecture and the hardware topology. Hence, instead of the TDG, the HAG, and the COMG the modified algorithm gets the set $\mathcal{R}$ of Runnables , the set $\mathcal{E}$ of ECUs, and the set Θ of FlexRay slots as input. Nevertheless, the performed operations of the algorithm remain the same.

The algorithm defines the mapping order of the Runnables via sorting them by their deadlines and release times (cf. Table 5.1). Before each mapping a schedulability test has to determine the feasible candidate ECUs in $\mathcal{E}$. Since we consider DM scheduling this, is solved by our extended response time analysis RTA* presented in Section 4.2.4. The initial mapping begins with the Runnables, which have no precedence constraints ($\mathcal{R}_{in}$), and maps them iteratively to the ECU hosting the last Runnable with the earliest finishing time. Considering a network with n ECUs, this implies that the first n Runnables will be mapped to empty ECUs.

For Runnables with predecessors the appropriate Runnable-to-ECU mapping is more complex. Hence, Algorithm 8 (GETMINIMUMRUNNABLEDELAY) returns the ECU with the minimum execution delay for each $Run_i \in \mathcal{R} \setminus \mathcal{R}_{in}$. This algorithm is a modification of Algorithm 2 and gets a Runnable Run_i as input instead of a task τ_i. It determines the direct predecessors of Run_i, their hosting ECUs $\mathcal{E}_{pre}$, and the last Runnables on these ECUs ($\mathcal{R}_{last}$). If one

Algorithm 7 INITIALRUNNABLEMAPPING

Input: Set of Runnables $\mathcal{R}$, ECUs $\mathcal{E}$, and slots Θ.
Output: RunnableMapping: $M_{init}^{Run} : \mathcal{R} \mapsto \mathcal{E}$.

```
1: SORTBYDEADLINEANDRELEASETIME(R)
2: for all Run_i ∈ R_in do
3:    E_tmp ← GETFEASIBLEECUS(Run_i, E)
4:    ECU_tmp ← GETEARLIESTFINISH(E_tmp)
5:    MAPRUNNABLE(Run_i, ECU_tmp)
6:    UPDATESTARTANDFINISHTIMES(R)
7: end for
8: for all Run_i ∈ R \ R_in do
9:    E_tmp ← GETFEASIBLEECUS(Run_i, E, Θ)
10:   ECU_tmp ← GETMINIMUMRUNNABLEDELAY(Run_i, E_tmp, Θ)
11:   MAPRUNNABLE(Run_i, ECU_tmp)
12:   UPDATESTARTANDFINISHTIMES(R)
13:   UPDATEAVAILABLESLOTS(Θ)
14: end for
15: return RunnableMapping: M_init^Run : R ↦ E
```

or more of the direct predecessors of Run_i are last Runnables, the algorithm maps Run_i to the same ECU as the predecessor with the latest finishing time. This results in the minimum possible delay for Run_i, because it avoids additional inter-ECU communication delay for the latest input of Run_i. If there are Runnables mapped to $\mathcal{E}_{pre}$ after all direct predecessors, the Inter-ECU communication for input to Run_i can take place during their execution. In this case the algorithm determines the ECU_{preMin} with the earliest finishing time. If there are ECUs that do not host any of the direct predecessors of Run_i, the one with the earliest finishing time ($ECU_{nonPreMin}$) is also considered. The algorithm compares the resulting execution delays and returns the ECU resulting in the shorter delay. More details on our algorithms for the initial mapping and the delay determination are given in Section 4.4.1.

Redundancy Runnable Mapping

By means of Algorithm 7 and Algorithm 8, our deployment approach determines a feasible initial mapping with minimized execution delays considering timing, order, and precedence constraints for the Runnables. However, in a network with n ECUs it has to perform n redundancy mappings to enable the compensation of each possible ECU failure. Thus, Algorithm 9 (REDUNDANCYRUNNABLEMAPPING) calculates the redundancy mapping $M_{red}^{Run} : \mathcal{R} \mapsto \mathcal{E}$ for a Runnable set of a failed ECU to the remaining ECUs. Because of the AUTOSAR-specific input representation, this modified version of Algorithm 3 gets an initial Runnable mapping M_{init}^{Run} and the set of

Algorithm 8 GETMINIMUMRUNNABLEDELAY

Input: Runnable Run_i, set of candidate ECUs $\mathcal{E}_{\text{tmp}}$, and available slots Θ.
Output: ECU with minimum execution delay.

1: $\mathcal{R}_{\text{directPre}} \leftarrow$ GETDIRECTPREDECESSORS(Run_i)
2: $\mathcal{E}_{\text{pre}} \leftarrow$ GETHOSTECUS($\mathcal{R}_{\text{directPre}}, \mathcal{E}_{\text{tmp}}$)
3: $\mathcal{R}_{\text{last}} \leftarrow$ GETLASTRUNNABLES($\mathcal{E}_{\text{pre}}$)
4: $\mathcal{R}_{\text{cap}} \leftarrow \mathcal{R}_{\text{last}} \cap \mathcal{R}_{\text{directPre}}$
5: **if** $\mathcal{R}_{\text{cap}} \neq \emptyset$ **then**
6: ECU $\leftarrow$ GETHOSTECU(GETLATESTFINISH($\mathcal{R}_{\text{cap}}, \mathcal{E}_{\text{tmp}}$)
7: **else**
8: $\text{ECU}_{\text{preMin}} \leftarrow$ GETEARLIESTFINISHANDSHORTESTCOMDELAY($\mathcal{E}_{\text{pre}}, \Theta$)
9: **if** $\mathcal{E}_{\text{tmp}} \setminus \mathcal{E}_{\text{pre}} \neq \emptyset$ **then**
10: $\text{ECU}_{\text{nonPreMin}} \leftarrow$ GETEARLIESTFINISHANDSHORTESTCOMDELAY($\mathcal{E}_{\text{tmp}} \setminus \mathcal{E}_{\text{pre}}, \Theta$)
11: $\Delta \leftarrow$ DIFF$\langle$GETDELAY($\text{ECU}_{\text{preMin}}, \Theta$), GETDELAY($\text{ECU}_{\text{nonPreMin}}, \Theta$)$\rangle$
12: **if** $\Delta > 0$ **then**
13: ECU $\leftarrow \text{ECU}_{\text{nonPreMin}}$
14: **else**
15: ECU $\leftarrow \text{ECU}_{\text{preMin}}$
16: **end if**
17: **else**
18: ECU $\leftarrow \text{ECU}_{\text{preMin}}$
19: **end if**
20: **end if**
21: **return** ECU

FlexRay slots Θ instead of a TMG and a COMG as input. However, the performed operations are the same as in Algorithm 3.

Initially Algorithm 9 determines the Runnable set $\mathcal{R}_{\text{fail}}$ which was executed on the failed ECU by means of the initial mapping $\text{M}_{\text{init}}^{\text{Run}}$ and the remaining ECUs $\mathcal{E}_{\text{rem}}$. Moreover, it initializes the redundancy mapping $\text{M}_{\text{red}}^{\text{Run}}$ with the Runnable mapping of the remaining ECUs kept from $\text{M}_{\text{init}}^{\text{Run}}$. This allows to combine Runnables to AUTOSAR OS tasks and the reuse of messages and slots in different reconfigurations. Similar to the initial mapping, the algorithm iteratively maps the Runnables from $\mathcal{R}_{\text{fail}}$. Hence, in each mapping step the redundancy mapping M_{red} is complemented by the currently performed mapping. Obviously, the feasible candidate ECUs $\mathcal{E}_{\text{tmp}}$ have to be determined before each mapping step. Finally, Algorithm 9 returns $\text{M}_{\text{red}}^{\text{Run}}$.

For each assignment our approach determines the Runnable-to-ECU mapping resulting in the minimum overall end-to-end latency of all event chains. Thus, it utilizes Algorithm 10 (GETECUMINOVERALLE2ELATENCY) to identify the corresponding ECU. This modified version of Algorithm 4 checks each ECU_i in $\mathcal{E}_{\text{rem}}$ based on their current mapping. It complements $\text{M}_{\text{cur}}^{\text{Run}}$ by inserting Run_i preserving order and precedence constraints by means of

Algorithm 9 REDUNDANCYRUNNABLEMAPPING

Input: Remaining ECUs $\mathcal{E}_{\text{rem}}$, InitialRunnableMapping $\text{M}_{\text{init}}^{\text{Run}}$, and slots Θ.
Output: RedundancyRunnableMapping $\text{M}_{\text{red}}^{\text{Run}} : \mathcal{R} \mapsto \mathcal{E}$.

1: $\mathcal{R}_{\text{fail}} \leftarrow$ GETFAILEDECUTASKSET($\text{M}_{\text{init}}^{\text{Run}}, \mathcal{E}_{\text{rem}}$)
2: $\text{M}_{\text{red}}^{\text{Run}} \leftarrow$ GETMAPPINGFORREMAININGECUS($\text{M}_{\text{init}}^{\text{Run}}, \mathcal{E}_{\text{rem}}$)
3: **for all** $\text{Run}_i \in \Gamma_{\text{fail}}$ **do**
4: $\quad \mathcal{E}_{\text{tmp}} \leftarrow$ GETFEASIBLEECUS($\text{Run}_i, \mathcal{E}_{\text{rem}}$)
5: $\quad \text{ECU}_{\text{tmp}} \leftarrow$ GETECUMINOVERALLE2E(Run_i, $\mathcal{E}_{\text{tmp}}, \text{M}_{\text{red}}^{\text{Run}}$)
6: $\quad \text{M}_{\text{red}}^{\text{Run}} \leftarrow \text{M}_{\text{red}}^{\text{Run}} \cup$ MAPRUNNABLE(Run, ECU_{tmp})
7: $\quad$ UPDATESTARTANDFINISHTIMES($\mathcal{R}$)
8: $\quad$ UPDATEAVAILABLESLOTS(Θ)
9: **end for**
10: **return** RedundancyRunnableMapping $\text{M}_{\text{red}}^{\text{Run}} : \mathcal{R} \mapsto \mathcal{E}$

Algorithm 10 GETECUMINOVERALLE2ELATENCY

Input: Runnable Run_i, candidate ECUs $\mathcal{E}_{\text{tmp}}$, available slots Θ, and current mapping $\text{M}_{\text{cur}}^{\text{Run}}$.
Output: ECU resulting in minimum overall end-to-end (E2E) latency.

1: **for all** $\text{ECU}_i \in \mathcal{E}_{\text{tmp}}$ **do**
2: $\quad \text{M}_i^{\text{Run}} \leftarrow \text{M}_{\text{cur}}^{\text{Run}} \cup$ MAPTASK(Run_i, ECU_i)
3: $\quad \text{E2E}_{\text{ECU}_i} \leftarrow$ GETOVERALLE2EDELAYANDCOMOVERHEAD($\text{M}_i^{\text{Run}}, \Theta$)
4: $\quad \text{E2E} \leftarrow \text{E2E} \cup \text{E2E}_{\text{ECU}_i}$
5: **end for**
6: ECU $\leftarrow$ ECUMINE2EANDCOMOVERHEAD(E2E)
7: **return** ECU

deadlines and release times. This insertion results in Runnable shiftings and growing execution delays due to the constraints on one or more of the ECUs. The algorithm calculates the overall end-to-end delay for all event chains implied by M_i^{Run} and stores it referencing to ECU_i. This results in a set E2E of end-to-end delays, i.e. one for each Runnable-to-ECU mapping. Finally, Algorithm 10 compares the values in E2E and returns the ECU with minimum overall end-to-end latency. A more detailed description on our algorithms for the redundancy mapping and the overall end-to-end latency determination are given in Section 4.4.1. The presented modified versions of the algorithms for the initial and redundancy task mappings differ only in the representation of their input data. Since the performed operations are the same, the time complexity remains the same as described in Section 4.4.1.

Runnable Mapping Result

Figure 5.13 depicts the resulting Gantt Charts for the TC and ACC systems in the given network topology with three ECUs, i.e. one chart for the initial mapping and one for each of the three required redundancy mappings. Additionally, Table 5.3 provides the corresponding execution orders and properties of the Runnables for the different configurations. Figure 5.13 illustrates how our Runnable mapping approach preserves the initial mapping of Runnables on the remaining ECUs and inserts the required redundant Runnables for the individual redundancy mappings. For instance, the Runnables Run_1, Run_4, Run_7, Run_{13}, Run_{16}, and Run_{17} are initially mapped to ECU_1. This mapping is kept in Redundacy Mapping 2 and Redundancy Mapping 3 where ECU_1 is running. In each redundancy mapping the required redundant Runnables are inserted according to their timing constraints as defined by Algorithm 9. In Redundancy Mapping 2 the Runnable Run_2 initally mapped to ECU_2 is inserted between Run_4 and Run_7 on ECU_1.

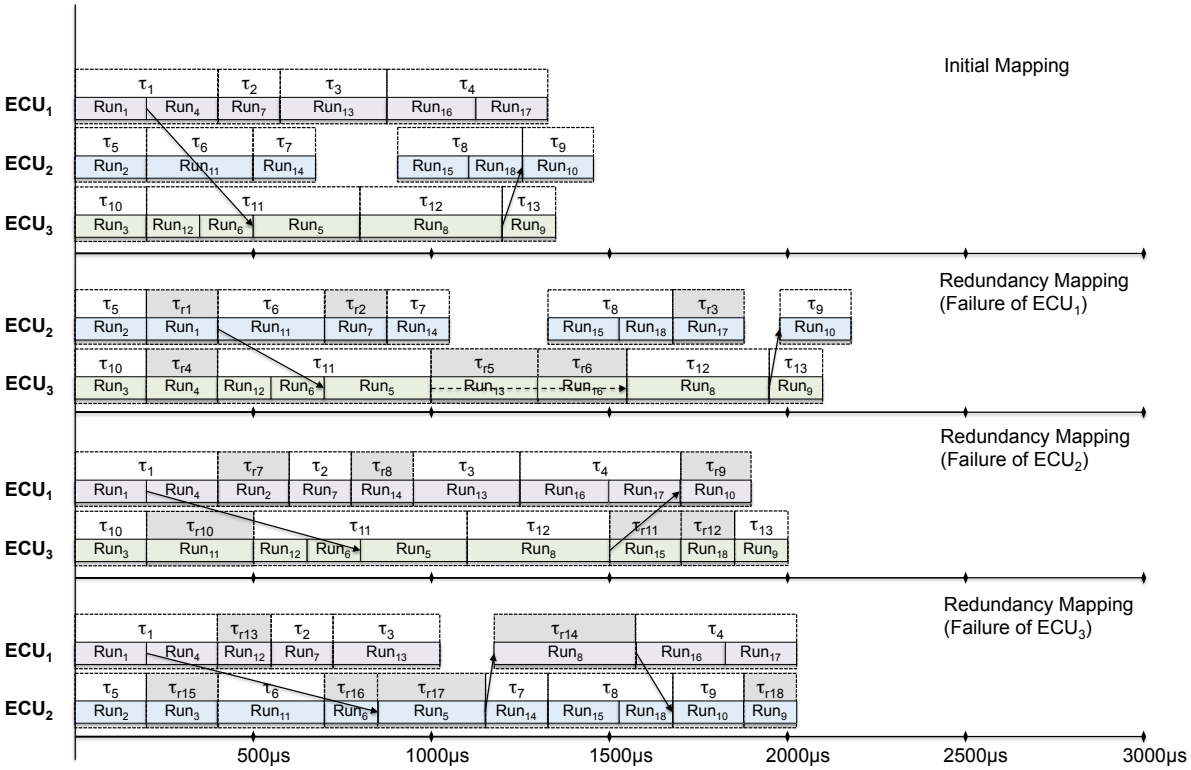


Figure 5.13: Gantt Charts of Runnable and task mappings for TC and ACC systems.

Furthermore, Table 5.3 shows that all Runnables in each configuration fulfill their timing constraints and meet their deadlines. This also implies that the defined maximum end-to-end latency constraints for all event chains from Runnables in $\mathcal{R}_{\text{in}}$ to Runnables in $\mathcal{R}_{\text{out}}$ are also fulfilled. As an example, Figure 5.13 illustrates the event chain $\text{SWC}_1 \rightarrow \text{SWC}_{10}$ defined on VFB Timing

(cf. Figure 5.11) for each configuration. Table 5.3 shows that the finishing time of Run_{10} has a maximum value of $2175\mu s$ in the redundancy mappings. Hence, the end-to-end latency constraint of $3000\mu s$ is clearly fulfilled for this event chain. All other event chains have even shorter end-to-end latency. Beside the feasibility of the determined Runnable-to-ECU mappings this result provides as additional feedback to the designer that he may consider ECUs with lower computational capacities and lower cost for the functional nodes because resulting longer WCETs probably still result in feasible deployments (cf. Section 4.2.5).

As described in Section 5.1.3, the AUTOSAR methodology defines the System View to specify the system design as output of the design phase to manage the overall results of the development process. This includes the modeling of the mapping of SWCs respectively Runnables to the ECUs and the resulting intra-ECU and inter-ECU communication of the performed deployment. Figure 5.14 depicts the output of our AUTOSAR-based deployment approach as System View model. It is based on the hardware architecture input model in Figure 5.12 and illustrates the resulting mapping for the TC and ACC system defined on VFB View (cf. Figure 5.10). The System View model defines which SWC is mapped to which ECU for the initial configuration. Additionally, it depicts which SWCs are redundantly mapped to which ECU for the individual reconfigurations. The System View model also illustrates the resulting inter-ECU messages over the FlexRay bus for all configurations.[32] Even though our deployment approach reduces the required remote communication, the figure shows that there are still several inter-ECU messages considering all configurations. For a complete feasible system design our deployment approach has to perform an appropriate bus mapping for these messages (cf. Figure 4.1). In Section 5.2.4 we describe how this is realized.

Task Mapping

As mentioned in Section 5.1.2, the unit of execution in AUTOSAR is an OS task. Hence, in AUTOSAR all Runnables assigned to an ECU must also be mapped to OS tasks to be scheduled and executed. An OS task may contain one or more Runnables. The resulting task set of an ECU is scheduled by AUTOSAR OS as described in Section 5.1.5. The simplest strategy would be to assign each Runnable to an individual task. But this is not practicable mainly due to two reasons: (i) the number of AUTOSAR OS tasks is limited and (ii) switching between tasks is time-consuming for an OS and should be avoided [SR08]. Therefore, we propose a more sophisticated Runnable-to-task mapping approach. This strategy utilizes the fact that in the reconfigurations the

[32] For a better recognizability the interconnections between the Runnables in $\mathcal{R}_{in}$ and $\mathcal{R}_{out}$ and the corresponding peripheral interface nodes are not drawn in.

initial mapping of Runnables to ECUs is preserved to reduce the number of required tasks. For this purpose, Runnables that are assigned to the same ECU and kept connected at each redundancy mapping, are encapsulated in one task.

Algorithm 11 RUNNABLETOTASKMAPPING

Input: Set of RunnableMappings $\mathfrak{M}^{\text{Run}}$ and set of ECUs $\mathcal{E}$.
Output: Set of task sets on ECUs $\mathcal{E}$: $\mathfrak{T}_{\mathcal{E}}$.

1: $\text{M}^{\text{Run}}_{\text{init}} \leftarrow$ GETINITIALRUNNABLEMAPPING($\mathfrak{M}^{\text{Run}}$)
2: **for all** $\text{ECU}_i \in \mathcal{E}$ **do**
3: $\quad \mathfrak{M}^{\text{Run}}_{\text{ECU}_i} \leftarrow$ GETREDUNDANCYMAPPINGS($\mathfrak{M}^{\text{Run}}, \text{ECU}_i$)
4: $\quad \mathcal{R}^{\text{init}}_{\text{ECU}_i} \leftarrow$ GETORDEREDRUNNABLES($\text{M}^{\text{Run}}_{\text{init}}, \text{ECU}_i$)
5: $\quad$ **for all** $\text{Run}_j \in \mathcal{R}^{\text{init}}_{\text{ECU}_i}$ **do**
6: $\quad\quad$ **if** CONNECTEDINALLREDUNDANCYMAPPINGS($\mathcal{R}_{\text{tmp}}, \text{Run}_j, \mathfrak{M}^{\text{Run}}_{\text{ECU}_i}$) **then**
7: $\quad\quad\quad \mathcal{R}_{\text{tmp}} \leftarrow \mathcal{R}_{\text{tmp}} \cup \text{Run}_j$
8: $\quad\quad$ **else**
9: $\quad\quad\quad \tau_{\text{tmp}} \leftarrow$ CREATETASK($\mathcal{R}_{\text{tmp}}$)
10: $\quad\quad\quad \Gamma_{\text{ECU}_i} \leftarrow \Gamma_{\text{ECU}_i} \cup \tau_{\text{tmp}}$
11: $\quad\quad\quad \mathcal{R}_{\text{tmp}} \leftarrow \text{Run}_j$
12: $\quad\quad$ **end if**
13: $\quad$ **end for**
14: $\quad$ **for all** $\text{M}^{\text{Run}}_k \in \mathfrak{M}^{\text{Run}}_{\text{ECU}_i}$ **do**
15: $\quad\quad \mathcal{R}^{k}_{\text{ECU}_i} \leftarrow$ GETORDEREDRUNNABLES($\text{M}^{\text{Run}}_k, \text{ECU}_i$)
16: $\quad\quad$ **for all** $\text{Run}_j \in \mathcal{R}^{k}_{\text{ECU}_i} \setminus \mathcal{R}^{\text{init}}_{\text{ECU}_i}$ **do**
17: $\quad\quad\quad \tau_{\text{tmp}} \leftarrow$ CREATETASK(Run_j)
18: $\quad\quad\quad \Gamma_{\text{ECU}_i} \leftarrow \Gamma_{\text{ECU}_i} \cup \tau_{\text{tmp}}$
19: $\quad\quad$ **end for**
20: $\quad$ **end for**
21: $\quad \mathfrak{T}_{\mathcal{E}} \leftarrow \mathfrak{T}_{\mathcal{E}} \cup \Gamma_{\text{ECU}_i}$
22: **end for**
23: **return** Set of task sets on ECUs $\mathcal{E}$: $\mathfrak{T}_{\mathcal{E}}$

Algorithm 11 (RUNNABLETOTASKMAPPING) represents in pseudocode how our Runnable-to-task mapping approach works. It gets the set $\mathfrak{M}^{\text{Run}}$ of all Runnable mappings and the set of ECUs $\mathcal{E}$ as input. Based on that input, it performs the Runnable-to-task mapping for each ECU and returns the resulting set of task sets $\mathfrak{T}_{\mathcal{E}}$ for all ECUs. As mentioned above, our approach utilizes the preservation of the initial Runnable mapping. Thus, at first it takes the initial mapping $\text{M}^{\text{Run}}_{\text{init}}$ from $\mathfrak{M}^{\text{Run}}$ before it iterates over each $\text{ECU}_i \in \mathcal{E}$ to perform their corresponding Runnable-to-task mappings.

In each iteration Algorithm 11 determines the set $\mathfrak{M}^{\text{Run}}_{\text{ECU}_i}$ of redundancy mappings where ECU_i is running and the set $\mathcal{R}^{\text{init}}_{\text{ECU}_i}$ of Runnables initially mapped to ECU_i in their scheduling respectively execution order. The algorithm goes

through all Runnables $\text{Run}_j \in \mathcal{R}^{\text{init}}_{\text{ECU}_i}$. In each step it checks if the current Runnable Run_j is directly connected to the set $\mathcal{R}_{\text{tmp}}$, i.e. directly follows in the execution order, in all redundancy mappings $\mathfrak{M}^{\text{Run}}_{\text{ECU}_i}$.[33] If this is true, Runnable Run_j is added to the set $\mathcal{R}_{\text{tmp}}$. Otherwise, a task τ_{tmp} is created and the currently determined set of connected Runnables is mapped to this task. Task τ_{tmp} is added to the task set Γ_{ECU_i} of ECU_i and $\mathcal{R}_{\text{tmp}}$ is reinitialized with Run_j for the next iteration.

The Runnable-to-task mapping strategy described above reduces the number of required tasks for the Runnables which are active by default in all configurations. However, the redundant Runnable instances which are inactive by default also have to be mapped to tasks to be scheduled by the AUTOSAR OS in their corresponding reconfiguration. Therefore, Algorithm 11 iterates over each redundancy mapping $\text{M}^{\text{Run}}_k \in \mathfrak{M}^{\text{Run}}_{\text{ECU}_i}$ and determines their corresponding Runnables $\mathcal{R}^k_{\text{ECU}_i}$ mapped to ECU_i. For each Runnable $\text{Run}_j \in \mathcal{R}^k_{\text{ECU}_i} \setminus \mathcal{R}^{\text{init}}_{\text{ECU}_i}$ which was not initially mapped to ECU_i, it creates a task τ_{tmp}, maps the current Runnable to this task and adds the created task to the task set Γ_{ECU_i}. When all Runnables are mapped to tasks, the task set Γ_{ECU_i} of ECU_i is added to the common set of task sets $\mathfrak{T}_{\mathcal{E}}$ for all ECUs. Finally, Algorithm 11 returns the resulting set of task sets. Like the algorithms above which are modified versions of the ones in Section 4.4, it terminates after iterating over all ECUs and Runnables. The iteration over all Runnables for each ECU results in a time complexity of $O(|\mathcal{E}| \cdot |\mathcal{R}|)$. Compared to the performed initial and redundancy Runnable mappings this running time is negligible.

Figure 5.13 illustrates the resulting Runnable-to-task mapping for the TC and ACC systems. The initial mapping tasks which are reused in one or more of the reconfigurations are marked in white. The additionally required tasks for the redundant Runnables are marked in light grey. For instance, on ECU_3 the Runnables Run_{12}, Run_6, and Run_5 are mapped to task τ_{11}. This task can be used for the inital mapping and the two redundancy mappings where ECU_3 is also running. Beside these tasks which are active in the initial configuration and the reconfigurations, additional inactive tasks are required for the inactive redundant Runnable instances. For Redundancy Mapping 1, this results in the tasks τ_{r4}, τ_{r5}, and τ_{r6} while Redundancy Mapping 2 requires the additional tasks τ_{r10}, τ_{r11}, and τ_{r12}.

Summarized, for the faultless system execution this results in 13 tasks for the initial mapping instead of 18 tasks which would be required for a one-to-one mapping. Also for the different possible reconfigurations, the number of required tasks is reduced to 14 respectively 15. For larger and more complex systems with a higher number of Runnables, the reduction of required tasks probably increases.

[33] In the first iteration $\mathcal{R}_{\text{tmp}} = \emptyset$ holds.

Inital Mapping:

	Run_i	r_i [μs]	d_i [μs]	s_i [μs]	f_i [μs]
ECU_1	Run_1	0	2100	0	200
	Run_4	0	2100	200	400
	Run_7	0	2400	400	575
	Run_{13}	300	2650	575	875
	Run_{16}	600	2800	875	1125
	Run_{17}	850	3000	1125	1325
ECU_2	Run_2	0	2100	0	200
	Run_{11}	0	2350	200	500
	Run_{14}	0	2550	500	675
	Run_{15}	600	2850	900	1100
	Run_{18}	800	3000	1100	1250
	Run_{10}	900	3000	1250	1450
ECU_3	Run_3	0	2100	0	200
	Run_{12}	0	2350	200	350
	Run_6	0	2400	350	500
	Run_5	200	2400	500	800
	Run_8	500	2800	800	1200
	Run_9	900	3000	1200	1350

Redundancy Mapping 1 (Failure of ECU_1):

	Run_i	r_i [μs]	d_i [μs]	s_i [μs]	f_i [μs]
ECU_2	Run_2	0	2100	0	200
	Run_1	0	2100	200	400
	Run_{11}	0	2350	400	700
	Run_7	0	2400	700	875
	Run_{14}	0	2550	875	1050
	Run_{15}	600	2850	1325	1525
	Run_{18}	800	3000	1525	1675
	Run_{17}	850	3000	1675	1875
	Run_{10}	900	3000	1975	2175
ECU_3	Run_3	0	2100	0	200
	Run_4	0	2100	200	400
	Run_{12}	0	2350	400	550
	Run_6	0	2400	550	700
	Run_5	200	2400	700	1000
	Run_{13}	300	2650	1000	1300
	Run_{16}	600	2800	1300	1550
	Run_8	500	2800	1550	1950
	Run_9	900	3000	1950	2100

Redundancy Mapping 2 (Failure of ECU_2):

	Run_i	r_i [μs]	d_i [μs]	s_i [μs]	f_i [μs]
ECU_1	Run_1	0	2100	0	200
	Run_4	0	2100	200	400
	Run_2	0	2100	400	600
	Run_7	0	2400	600	775
	Run_{14}	0	2550	775	950
	Run_{13}	300	2650	950	1250
	Run_{16}	600	2800	1250	1500
	Run_{17}	850	3000	1500	1700
	Run_{10}	900	3000	1700	1900
ECU_2	Run_3	0	2100	0	200
	Run_{11}	0	2350	200	500
	Run_{12}	0	2350	500	650
	Run_6	0	2400	650	800
	Run_5	200	2400	800	1100
	Run_8	500	2800	1100	1500
	Run_{15}	600	2850	1500	1700
	Run_{18}	800	3000	1700	1850
	Run_9	900	3000	1850	2000

Redundancy Mapping 3 (Failure of ECU_3):

	Run_i	r_i [μs]	d_i [μs]	s_i [μs]	f_i [μs]
ECU_1	Run_1	0	2100	0	200
	Run_4	0	2100	200	400
	Run_{12}	0	2350	400	550
	Run_7	0	2400	550	725
	Run_{13}	300	2650	725	1025
	Run_8	500	2800	1175	1575
	Run_{16}	600	2800	1575	1825
	Run_{17}	850	3000	1825	2025
ECU_2	Run_2	0	2100	0	200
	Run_3	0	2100	200	400
	Run_{11}	0	2350	400	700
	Run_6	0	2400	700	850
	Run_5	200	2400	850	1150
	Run_{14}	0	2550	1150	1325
	Run_{15}	600	2850	1325	1525
	Run_{18}	800	3000	1525	1675
	Run_{10}	900	3000	1675	1875
	Run_9	900	3000	1875	2025

Table 5.3: Execution order and properties of Runnables resulting from initial and redundancy mappings.

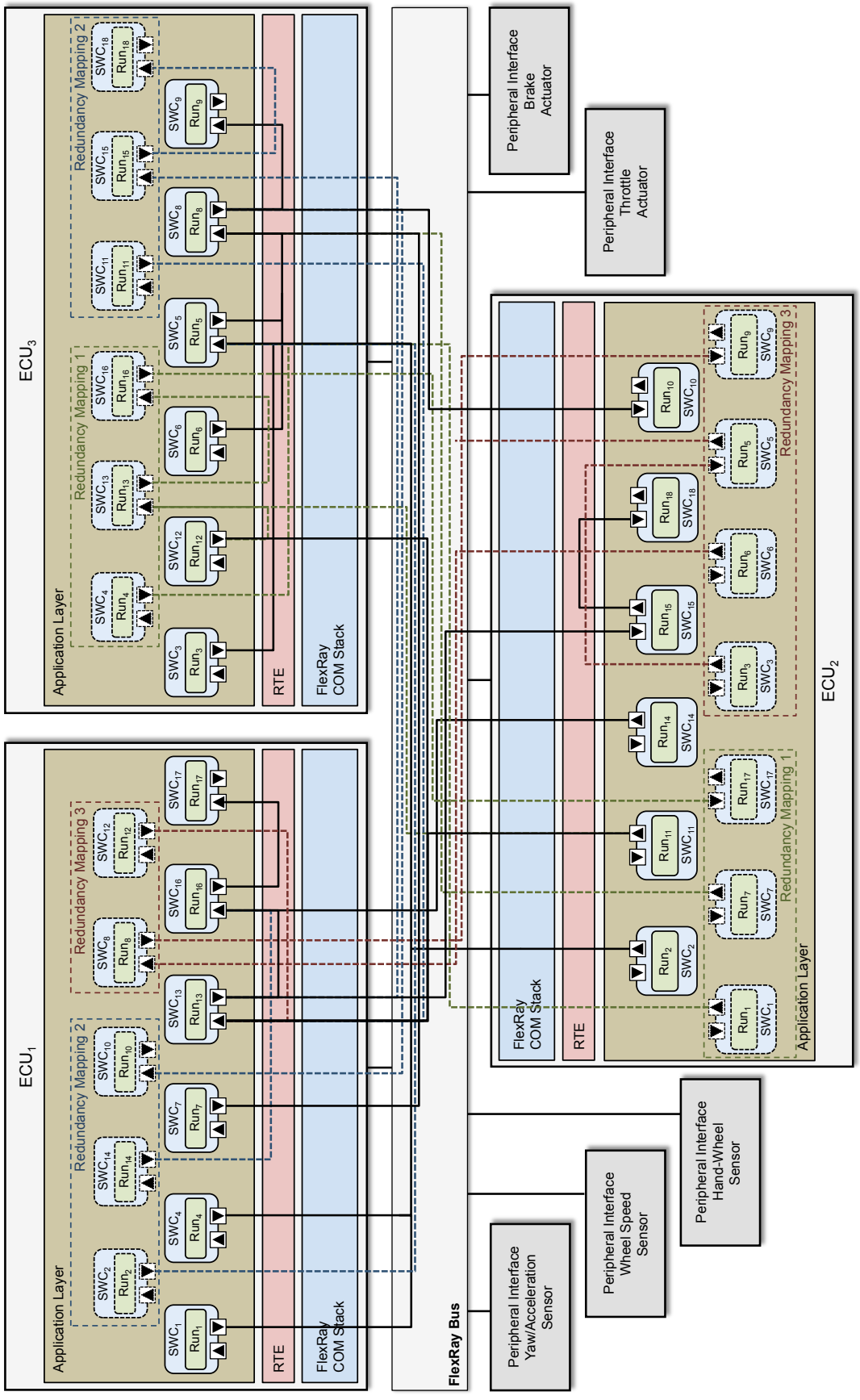


Figure 5.14: System View model of TC and ACC system after initial and redundancy mappings of SWCs respectively Runnables.

5.2.4 Bus Mapping

As part of the fault-tolerant deployment for a feasible bus mapping the number and size of inter-ECU messages and the bus properties have to be considered. The number of inter-ECU messages depends on the Runnable mappings and their size depends on the given software architecture. The Runnable mappings result in an available transmission interval $\Upsilon_{m_i} = [r_{m_i}, d_{m_i}[= [f_{\text{tx}}, s_{\text{rx}}[$ for each inter-ECU message m_i (cf. Section 5.2.1). It defines that m_i may be transmitted in a slot θ of the continuous set of slots $\Theta_i \in \Upsilon_{m_i}$. The number of slots in Θ_i depends on the configured slot size.

Algorithm 12 (AUTOSARBUSMAPPING) presents our AUTOSAR-based bus mapping approach in psuedocode. It is a modified version of the bus mapping Algorithm 5. Like the Runnable mapping algorithms presented above, this algorithm is also based on AUTOSAR-specific input but the performed operations remain the same. Instead of the task mapping TMGs, the HAG, and the COMG it gets the set $\mathfrak{M}^{\text{Run}}$ of determined Runnable mappings, the set $\mathcal{E}$ of ECUs, and the set Θ of FlexRay slots as input. Since only the representation of the input changed and the performed operations are the same as in Algorithm 5, the time complexity remains unchanged. Algorithm 12 starts with the mapping of the inter-ECU messages $\mathcal{M}^{\text{bus}}_{\text{M}^{\text{Run}}_{\text{init}}}$ of the initial Runnable mapping $\text{M}^{\text{Run}}_{\text{init}}$ and the set $\mathcal{E}_{\text{tx}}$ of their corresponding sender ECUs. For each sender $\text{ECU}_i \in \mathcal{E}_{\text{tx}}$ it identifies the transmitted inter-ECU messages $m_j \in \mathcal{M}^{\text{bus}}_{\text{ECU}_i}$ and their available slots Θ_{m_j} for transmission. Each message m_j is mapped to the first available slot θ_{map} and finally the resulting start times s_{m_j} and finishing times f_{m_j} of all messages $m_j \in \mathcal{M}^{\text{bus}}_{\text{ECU}_i}$ are updated.

Afterwards, for each redundancy mapping $\text{M}^{\tau}_i \in \mathfrak{M}^{\text{Run}} \setminus \{\text{M}^{\text{Run}}_{\text{init}}\}$, the algorithm also determines the inter-ECU messages $m_k \in \mathcal{M}^{\text{bus}}_{\text{ECU}_j}$ transmitted by ECU_j. It iterates over all messages m_k and determines their available slots Θ_{m_k} for transmission. For possible slot reuse it checks if m_k from ECU_j was initially mapped to a slot $\theta \in \Theta_{m_k}$. If this is true, the set Θ_{m_k} is set to just this assigned slot for reusing the message. In each iteration the determined set of available slots Θ_{m_k} complements the set Ξ_{ECU_j} of available slot sets for all inter-ECU messages sent by ECU_j. For the actual mapping, the algorithm determines the overlapping slots in the sets of available slots in Ξ_{ECU_j} and maps each message to the first available overlapping slot. This implicitly realizes slot reuse and frame packing if possible. As described above, the messages initially mapped to a specific slot can only be mapped to the same slot, i.e. the message is reused. Messages with overlapping sets of available slots are mapped to the first available overlapping slot if their summed payload allows frame packing. Otherwise, they have to be transmitted in separate slots. After the actual mapping, the algorithm updates the start times s_{m_k} and finishing times f_{m_k} of all messages m_k. Finally, Algorithm 12 returns a combined bus

Algorithm 12 AUTOSARBUSMAPPING

Input: Set of RunnableMappings $\mathfrak{M}^{\text{Run}}$, ECUs $\mathcal{E}$, and slots Θ.
Output: BusMapping $\text{M}^{\text{bus}} : (\mathcal{M}^{\text{bus}}, \mathcal{E}) \mapsto \Theta$

1: $\text{M}^{\text{Run}}_{\text{init}} \leftarrow$ GETINITIALMAPPING($\mathfrak{M}^{\text{Run}}$)
2: $\mathcal{M}^{\text{bus}}_{\text{M}^{\text{Run}}_{\text{init}}} \leftarrow$ GETINTERECUMSGS($\text{M}^{\text{Run}}_{\text{init}}$)
3: $\mathcal{E}_{\text{tx}} \leftarrow$ GETSENDERECUS($\mathcal{M}^{\text{bus}}_{\text{M}^{\text{Run}}_{\text{init}}}, \mathcal{E}$)
4: **for all** $\text{ECU}_i \in \mathcal{E}_{\text{tx}}$ **do**
5: $\quad\mathcal{M}^{\text{bus}}_{\text{ECU}_i} \leftarrow$ GETTXMESSAGESFROMECU($\text{M}^{\text{Run}}_{\text{init}}, \text{ECU}_i$)
6: $\quad$**for all** $m_j \in \mathcal{M}^{\text{bus}}_{\text{ECU}_i}$ **do**
7: $\quad\quad\Theta_{m_j} \leftarrow$ GETSLOTSINTXINTERVAL(m_j)
8: $\quad\quad\theta_{\text{map}} \leftarrow$ GETFIRSTAVAILABLESLOT(Θ_{m_j})
9: $\quad\quad$MAPTOSLOT(m_j, θ_{map})
10: $\quad$**end for**
11: $\quad$UPDATESTARTANDFINISHTIMES($\mathcal{M}^{\text{bus}}_{\text{ECU}_i}$)
12: **end for**
13: **for all** $\text{M}^{\text{Run}}_i \in \mathfrak{M}^{\text{Run}} \setminus \{\text{M}^{\text{Run}}_{\text{init}}\}$ **do**
14: $\quad\mathcal{M}^{\text{bus}}_{\text{M}^{\text{Run}}_i} \leftarrow$ GETINTERECUMSGS(M^{Run}_i)
15: $\quad\mathcal{E}_{\text{tx}} \leftarrow$ GETSENDERECUS($\mathcal{M}^{\text{bus}}_{\text{M}^{\text{Run}}_i}, \mathcal{E}$)
16: $\quad$**for all** $\text{ECU}_j \in \mathcal{E}_{\text{tx}}$ **do**
17: $\quad\quad\mathcal{M}^{\text{bus}}_{\text{ECU}_j} \leftarrow$ GETTXMESSAGESFROMECU($\text{M}^{\text{Run}}_i, \text{ECU}_j$)
18: $\quad\quad$**for all** $m_k \in \mathcal{M}^{\text{bus}}_{\text{ECU}_j}$ **do**
19: $\quad\quad\quad\Theta_{m_k} \leftarrow$ GETSLOTSINTXINTERVAL(m_k)
20: $\quad\quad\quad$**if** CHECKFORSLOTREUSE(Θ_{m_k}) = **true then**
21: $\quad\quad\quad\quad\Theta_{m_k} \leftarrow$ GETASSIGNEDSLOT(m_k)
22: $\quad\quad\quad$**end if**
23: $\quad\quad\quad\Xi_{\text{ECU}_j} \leftarrow \Xi_{\text{ECU}_j} \cup \Theta_{m_k}$
24: $\quad\quad$**end for**
25: $\quad\quad$MAPTOFIRSTAVAILABLEOVERLAPPINGSLOTS(Ξ_{ECU_j})
26: $\quad\quad$UPDATESTARTANDFINISHTIMES($\mathcal{M}^{\text{bus}}_{\text{ECU}_j}$)
27: $\quad$**end for**
28: **end for**
29: **return** $\text{M}^{\text{bus}} : (\mathcal{M}^{\text{bus}}, \mathcal{E}) \mapsto \Theta$

mapping M^{bus} for all configurations. More details on the single steps of our bus mapping algorithm are given in Section 4.4.2.

The message sizes of the TC and ACC systems are 10 to 22 bit (cf. Table 5.2). As mentioned in Section 5.2.2, for the FlexRay bus setup we consider a slot size of $\theta_{\text{size}} = 25\mu s$. Table 5.4 summarizes the properties of the inter-ECU messages resulting from the Runnable and bus mappings. For each message m_i in each configuration, it provides the sender Run_{tx} and receiver Run_{tx} with their hosting ECUs. It shows the available transmission time interval

$\Upsilon_{m_i} = [f_{\text{tx}}, s_{\text{rx}}[$ and the corresponding available slot sets Θ_{m_i} resulting from the initial and redundancy mappings of the sender and receiver Runnables (cf. Table 5.3) for each m_i. Finally, it provides the slot θ_{m_i} assigned to m_i with its corresponding start time $s_{\theta_{m_i}}$ and finishing time $f_{\theta_{m_i}}$.

As shown in Table 5.2, all inter-ECU messages in the initial configuration are mapped to the first available slot in Θ_{m_i}. For nearly all messages, this is the first slot of the available slot set. However, m_2 is mapped to θ_{10} because it has another sender ECU than m_1 mapped to θ_9. Based on this initial mapping the bus mappings for the reconfigurations are performed. If possible, a message is also mapped to the first slot in Θ_{m_i}, e.g. m_{12} in Reconfiguration 1, m_2 in Reconfiguration 2, and m_{14} in Reconfiguration 3. If the first slot is already assigned to another sender ECU, m_i is mapped to the next available slot, e.g. m_7 and m_{10} in Reconfiguration 1. As described above, slots initially assigned to a specific message and sender ECU are reused in the reconfigurations if possible. In this example, the slot θ_9 for m_1 is reused in Reconfiguration 1 and 2. Also messages in the slots θ_{10} and θ_{17} are reused in one or more reconfigurations. Furthermore, the bus mapping approach utilizes frame packing to reduce the number of assigned slots if messages from the same sender ECU can be combined in one configuration. Here, for instance, slot θ_{27} is used for m_{10} and m_{11} in Reconfiguration 2 while θ_{47} and θ_{64} are used for frame packing in Reconfiguration 3.

Summarized, Table 5.2 shows that there a is a total number of 34 inter-ECU messages resulting from the Runnable mappings for the TC and ACC system. Nine messages for the initial configuration and 25 for the different reconfigurations. By applying slot reuse and frame packing our bus mapping approach reduces the number of required slots for the reconfigurations from 25 to 17. In a larger and more complex system with more inter-ECU messages, the reduction of required slots will probably further increase and therefore support the system designer to determine feasible bus mappings.

Initial Configuration:

m_i	$m_{(\mathrm{Run_{tx}},\mathrm{Run_{rx}})}$	$\mathrm{ECU_{tx}}$	$\mathrm{ECU_{rx}}$	$\Upsilon_{m_i} = [f_{\mathrm{tx}}, s_{\mathrm{rx}}[$	Θ_{m_i}	θ_{m_i}	$s_{\theta_{m_i}}$	$f_{\theta_{m_i}}$
m_1	$m_{(\mathrm{Run_1},\mathrm{Run_5})}$	$\mathrm{ECU_1}$	$\mathrm{ECU_3}$	$[200\mu s, 500\mu s[$	$\{\theta_9, \ldots, \theta_{20}\}$	θ_9	$200\mu s$	$225\mu s$
m_2	$m_{(\mathrm{Run_2},\mathrm{Run_5})}$	$\mathrm{ECU_2}$	$\mathrm{ECU_3}$	$[200\mu s, 500\mu s[$	$\{\theta_9, \ldots, \theta_{20}\}$	θ_{10}	$225\mu s$	$250\mu s$
m_4	$m_{(\mathrm{Run_4},\mathrm{Run_5})}$	$\mathrm{ECU_1}$	$\mathrm{ECU_3}$	$[400\mu s, 500\mu s[$	$\{\theta_{17}, \ldots, \theta_{20}\}$	θ_{17}	$400\mu s$	$425\mu s$
m_7	$m_{(\mathrm{Run_7},\mathrm{Run_8})}$	$\mathrm{ECU_1}$	$\mathrm{ECU_3}$	$[575\mu s, 800\mu s[$	$\{\theta_{24}, \ldots, \theta_{32}\}$	θ_{24}	$575\mu s$	$600\mu s$
m_9	$m_{(\mathrm{Run_8},\mathrm{Run_{10}})}$	$\mathrm{ECU_3}$	$\mathrm{ECU_2}$	$[1200\mu s, 1250\mu s[$	$\{\theta_{49}, \ldots, \theta_{50}\}$	θ_{49}	$1200\mu s$	$1225\mu s$
m_{10}	$m_{(\mathrm{Run_{11}},\mathrm{Run_{13}})}$	$\mathrm{ECU_2}$	$\mathrm{ECU_1}$	$[500\mu s, 575\mu s[$	$\{\theta_{21}, \ldots, \theta_{23}\}$	θ_{21}	$500\mu s$	$525\mu s$
m_{11}	$m_{(\mathrm{Run_{12}},\mathrm{Run_{13}})}$	$\mathrm{ECU_3}$	$\mathrm{ECU_1}$	$[350\mu s, 575\mu s[$	$\{\theta_{15}, \ldots, \theta_{23}\}$	θ_{15}	$350\mu s$	$375\mu s$
m_{12}	$m_{(\mathrm{Run_{14}},\mathrm{Run_{16}})}$	$\mathrm{ECU_2}$	$\mathrm{ECU_1}$	$[675\mu s, 875\mu s[$	$\{\theta_{28}, \ldots, \theta_{35}\}$	θ_{28}	$675\mu s$	$700\mu s$
m_{14}	$m_{(\mathrm{Run_{13}},\mathrm{Run_{15}})}$	$\mathrm{ECU_1}$	$\mathrm{ECU_2}$	$[875\mu s, 900\mu s[$	$\{\theta_{36}\}$	θ_{36}	$875\mu s$	$900\mu s$

Reconfiguration 1 (Failure of $\mathrm{ECU_1}$):

m_i	$m_{(\mathrm{Run_{tx}},\mathrm{Run_{rx}})}$	$\mathrm{ECU_{tx}}$	$\mathrm{ECU_{rx}}$	$\Upsilon_{m_i} = [f_{\mathrm{tx}}, s_{\mathrm{rx}}[$	Θ_{m_i}	θ_{m_i}	$s_{\theta_{m_i}}$	$f_{\theta_{m_i}}$
m_1	$m_{(\mathrm{Run_1},\mathrm{Run_5})}$	$\mathrm{ECU_2}$	$\mathrm{ECU_3}$	$[400\mu s, 700\mu s[$	$\{\theta_{17}, \ldots, \theta_{28}\}$	θ_{18}	$425\mu s$	$450\mu s$
m_2	$m_{(\mathrm{Run_2},\mathrm{Run_5})}$	$\mathrm{ECU_2}$	$\mathrm{ECU_3}$	$[200\mu s, 500\mu s[$	$\{\theta_9, \ldots, \theta_{28}\}$	θ_{10} (r)	$225\mu s$	$250\mu s$
m_7	$m_{(\mathrm{Run_7},\mathrm{Run_8})}$	$\mathrm{ECU_2}$	$\mathrm{ECU_3}$	$[875\mu s, 1550\mu s[$	$\{\theta_{36}, \ldots, \theta_{62}\}$	θ_{37}	$900\mu s$	$925\mu s$
m_9	$m_{(\mathrm{Run_8},\mathrm{Run_{10}})}$	$\mathrm{ECU_3}$	$\mathrm{ECU_2}$	$[1950\mu s, 1975\mu s[$	$\{\theta_{79}\}$	θ_{79}	$1950\mu s$	$1975\mu s$
m_{10}	$m_{(\mathrm{Run_{11}},\mathrm{Run_{13}})}$	$\mathrm{ECU_2}$	$\mathrm{ECU_3}$	$[700\mu s, 1000\mu s[$	$\{\theta_{29}, \ldots, \theta_{40}\}$	θ_{30}	$725\mu s$	$750\mu s$
m_{12}	$m_{(\mathrm{Run_{14}},\mathrm{Run_{16}})}$	$\mathrm{ECU_2}$	$\mathrm{ECU_3}$	$[1050\mu s, 1300\mu s[$	$\{\theta_{43}, \ldots, \theta_{52}\}$	θ_{43}	$1050\mu s$	$1075\mu s$
m_{14}	$m_{(\mathrm{Run_{13}},\mathrm{Run_{15}})}$	$\mathrm{ECU_3}$	$\mathrm{ECU_2}$	$[1300\mu s, 1325\mu s[$	$\{\theta_{53}\}$	θ_{53}	$1300\mu s$	$1325\mu s$
m_{15}	$m_{(\mathrm{Run_{16}},\mathrm{Run_{17}})}$	$\mathrm{ECU_3}$	$\mathrm{ECU_2}$	$[1550\mu s, 1675\mu s[$	$\{\theta_{63}, \ldots, \theta_{67}\}$	θ_{63}	$1550\mu s$	$1575\mu s$

Reconfiguration 2 (Failure of $\mathrm{ECU_2}$):

m_i	$m_{(\mathrm{Run_{tx}},\mathrm{Run_{rx}})}$	$\mathrm{ECU_{tx}}$	$\mathrm{ECU_{rx}}$	$\Upsilon_{m_i} = [f_{\mathrm{tx}}, s_{\mathrm{rx}}[$	Θ_{m_i}	θ_{m_i}	$s_{\theta_{m_i}}$	$f_{\theta_{m_i}}$
m_1	$m_{(\mathrm{Run_1},\mathrm{Run_5})}$	$\mathrm{ECU_1}$	$\mathrm{ECU_3}$	$[200\mu s, 800\mu s[$	$\{\theta_9, \ldots, \theta_{32}\}$	θ_9 (r)	$200\mu s$	$225\mu s$
m_2	$m_{(\mathrm{Run_2},\mathrm{Run_5})}$	$\mathrm{ECU_1}$	$\mathrm{ECU_3}$	$[600\mu s, 800\mu s[$	$\{\theta_{25}, \ldots, \theta_{32}\}$	θ_{25}	$600\mu s$	$625\mu s$
m_4	$m_{(\mathrm{Run_4},\mathrm{Run_5})}$	$\mathrm{ECU_1}$	$\mathrm{ECU_3}$	$[400\mu s, 800\mu s[$	$\{\theta_{17}, \ldots, \theta_{32}\}$	θ_{17} (r)	$400\mu s$	$425\mu s$
m_7	$m_{(\mathrm{Run_7},\mathrm{Run_8})}$	$\mathrm{ECU_1}$	$\mathrm{ECU_3}$	$[775\mu s, 1100\mu s[$	$\{\theta_{32}, \ldots, \theta_{44}\}$	θ_{32}	$775\mu s$	$800\mu s$
m_9	$m_{(\mathrm{Run_8},\mathrm{Run_{10}})}$	$\mathrm{ECU_3}$	$\mathrm{ECU_1}$	$[1500\mu s, 1700\mu s[$	$\{\theta_{61}, \ldots, \theta_{68}\}$	θ_{61}	$1500\mu s$	$1525\mu s$
m_{10}	$m_{(\mathrm{Run_{11}},\mathrm{Run_{13}})}$	$\mathrm{ECU_3}$	$\mathrm{ECU_1}$	$[500\mu s, 950\mu s[$	$\{\theta_{21}, \ldots, \theta_{38}\}$	θ_{27} (fp)	$650\mu s$	$675\mu s$
m_{11}	$m_{(\mathrm{Run_{12}},\mathrm{Run_{13}})}$	$\mathrm{ECU_3}$	$\mathrm{ECU_1}$	$[650\mu s, 950\mu s[$	$\{\theta_{27}, \ldots, \theta_{38}\}$	θ_{27} (fp)	$650\mu s$	$675\mu s$
m_{14}	$m_{(\mathrm{Run_{13}},\mathrm{Run_{15}})}$	$\mathrm{ECU_1}$	$\mathrm{ECU_3}$	$[1250\mu s, 1500\mu s[$	$\{\theta_{51}, \ldots, \theta_{60}\}$	θ_{51}	$1250\mu s$	$1275\mu s$

Reconfiguration 3 (Failure of $\mathrm{ECU_3}$):

m_i	$m_{(\mathrm{Run_{tx}},\mathrm{Run_{rx}})}$	$\mathrm{ECU_{tx}}$	$\mathrm{ECU_{rx}}$	$\Upsilon_{m_i} = [f_{\mathrm{tx}}, s_{\mathrm{rx}}[$	Θ_{m_i}	θ_{m_i}	$s_{\theta_{m_i}}$	$f_{\theta_{m_i}}$
m_1	$m_{(\mathrm{Run_1},\mathrm{Run_5})}$	$\mathrm{ECU_1}$	$\mathrm{ECU_2}$	$[200\mu s, 850\mu s[$	$\{\theta_9, \ldots, \theta_{34}\}$	θ_9 (r)	$200\mu s$	$225\mu s$
m_4	$m_{(\mathrm{Run_4},\mathrm{Run_5})}$	$\mathrm{ECU_1}$	$\mathrm{ECU_2}$	$[400\mu s, 850\mu s[$	$\{\theta_{17}, \ldots, \theta_{34}\}$	θ_{17} (r)	$400\mu s$	$425\mu s$
m_5	$m_{(\mathrm{Run_6},\mathrm{Run_8})}$	$\mathrm{ECU_2}$	$\mathrm{ECU_1}$	$[850\mu s, 1175\mu s[$	$\{\theta_{35}, \ldots, \theta_{47}\}$	θ_{47} (fp)	$1325\mu s$	$1325\mu s$
m_6	$m_{(\mathrm{Run_5},\mathrm{Run_8})}$	$\mathrm{ECU_2}$	$\mathrm{ECU_1}$	$[1150\mu s, 1175\mu s[$	$\{\theta_{47}\}$	θ_{47} (fp)	$1150\mu s$	$1175\mu s$
m_8	$m_{(\mathrm{Run_8},\mathrm{Run_9})}$	$\mathrm{ECU_1}$	$\mathrm{ECU_2}$	$[1575\mu s, 1875\mu s[$	$\{\theta_{64}, \ldots, \theta_{76}\}$	θ_{64} (fp)	$1575\mu s$	$1600\mu s$
m_9	$m_{(\mathrm{Run_8},\mathrm{Run_{10}})}$	$\mathrm{ECU_1}$	$\mathrm{ECU_2}$	$[1575\mu s, 1675\mu s[$	$\{\theta_{64}, \ldots, \theta_{68}\}$	θ_{64} (fp)	$1575\mu s$	$1600\mu s$
m_{10}	$m_{(\mathrm{Run_{11}},\mathrm{Run_{13}})}$	$\mathrm{ECU_2}$	$\mathrm{ECU_1}$	$[700\mu s, 725\mu s[$	$\{\theta_{29}\}$	θ_{29}	$700\mu s$	$725\mu s$
m_{12}	$m_{(\mathrm{Run_{14}},\mathrm{Run_{16}})}$	$\mathrm{ECU_1}$	$\mathrm{ECU_2}$	$[1325\mu s, 1575\mu s[$	$\{\theta_{54}, \ldots, \theta_{63}\}$	θ_{54}	$1325\mu s$	$1350\mu s$
m_{14}	$m_{(\mathrm{Run_{13}},\mathrm{Run_{15}})}$	$\mathrm{ECU_2}$	$\mathrm{ECU_1}$	$[1025\mu s, 1325\mu s[$	$\{\theta_{42}, \ldots, \theta_{53}\}$	θ_{42}	$1025\mu s$	$1050\mu s$

Table 5.4: Properties of inter-ECU messages resulting from Runnable and bus mappings.

5.2.5 Reconfiguration with AUTOSAR

After determining a feasible AUTOSAR-compliant SWC-to-ECU and Runnable-to-task mapping, two challenges remain to be solved for our fault-tolerant system design approach: (i) Detect a failed ECU and (ii) activate the appropriate redundant tasks within the ECU network according to the fault-tolerant reconfiguration.

In Section 4.2.2 we presented our distributed coordinator concept that allows self-reconfiguration of the remaining ECUs in case of a node failure. We described two different generic options for the implementation of the required coordinator component: The hardware-based extension of the FlexRay communication controller and the more flexible software-based protocol-independent solution by means of additional distributed coordinator tasks. Here, we propose a third option for the distributed coordinator concept implementation based on existing AUTOSAR concepts and mechanisms. This solution requires no hardware extensions and no dedicated additional coordinator tasks on Application Layer.

While AUTOSAR specifies a BSW called Watchdog Manager [AUT13l] to manage errors of BSW modules and SWCs respectively Runnables running on an ECU, there is no explicit specification regarding detection of failed nodes within an ECU network. As we described in Section 5.1.1, the AUTOSAR Software Architecture allows to design and implement Complex Device Drivers providing the possibility to integrate special purpose functionality. This implies that a CDD module may have interfaces to standard modules of the BSW and to SWCs via the RTE [AUT13c]. Therefore, we propose to extend the AUTOSAR BSW by integrating a corresponding CDD implementing the required functionality to detect failed ECUs.

Figure 5.15 depicts the integration of this CDD for reconfiguration in the BSW of the AUTOSAR Software Architecture. It illustrates that the CDD is connected to the FlexRay Interface within the Communication Stack. The FlexRay Interface module is the exclusive bus specific access point to the COM Stack and the CDD can use the standard API of the FlexRay Interface modules to access the I-PDUs [AUT13c]. The bus interface puts the I-PDUs into frames and prepares them for the transmission over the bus accessed via the corresponding bus driver (cf. Section 5.1.4). As described in Section 4.2.2, the CDD can detect an error and localize the corresponding ECU failure by monitoring and analyzing the received I-PDUs respectively frames. Since the AUTOSAR FlexRay Interface specification is based on the official standards and norms of the FlexRay consortium [AUT13j], the protocol-specific functionality provided by the BSW of the AUTOSAR COM Stack can be utilized to check if valid frames with correct payload data are

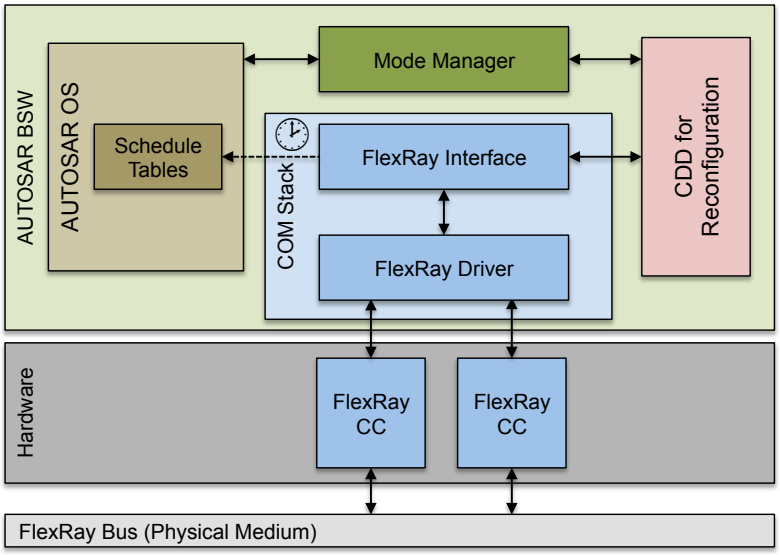


Figure 5.15: Integration of CDD for reconfiguration in AUTOSAR BSW.

received. Combined with the static slot-to-sender assignment, each ECU can identify failed ECUs (cf. Section 2.2.3).

When a failed ECU is detected, each remaining ECU has to activate its appropriate redundant tasks. For this purpose, we propose to utilize the Schedule Table concept of AUTOSAR OS presented in Section 5.1.5. For each ECU, we define one specific Schedule Table for each configuration of this ECU, i.e. a Schedule Table activates only those tasks that are part of the corresponding configuration. Utilizing the different states that each Schedule Table can enter, the Schedule Table with the currently required configuration is started (RUNNING) while all the others are stopped (NOT_RUNNING). However, the guarantee of a correct temporal behavior in a distributed real-time system requires the availability of a global time base of proper precision among all involved nodes (cf. Section 2.2.2).

Thus, the Schedule Table concept also provides a status which defines that the Schedule Table is started and runs synchronous to a global time base (RUNNING_AND_SYNCHRONOUS) [AUT13n]. Furthermore, like the author in [Jan07], we propose to utilize the Flexray Interface respectively FlexRay clock to support the required synchronization of Schedule Tables running on different ECUs within the network. As described in Section 4.2.5 for the time-triggered periodic real-time tasks considered by our approach, the Schedule Tables are configured to run with a periodic repetition rate synchronized to the FlexRay cycle until they are stopped. It is important to remind that tasks are only activated by Schedule Tables, i.e. tasks require an appropriate priority assignment to ensure that they are scheduled on time. As mentioned above, we propose to utilize the most widely applied Rate (Deadline) Monotonic Priority Assignment approach for this purpose.

Having the AUTOSAR-compliant concepts to detect a failed ECU within a network and to manage different task activation patterns on an ECU, we need to combine these concepts. This is realized by utilizing the BSW *Mode Manager* module [AUT13g]. AUTOSAR supports different modes for the hosting ECU respectively vehicle. Modes are defined via *ModeDeclarations*. For instance in the ECU state management, there may exist the ModeDeclarations STARTUP, RUN, POST_RUN, and SLEEP [AUT13d]. Mode Managers are responsible for switching between modes to arbitrate mode requests from other BSW modules or Application Layer SWCs [AUT13g].

For our AUTOSAR-based DCC implementation, we propose to define one specific mode per configuration on a particular ECU. By means of these modes, the CDD can request a mode switch from the Mode Manager when a failed ECU is detected. Thus, the CDD for reconfiguration has an interface to the Mode Manager module as illustrated in Figure 5.15. Via its interface to the AUTOSAR OS, the Mode Manger performs the appropriate mode switch which enforces that the currently running Schedule Table is stopped and, depending on the failed ECU, the corresponding Schedule Table is started synchronously to the global time base by means of the FlexRay Interface.

By utilizing and extending the existing AUTOSAR BSW modules and concepts as described above, we are are able to realize the required reconfiguration steps without any hardware modifications or additional run-time consuming tasks on the Application Layer.

5.3 Summary

In this chapter we presented the application of our approach to the design of automotive real-time systems. Therefore, we presented its integration to the modeling and deployment of automotive software based on AUTOSAR, a well-defined and standardized methodology for the development of automotive systems.

An introduction to the AUTOSAR standard provided the required information about its basic concepts for the application of our fault-tolerant design approach. Based on these means, the chapter described how we integrated our design approach to perform a fault-tolerant deployment of real-time software in AUTOSAR networks including the modeling of software and hardware architecture and the appropriate mappings. Therefore, we presented modified versions of our algorithms from Chapter 4 and applied them to real-world examples. Finally, it was described how to realize fault tolerance by means of reconfiguration utilizing and extending the existing AUTOSAR BSW modules and concepts.

Chapter 6

Conclusion and Outlook

Nowadays, there exist numerous safety-critical distributed systems with hard real-time constraints. This means that missing of timing constraints in such systems may cause catastrophic consequences on the environment or even people. Hence, the main attributes for the dependability of these safety-critical systems are reliability and safety. A common strategy to ensure reliability and safety is fault tolerance. This thesis addresses the rising challenges and presents a novel approach and concepts for the design of fault-tolerant distributed real-time systems. This chapter concludes the thesis by summarizing the main contributions and providing an outlook for future work.

6.1 Conclusion

The main objective of this thesis is to support the system designer by means of our fault-tolerant design approach. Therefore, we extended and refined the existing design flow steps for distributed systems introduced in Chapter 1 and provided further concepts. In the following, the main contributions and properties of our work presented in Chapter 4 and 5 are summarized.

Compensation of arbitrary operational hardware faults Different possible operational hardware faults during system runtime can be distinguished by means of their occurrence and duration and result in different behaviors of the faulty component. In a distributed system, these faults may result in either network or node failures. As described in Section 4.2, the compensation of one arbitrary component fault is sufficient in most cases to ensure fault tolerance. To compensate one network failure, we utilize the FlexRay protocol which offers one redundant communication channel. To compensate the possible ECU failures, our approach makes use of dynamic redundancy and reconfiguration by means of the Cold Standby task replica concept.

Reconfigurable distributed system topology In current distributed systems ECUs are often hardwired to sensors or actuators to exchange data with the environment. This results in placement constraints for tasks which have to exchange data with sensors or actuators because they have to be executed on the ECUs connected to the I/O. A failure of such an ECU cannot be compensated with dynamic redundancy and reconfiguration because required connections get lost. Therefore, in Section 4.2.1 we proposed a reconfigurable distributed system topology with peripheral interfaces to the I/O and dedicated nodes for functional task execution. This reconfigurable toplogy offers the necessary flexibility for our approach.

Distributed coordinator concept To realize fault tolerance through dynamic redundancy, a component which coordinates and performs the required reconfiguration steps has to be integrated in the system. In Section 4.2.2, we discussed several methods how to realize this coordination and presented our advantageous distributed coordinator concept which allows the self-reconfiguration of the system. This concept adds distributed coordinator tasks to every node of the system which organize the required reconfiguration by means of information about active and redundant tasks and communication properties in all possible configurations. Based on these information and the properties of the FlexRay protocol, the distributed coordinator tasks are able to detect arbitrary ECU failures and perform the corresponding reconfiguration.

Extended schedulability tests Task scheduling in distributed systems has to cover the local level of each processor and the global scheduling, i.e. the mapping of tasks to ECUs. Moreover, tasks have to communicate locally via intra-ECU communication or remotely over the network via inter-ECU communication depending on their mapping. The transmission time of an inter-ECU message is negligible. The transmission of inter-ECU messages may result in a communication delay, i.e. the execution of a task which receives input data from another task over the network may be delayed by the transmission time of the message. This means, that beside the task WCETs also the resulting communication delays have to be considered for the schedulability test. In this thesis, we consider RM/DM scheduling that is still most frequently applied in many important real-time domains and EDF scheduling which has some significant advantages over RM/DM scheduling. Therefore, in Section 4.2.4 we describe our extended schedulibality tests for these strategies which integrate the resulting communication delays.

Modeling of system properties To perform an appropriate system design, our approach requires information about the properties of the given software architecture and hardware topology as well as about the resulting timing constraints and communication properties. In Section 4.2 and 4.3, we presented

our graph-based modeling of the system properties as input for our approach. We defined a task dependency graph representing the software architecture, that is annotated with the corresponding timing properties and constraints. For the hardware architecture we proposed a hardware architecture graph representing the available ECUs in the network, that is annotated with the execution time factor of the computational performance for each ECU. As final input graph, we modeled a communication graph representing the communication cycle and its slots, that is annotated with slot time intervals resulting from the bus setup.

Deployment and design result The fault-tolerant system deployment presented in Section 4.4 is an essential part of our approach and provides the design result. It comprises different algorithms for the determination of feasible, flexible, and efficient task and bus mappings for the initial configuration and based on that for all necessary reconfigurations to design a fault-tolerant system. Based on the model input described above, the initial and the required redundancy task mappings as well as a combined bus mapping are performed. Finally, our approach returns the resulting design graph representing the combination of the mapping graphs for all configurations as feedback for the designer. Depending on the given input, our deployment approach is able to return a complete feasible solution for all possible configurations or just for a subset. Based on this feedback the designer can design the system according to the determined solution or perform a further iteration of the design approach (cf. Section 4.5).

Application to AUTOSAR Nowadays, AUTOSAR is the established de facto standard for the development of automotive systems. In Section 5.2, we described the application of our approach to the design of automotive systems and the deployment of real-time software based on AUTOSAR. The AUTOSAR methodology strictly specifies the representation of the software architecture and the hardware topology. For this purpose we presented modified versions of our deployment algorithms which get this AUTOSAR-specific representation as input and perform the same operations to determine the corresponding Runnable, task, and bus mappings. This fault-tolerant deployment returns an AUTOSAR-compliant system model which represents the determined initial and redundancy mappings as well as the resulting inter-ECU messages. Additionally, we proposed a further concept to coordinate and realize the required reconfigurations that is based on existing AUTOSAR components and does not require hardware extensions or additional tasks.

Page 197 lists the publications [Klo+09], [Klo+10b], [Klo+10a], [Klo+10c], [KKM11], [Klo+11], [KMR12], and [Klo+13] which we published in the context of this thesis. These publications are the basis for the contributions described above and the overall work presented in this thesis.

6.2 Outlook

Even though our design approach presented in this thesis supports the system designer to develop fault-tolerant distributed real-time systems, there are still some open issues and improvements to be addressed in future work. Besides our design approach to determine a feasible system deployment, we also presented novel concepts for reconfigurable distributed systems and distributed coordination, to enable the self-reconfiguration during system runtime. However, the actual setup of such a network goes beyond the scope of this thesis. Nevertheless, it is planned to build-up a reconfigurable network topology in future work, to implement and extensively test the reconfiguration concepts on real hardware components.

As mentioned above, here we focus on the compensation of one arbitrary component fault. However, with minor modifications to the presented algorithms our approach is also applicable to determine solutions for the compensation of more than one ECU failure. But to compensate multiple network failures, also the fault-tolerance of the underlying network topology has to be further increased. Hence, especially in very large and fault-prone systems the consideration of multiple simultaneous failures is reasonable and deserves further investigation in future research.

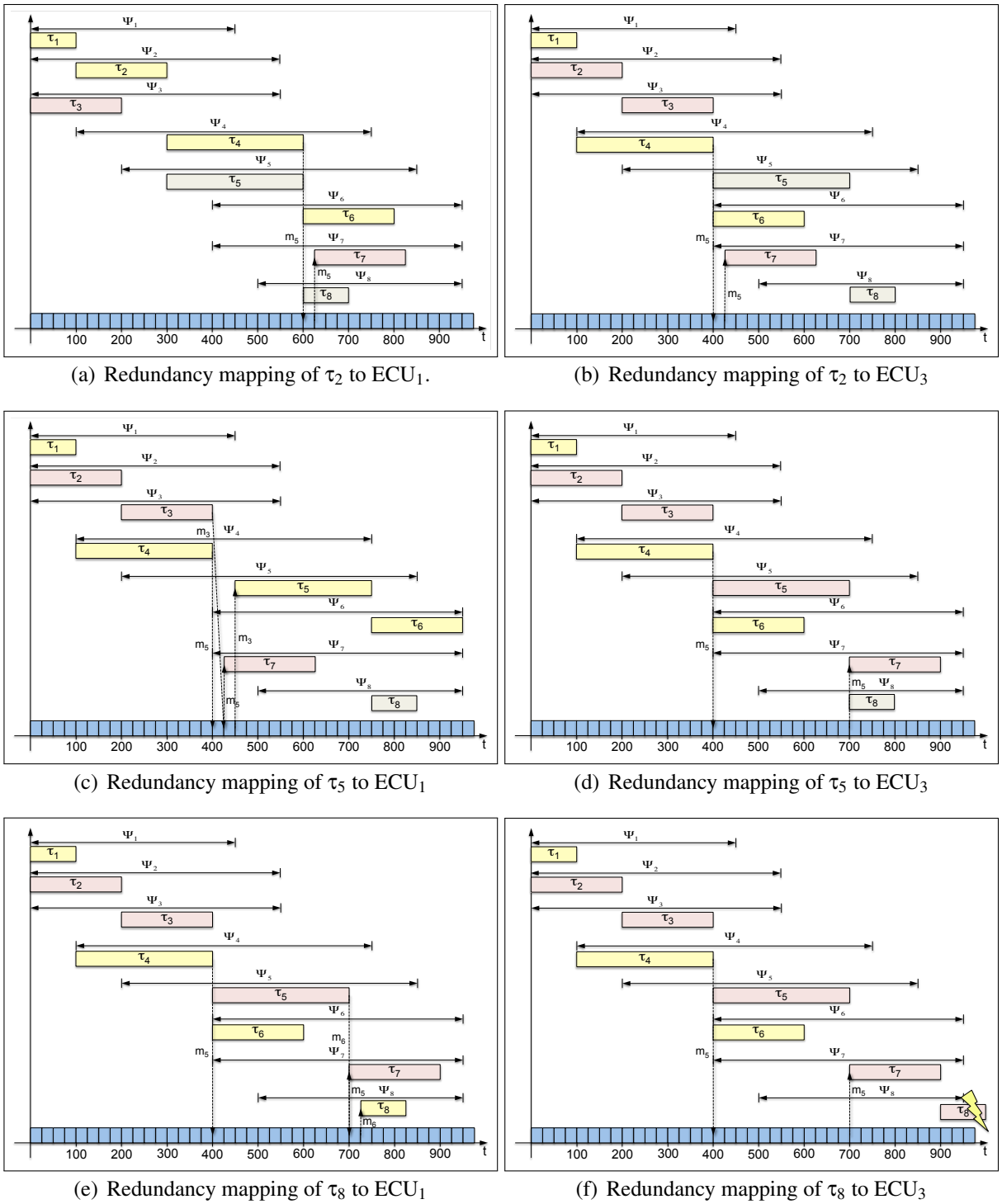
Moreover, we consider the well established network topology with single-core ECUs for our distributed system design approach. Nevertheless, also in the automotive domain multi-core processor hardware is currently arising and seen as a solution to the problem of increasing ECU processing power without increasing power consumption, generating heat, or producing more electromagnetic radiation [MB09]. To address this development, our design approach could be extended to support an efficient and flexible fault-tolerant deployment to multi-core ECUs.

Our approach offers support for the fault-tolerant design of pure hard real-time systems, i.e. safety-critical systems. However, only a subset of the functionality of a complete system may be safety-critical. For instance, an aircraft system is composed of a mixture of safety-critical and non-critical parts as it contains a non-critical passenger entertainment system that is isolated from the safety-critical flight systems. Therefore, these systems are called *mixed criticality systems*. The so-called runtime robustness is a specific form of fault tolerance for mixed criticality systems that considers different criticality levels: If not all components can be served satisfactorily, the system ensures that lower criticality components are denied their requested levels of service before higher criticality components [BBD11]. By means of this knowledge, our approach could be extended to support the fault-tolerant design of mixed criticality systems.

Appendix A

Additional Figures from Chapter 4

This appendix provides additional figures containing the Gantt Charts and TMGs representing the corresponding task mappings for the possible reconfigurations of the example presented in Section 4.4.



(a) Redundancy mapping of τ_2 to ECU_1.

(b) Redundancy mapping of τ_2 to ECU_3

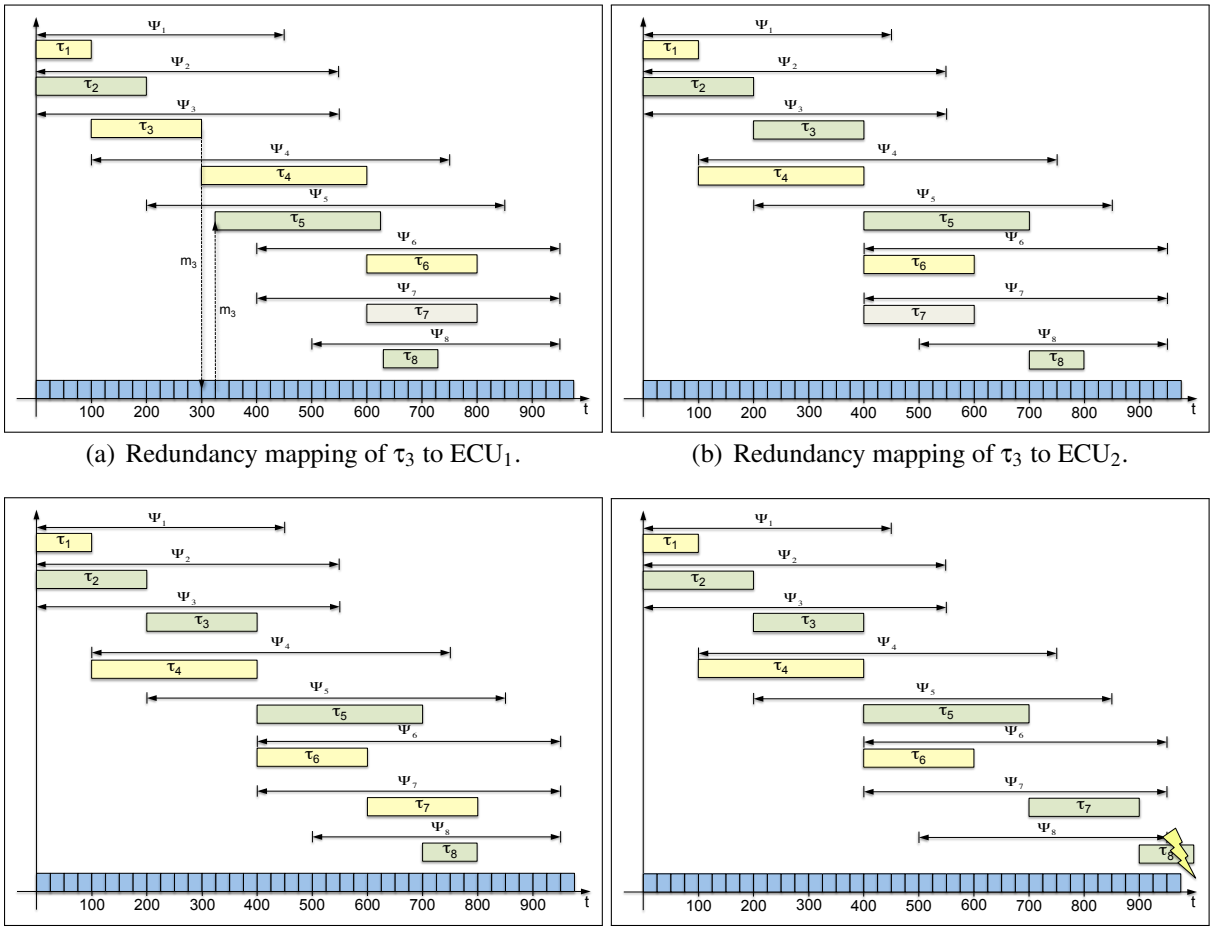
(c) Redundancy mapping of τ_5 to ECU_1

(d) Redundancy mapping of τ_5 to ECU_3

(e) Redundancy mapping of τ_8 to ECU_1

(f) Redundancy mapping of τ_8 to ECU_3

Figure A.1: Gantt Charts for local schedulings and overall end-to-end delays resulting from redundancy task mappings to different remaining ECUs (Failure of ECU_2).



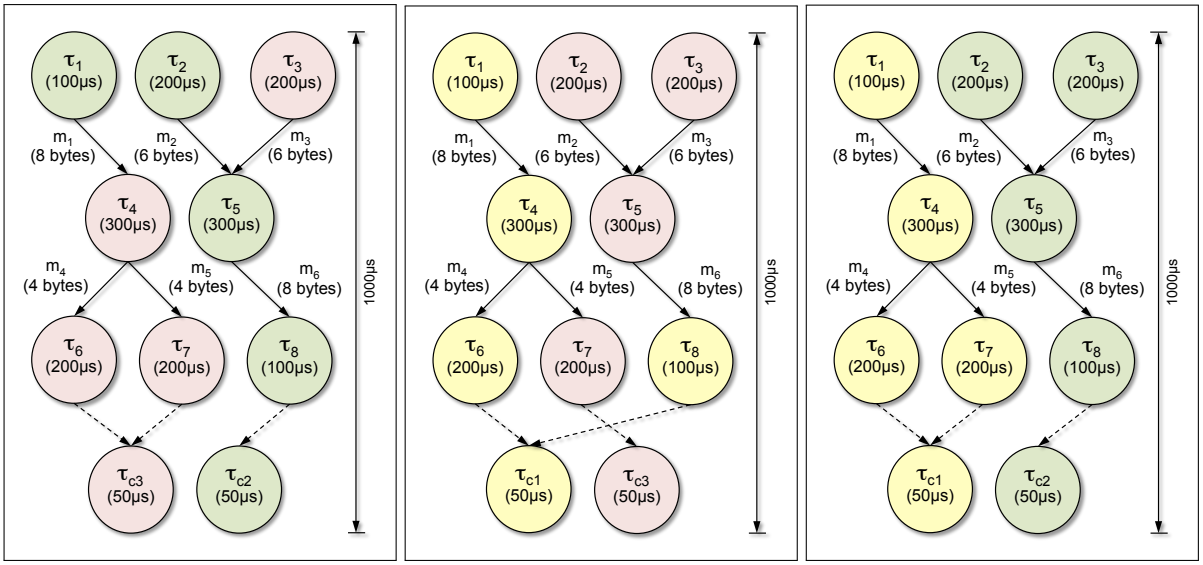
(a) Redundancy mapping of τ_3 to ECU_1.

(b) Redundancy mapping of τ_3 to ECU_2.

(c) Redundancy mapping of τ_7 to ECU_1.

(d) Redundancy mapping of τ_7 to ECU_2.

Figure A.2: Gantt Charts for local schedulings and overall end-to-end delays resulting from redundancy task mappings to different remaining ECUs (Failure of ECU_3).

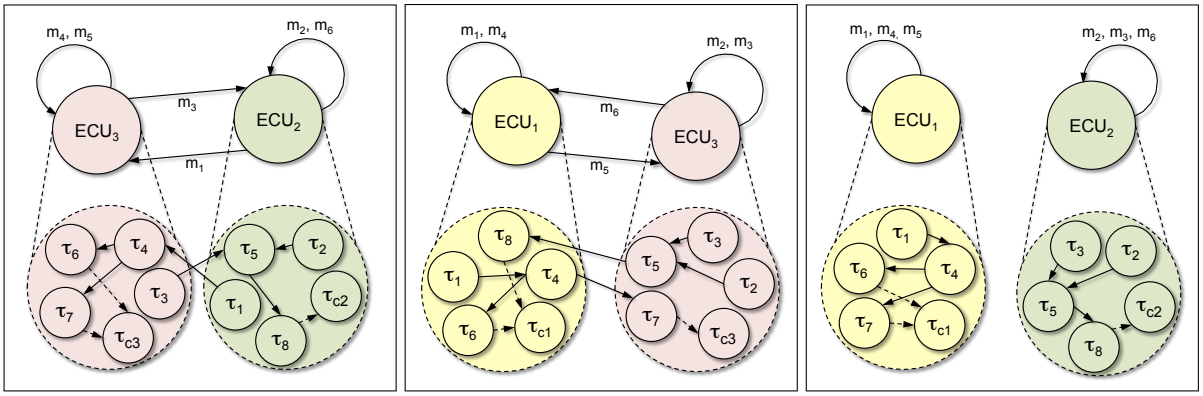


(a) Task mapping of reconfiguration for failure of ECU_1.

(b) Task mapping of reconfiguration for failure of ECU_2.

(c) Task mapping of reconfiguration for failure of ECU_3.

Figure A.3: Task mappings of all possible reconfigurations for input graphs shown in Figure 4.12(b) and 4.13 .



(a) TMG of reconfiguration for failure of ECU_1.

(b) TMG of reconfiguration for failure of ECU_2.

(c) TMG of reconfiguration for failure of ECU_3.

Figure A.4: TMGs of all possible reconfigurations for input graphs shown in Figure 4.12(b) and 4.13 .

List of Acronyms

ABS	Antilock Braking System
ACC	Adaptive Cruise Control
BSW	Basis Software
COMG	Communication Graph
DAG	Directed Acyclic Graph
DCC	Distributed Coordinator Concept
DG	Design Graph
DM	Deadline Monotonic
ECU	Electronic Control Unit
EDF	Earliest Deadline First
EDF*	Earliest Deadline First with Precedence Constraints
FIT	Failures In Time
HAG	Hardware Architecture Graph
I/O	Input/Output
PDC	Processor Demand Criterion
PDC*	Extended Processor Demand Criterion
PDU	Protocol Data Unit
RM	Rate Monotonic
RTA	Response Time Analysis
RTA*	Extended Response Time Analysis
RTE	Runtime Environment

SPOF Single-Point-Of-Failure

SWC Software Component

TC Traction Control

TDMA Time Division Multiple Access

TDG Task Dependency Graph

TMG Task Mapping Graph

TTP Time Triggered Protocol

WCET Worst Case Execution Time

WCRT Worst Case Response Time

VFB Virtual Function Bus

List of Notations

δ_i	Communication delay of task τ_i
Γ	Set of periodic tasks
$\mathcal{E}$	Set of ECUs
$\mathcal{M}$	Set of periodic messages
$\mathcal{M}^{\text{bus}}$	Set of inter-node messages
$\mathcal{M}^{\text{bus}}_{\text{ECU}_{\text{tx}}}$	Set of inter-node messages sent by ECU_{tx}
$\mathcal{M}^{\text{bus}}_{\text{M}^{\tau}_i}$	Set of inter-node messages in task mapping M^{τ}_i
$\mathcal{R}$	Set of Runnables
$\mathfrak{c}(\tau_i)$	Labeling function to annotate the WCET to each task τ_i in a TDG
$\mathfrak{d}(m_i)$	Labeling function to annotate data size to each message m_i in a TDG
$\mathfrak{f}(\text{ECU}_j)$	Labeling function to annotate execution time factor to each ECU_j in a HAG
$\mathfrak{M}^{\text{Run}}$	Set of Runnable mappings
$\mathfrak{M}^{\tau}$	Set of task mappings
$\mathfrak{s}(\theta_i)$	Labeling function to annotate slot time interval to each slot θ_i in a COMG
$\mathfrak{T}_{\mathcal{E}}$	Set of task sets on ECUs $\mathcal{E}$
$\Omega_{\text{ECU}_{\text{tx}}}(\mathcal{M})$	Overlapping available FlexRay slots for inter-ECU messages $\mathcal{M}$ sent by ECU_{tx} ($\mathcal{M} \subseteq \mathcal{M}^{\text{bus}}_{\text{ECU}_{\text{tx}}}$)
ϕ_i	Phase of task τ_i
$\Psi_{i,j}$	Available execution time interval of j-th instance of τ_i
$\psi_{m_{(\tau_j,\tau_i)}}$	Transmission time of message $m_{(\tau_j,\tau_i)}$
τ_i	Generic periodic task
$\tau_{i,j}$	j-th instance of task τ_i

$\text{duration}(\theta)$	Duration (time) of FlexRay slots
ECU_i	Generic ECU
ECU_{τ_i}	ECU hosting task τ_i
$\text{maxPayload}(\theta)$	Maximum available payload of FlexRay slots
M^{bus}	Bus (message) mapping
M^{Run}	Runnable mapping
M^{τ}	Task mapping
$\text{payload}(m_i)$	Payload length of message m_i
Prio_i	Priority of task τ_i
Run_i	Generic Runnable
$\text{size}(\theta)$	Size of FlexRay slots
Θ	Generic set of FlexRay slots ($\Theta \subseteq \Theta_{\text{cycle}}$)
Θ_{cycle}	FlexRay communication cycle
θ_i	Generic FlexRay slot
Θ_{m_i}	Set of available FlexRay slots for inter-ECU message m_i
θ_{m_i}	FlexRay slot assigned to inter-ECU message m_i
$\Upsilon_{m_{i,j}}$	Available transmission time interval of j-th instance of m_i
Ξ_{ECU_i}	Set of sets of available FlexRay slots for all inter-ECU messages sent by ECU_i
$\zeta_{m_{(\tau_j,\tau_i),k}}$	Transmission start delay for k-th instance of message $m_{(\tau_j,\tau_i)}$
B	FlexRay base cycle
C_i	Computation time (WCET) of task τ_i
C_{ECU_j,τ_i}	WCET of task τ_i executed on ECU_j
$ct(\tau_i)$	Labeling function to annotate τ_i with its WCET C_i
$D_i/D_{i,j}$	Relative deadline of task τ_i (j-th instance)
$d_i/d_{i,j}$	Absolute deadline of task τ_i (j-th instance)
$D_{m_i}/D_{m_{i,j}}$	Relative deadline of message m_i (j-th instance)

$d_{m_i}/d_{m_{i,j}}$	Absolute deadline of message m_i (j-th instance)
$ds(m_i)$	Labeling function to annotate m_i with its data size L_{m_i}
$f_i/f_{i,j}$	Finishing time of task τ_i (j-th instance)
f_{θ_i}	Finishing time of FlexRay slot θ_i
$f_{\theta_{m_i}}$	Finishing time of FlexRay slot assigned to inter-ECU message m_i
$f_{m_i}/f_{m_{i,j}}$	Transmission finishing time of message m_i (j-th instance)
G_{COM}	Communication graph
G_{DG}	Design graph
G_{HAG}	Hardware architecture graph
G_{TDG}	Task dependency graph
G_{TMG}	Task mapping graph
H_Γ	Hyperperiod of task set Γ
J_i	Real-time task
L_{m_i}	Length of message m_i
m_i	Generic message
$m_{i,j}$	j-th instance of message m_i
$R_i/R_{i,j}$	Response time of task τ_i (j-th instance)
$r_i/r_{i,j}$	Release time of task τ_i (j-th instance)
$r_{m_i}/r_{m_{i,j}}$	Release time of message m_i (j-th instance)
Rep	FlexRay cycle repetition
$s_i/s_{i,j}$	Start time of task τ_i (j-th instance)
s_{θ_i}	Start time of FlexRay slot θ_i
$s_{\theta_{m_i}}$	Start time of FlexRay slot assigned to inter-ECU message m_i
$s_{m_i}/s_{m_{i,j}}$	Transmission start time of message m_i (j-th instance)
T_i	Period of task τ_i
T_{m_i}	Period of message m_i
U	Utilization

List of Figures

List of Tables

List of Algorithms

List of Listings

List of Own Publications

[KKM11] K. Klobedanz, A. Koenig and W. Mueller. "A Reconfiguration Approach for Fault-tolerant FlexRay Networks". In: *Design, Automation and Test in Europe (DATE) 2011*. 2011.

[Klo+09] Kay Klobedanz, Christoph Kuznik, Ahmed Elfeky and Wolfgang Mueller. "Development of Automotive Communication Based Real-Time Systems - A Steer-by-Wire Case Study". In: *International Embedded Systems Symposium (IESS) 2009*. 2009.

[Klo+10a] Kay Klobedanz, Bertrand Defo, Wolfgang Mueller and Timo Kerstan. "Distributed Coordination of Task Migration for Fault-Tolerant FlexRay Networks". In: *Proceedings of the fifth IEEE Symposium on Industrial Embedded Systems (SIES2010)*. 2010.

[Klo+10b] Kay Klobedanz, Christoph Kuznik, Andreas Thuy and Wolfgang Mueller. "Timing Modeling and Analysis for AUTOSAR-Based Software Development - A Case Study". In: *Design, Automation and Test in Europe (DATE) 2010*. Dresden, Germany, 2010.

[Klo+10c] Kay Klobedanz et al. "Task Migration for Fault-Tolerant FlexRay Networks". In: *Distributed, Parallel and Biologically Inspired Systems - 7th IFIP TC 10 Working Conference, DIPES 2010*. 2010.

[Klo+11] K. Klobedanz, A. Koenig, W. Mueller and A. Rettberg. "Self-Reconfiguration for Fault-Tolerant FlexRay Networks". In: *Second IEEE Workshop on Self-Organizing Real-Time Systems (SORT) 2011*. 2011.

[Klo+13] Kay Klobedanz, Jan Jatzkowski, Achim Rettberg and Wolfgang Mueller. "Fault-Tolerant Deployment of Real-Time Software in AUTOSAR ECU Networks". In: *International Embedded Systems Symposium (IESS) 2013*. 2013.

[KMR12] K. Klobedanz, W. Mueller and A. Rettberg. "An Approach for Self-Reconfiguring and Fault-Tolerant Distributed Real-Time Systems". In: *Third IEEE Workshop on Self-Organizing Real-Time Systems (SORT) 2012*. 2012.

Bibliography

[3TU10] 3TU - The three leading universities of technology in the Netherlands. *3TU MSc in Embedded Systems*. Brochure. 2010. URL: http://www.3tu.nl/en/publications/2010_3tu_brochure_es.pdf (visited on 01/11/2013).

[ABJ01] Bjorn Andersson, Sanjoy Baruah and Jan Jonsson. "Static-Priority Scheduling on Multiprocessors". In: *Proceedings of the 22Nd IEEE Real-Time Systems Symposium*. RTSS '01. Washington, DC, USA: IEEE Computer Society, 2001.

[Abs12] AbsInt Angewandte Informatik GmbH. *aiT: Worst-Case Execution Time Analyzers*. 2012. URL: http://www.absint.com/ait/ (visited on 22/11/2012).

[AE12] M. Amerion and M. Ektesabi. "A Survey on Scheduling and Optimization Techniques for Static Segment of FlexRay Protocol". In: *Proceedings of the World Congress on Engineering and Computer Science 2012*. Vol. 2. 2012.

[AJ00] B. Andersson and J. Jonsson. "Fixed-priority Preemptive Multiprocessor Scheduling: To Partition or Not to Partition". In: *Proceedings of the Seventh International Conference on Real-Time Systems and Applications*. RTCSA '00. 2000.

[AJ03] B. Andersson and J. Jonsson. "The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%". In: *Proceedings of the 15th Euromicro Conference on Real-Time Systems*. 2003, pp. 33–40.

[AS00] J.H. Anderson and A. Srinivasan. "Early-release fair scheduling". In: *12th Euromicro Conference on Real-Time Systems (Euromicro RTS 2000)*. 2000, pp. 35–43.

[AS01] J.H. Anderson and A. Srinivasan. "Mixed Pfair/ERfair scheduling of asynchronous periodic tasks". In: *13th Euromicro Conference on Real-Time Systems*. 2001, pp. 76–85.

[ASH06] Eric Armengaud, Andreas Steininger and Martin Horauer. "Automatic Parameter Identification in FlexRay Based Automotive Communication Networks". In: *11th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'06)* (Sept. 2006).

[Aud+91] N.C. Audsley, A. Burns, M. F. Richardson and A. J. Wellings. "Hard Real-Time Scheduling: The Deadline-Monotonic Approach". In: *Proceedings of 8th IEEE Workshop on Real-Time Operating Systems and Software*. 1991.

[Aud+93] N. Audsley et al. "Applying New Scheduling Theory to Static Priority Pre-Emptive Scheduling". In: *Software Engineering Journal* 8 (1993).

[AUT13a] AUTOSAR Development Cooperation. *AUTOSAR (AUTomotive Open System ARchitecture) Homepage*. 2013. URL: http://www.autosar.org/ (visited on 31/07/2013).

[AUT13b] AUTOSAR Development Cooperation. *AUTOSAR Methodology, Ver. 3.0.0*. AUTOSAR Release 4.1. 2013. URL: http://www.autosar.org/download/R4.1/AUTOSAR_TR_Methodology.pdf (visited on 31/07/2013).

[AUT13c] AUTOSAR Development Cooperation. *Complex Driver Design and Integration Guideline, Ver. 1.0.0*. AUTOSAR Release 4.1. 2013. URL: http://www.autosar.org/download/R4.1/AUTOSAR_EXP_CDDDesignAndIntegrationGuideline.pdf (visited on 03/08/2013).

[AUT13d] AUTOSAR Development Cooperation. *Guide to Modemanagement 2.0.0*. AUTOSAR Release 4.1. 2013. URL: http://www.autosar.org/download/R4.1/AUTOSAR_EXP_ModemanagementGuide.pdf (visited on 04/09/2013).

[AUT13e] AUTOSAR Development Cooperation. *Layered Software Architecture*. AUTOSAR Release 4.1. 2013. URL: http://www.autosar.org/download/R4.1/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf (visited on 31/07/2013).

[AUT13f] AUTOSAR Development Cooperation. *Software Component Template, Ver. 4.3.0*. AUTOSAR Release 4.1. 2013. URL: http://www.autosar.org/download/R4.1/AUTOSAR_TPS_SoftwareComponentTemplate.pdf (visited on 31/07/2013).

[AUT13g] AUTOSAR Development Cooperation. *Spec. of Basic Software Mode Manager 1.3.0*. AUTOSAR Release 4.1. 2013. URL: http://www.autosar.org/download/R4.1/AUTOSAR_SWS_BSWModeManager.pdf (visited on 04/09/2013).

[AUT13h] AUTOSAR Development Cooperation. *Spec. of Communication, Ver. 5.0.0*. AUTOSAR Release 4.1. 2013. URL: http://www.autosar.org/download/R4.1/AUTOSAR_SWS_COM.pdf (visited on 27/08/2013).

[AUT13i] AUTOSAR Development Cooperation. *Spec. of ECU Configuration, Ver. 3.0.0*. AUTOSAR Release 4.1. 2013. URL: http://www.autosar.org/download/R4.1/AUTOSAR_TPS_ECUConfiguration.pdf (visited on 07/08/2013).

[AUT13j] AUTOSAR Development Cooperation. *Spec. of FlexRay Interface, Ver. 4.0.0*. AUTOSAR Release 4.1. 2013. URL: http://www.autosar.org/download/R4.1/AUTOSAR_SWS_FlexRayInterface.pdf (visited on 31/07/2013).

[AUT13k] AUTOSAR Development Cooperation. *Spec. of Timing Extensions, Ver. 2.0.0*. AUTOSAR Release 4.1. 2013. URL: http://www.autosar.org/download/R4.1/AUTOSAR_TPS_TimingExtensions.pdf (visited on 31/07/2013).

[AUT13l] AUTOSAR Development Cooperation. *Spec. of Watchdog Manager, Ver. 2.3.0*. AUTOSAR Release 4.1. 2013. URL: http://www.autosar.org/download/R4.1/AUTOSAR_SWS_WatchdogManager.pdf (visited on 03/08/2013).

[AUT13m] AUTOSAR Development Cooperation. *Specification of BSW Module Description Template, Ver. 2.3.0*. AUTOSAR Release 4.1. 2013. URL: http://www.autosar.org/download/R4.1/AUTOSAR_TPS_BSWModuleDescriptionTemplate.pdf (visited on 27/08/2013).

[AUT13n] AUTOSAR Development Cooperation. *Specification of Operating System, Ver. 5.1.0*. AUTOSAR Release 4.1. 2013. URL: http://autosar.org/download/R4.1/AUTOSAR_SWS_OS.pdf (visited on 27/08/2013).

[AUT13o] AUTOSAR Development Cooperation. *Specification of PDU Router, Ver. 4.0.0*. AUTOSAR Release 4.1. 2013. URL: http://www.autosar.org/download/R4.1/AUTOSAR_SWS_PDURouter.pdf (visited on 27/08/2013).

[AUT13p] AUTOSAR Development Cooperation. *Specification of RTE, Ver. 3.3.0*. AUTOSAR Release 4.1. 2013. URL: http://www.autosar.org/download/R4.1/AUTOSAR_SWS_RTE.pdf (visited on 31/07/2013).

[AUT13q] AUTOSAR Development Cooperation. *System Template, Ver. 4.3.0*. AUTOSAR Release 4.1. 2013. URL: http://www.autosar.org/download/R4.1/AUTOSAR_TPS_SystemTemplate.pdf (visited on 07/08/2013).

[AUT13r] AUTOSAR Development Cooperation. *Virtual Function Bus, Ver. 3.0.0*. AUTOSAR Release 4.1. 2013. URL: http://www.autosar.org/download/R4.1/AUTOSAR_EXP_VFB.pdf (visited on 31/07/2013).

[Bak05] Theodore P. Baker. "An Analysis of EDF Schedulability on a Multiprocessor". In: *IEEE Transactions on Parallel and Distributed Systems* 16.8 (2005).

[Bar+96] Sanjoy K. Baruah, N. K. Cohen, C. Greg Plaxton and Donald A. Varvel. "Proportionate Progress: A Notion of Fairness in Resource Allocation". In: *Algorithmica* 15.6 (1996), pp. 600–625.

[BB09] Theodore P. Baker and Sanjoy K. Baruah. "An Analysis of Global Edf Schedulability for Arbitrary-deadline Sporadic Task Systems". In: *Real-Time Syst.* 43.1 (2009), pp. 3–24.

[BBD11] S.K. Baruah, A. Burns and R.I. Davis. "Response-Time Analysis for Mixed Criticality Systems". In: *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*. 2011.

[BCL05] M. Bertogna, M. Cirinei and G. Lipari. "Improved schedulability analysis of EDF on multiprocessor platforms". In: *Proceedings. 17th Euromicro Conference on Real-Time Systems, 2005. (ECRTS 2005)*. 2005, pp. 209–218.

[BF07] Sanjoy K. Baruah and Nathan Fisher. "The Partitioned Dynamic-priority Scheduling of Sporadic Task Systems". In: *Real-Time Syst.* 36.3 (2007), pp. 199–226.

[BHR90] Sanjoy K. Baruah, Rodney R. Howell and Louis Rosier. "Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic, Real-Time Tasks on One Processor". In: *Real-Time Systems* 2 (1990).

[Bla76] Jacek Blazewicz. "Scheduling Dependent Tasks with Different Arrival Times to Meet Deadlines". In: *Proceedings of the International Workshop organized by the Commision of the European Communities on Modelling and Performance Evaluation of Computer Systems*. 1976.

[Blo+12a] Hans Blom et al. *TIMMO-2-USE D11: Language Syntax, Semantics, Metamodel V2*. Project Deliverable. Version 1.2. TIMMO-2-USE Consortium, Aug. 2012. URL: http://www.timmo-2-use.org/deliverables/TIMMO-2-USE_D11.pdf (visited on 04/12/2012).

[Blo+12b] Hans Blom et al. *TIMMO-2-USE D14: Validator Documentation*. Project Deliverable. Version 0.9. TIMMO-2-USE Consortium, Aug. 2012.

[BMS08] V. Bonifaci, A. Marchetti-Spaccamela and S. Stiller. "A Constant-Approximate Feasibility Test for Multiprocessor Real-Time Scheduling". In: *Proceedings of the European Symposium on Algorithms*. 2008, pp. 210–221.

[BMW13] BMW CarIT GmbH. *BMW CarIT GmbH - Projects: AUTOSAR Timing Specification*. 2013. URL: http://www.bmw-carit.com/projects/autosar-timing-specification.php (visited on 29/08/2013).

[Bre+08] Robert Brendle et al. "Dynamic Reconfiguration of FlexRay Schedules for Response Time Reduction in Asynchronous Fault-tolerant Networks". In: *Proceedings of the 21st International Conference on Architecture of Computing Systems - ARCS 2008*. Springer-Verlag, 2008, pp. 117–129.

[Bur91] A. Burns. *Scheduling Hard Real-Time Systems: A Review*. 1991.

[But05] Giorgio C. Buttazzo. "Rate monotonic vs. EDF: judgment day". In: *Real-Time Systems* 29.1 (2005).

[But11] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. 3rd. Springer, 2011.

[BW09] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. 4th. Addison-Wesley, 2009.

[Car+04] John Carpenter et al. "A categorization of real-time multiprocessor scheduling problems and algorithms". In: *Handbook on Scheduling Agorithms, Methods, and Models*. Chapman Hall/CRC, Boca, 2004.

[CGP99] Edmund M. Clarke Jr., Orna Grumberg and Doron A. Peled. *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.

[CJ97] R. Chow and T. Johnson. *Distributed operating systems and algorithms*. Addison-Wesley, 1997.

[CLR90] T.H. Cormen, C.E. Leiserson and R.L. Rivest. *Introduction to Algorithms*. The MIT electrical engineering and computer science series. MIT Press, 1990.

[Con12] TIMMO-2-USE Consortium. *TIMMO-2-USE Project Homepage*. 2012. URL: www.timmo-2-use.org (visited on 04/12/2012).

[Cot+02] Francis Cottet, Joelle Delacroix, Claude Kaiser and Zoubir Mammeri. *Scheduling in Real-Time Systems*. Wiley, 2002.

[CSB90] H. Chetto, M. Silly and T. Bouchentouf. "Dynamic scheduling of real-time tasks under precedence constraints". In: *Journal of Real-Time Systems* 2 (1990).

[DB11] Robert I. Davis and Alan Burns. "A Survey of Hard Real-time Scheduling for Multiprocessor Systems". In: *ACM Comput. Surv.* 43.4 (2011). ISSN: 0360-0300.

[DD86] S Davari and S. K. Dhall. “On a Periodic Real-Time Task Allocation Problem”. In: *Proceedings of 19th Annual International Conference on System Sciences*. 1986.

[Dev12] J.L. Devore. *Probability & Statistics for Engineering and the Sciences*. Brooks/-Cole, 2012.

[Din+05] S. Ding, N. Murakami, H. Tomiyama and H. Takada. “A GA-based scheduling method for FlexRay systems”. In: *EMSOFT ’05: Proceedings of the 5th ACM international conference on Embedded software*. 2005.

[Din10] Shan Ding. “Scheduling Approach for Static Segment using Hybrid Genetic Algorithm in FlexRay Systems”. In: *IEEE 10th International Conference on Computer and Information Technology (CIT) 2010*. 2010.

[DL78] Sudarshan K. Dhall and C. L. Liu. “On a Real-Time Scheduling Problem”. In: *Operations Research* 26.1 (1978), pp. 127–140.

[Dri+03] K. Driscoll, B. Hall, Hakan Sivencrona and P. Zumsteg. “Byzantine Fault Tolerance, from Theory to Reality.” In: *SAFECOMP*. Ed. by Stuart Anderson, Massimo Felici and Bev Littlewood. Lecture Notes in Computer Science. 2003.

[DTT08] S. Ding, H. Tomiyama and H. Takada. “An Effective GA-Based Scheduling Algorithm for FlexRay Systems”. In: *IEICE - Trans. Inf. Syst.* (2008).

[EDB10] J. Erickson, U. Devi and S. Baruah. “Improved Tardiness Bounds for Global EDF”. In: *22nd Euromicro Conference on Real-Time Systems (ECRTS)*. 2010, pp. 14–23.

[Eis+10] Friedrich Eisenbrand et al. “Scheduling periodic tasks in a hard real-time environment”. In: *Proceedings of the 37th international colloquium conference on Automata, languages and programming*. ICALP’10. Springer-Verlag, 2010.

[EJ09] C. Ebert and C. Jones. “Embedded Software: Facts, Figures, and Future”. In: *Computer* 42.4 (2009), pp. 42–52.

[Eng12] IAV Automotive Engineering. *Vehicle Electrical System*. 2012. URL: `http://www.iav.com/en/engineering/vehicle-electronics/vehicle-electrical-system` (visited on 22/10/2012).

[FBB06] Nathan Fisher, Sanjoy Baruah and Theodore P. Baker. “The partitioned scheduling of sporadic tasks according to static-priorities”. In: *Proceedings of the EuroMicro Conference on Real-Time Systems*. IEEE Computer Society Press, 2006, pp. 118–127.

[Fen+06] H. Fennel et al. “Achievements and exploitation of the AUTOSAR development partnership”. In: *SAE Convergence*. 2006.

[FFR12] Christoph Ficek, Nico Feiertag and Kai Richter. “Applying the AUTOSAR timing protection to build safe and efficient ISO 26262 mixed-criticality systems”. In: *Proceedings of the 6th Embedded Real Time Software and Systems (ERTS2)*. 2012.

[Fle05a] FlexRay Consortium. *FlexRay Communications System Protocol Specification Ver. 2.1*. 2005.

[Fle05b] FlexRay Consortium. *FlexRay Requirements Specification Ver. 2.1*. 2005.

[Fle05c] FlexRay Consortium. *Preliminary Central Bus Guardian Specification Ver. 2.0.9*. 2005.

[Fle10] FlexRay Consortium. *FlexRay Communications System Protocol Specification Ver. 3.0.1*. 2010.

[For+10] Julien Forget et al. "Scheduling Dependent Periodic Tasks without Synchronization Mechanisms". In: *Proceedings of the 2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*. 2010.

[GFB03] Joël Goossens, Shelby Funk and Sanjoy Baruah. "Priority-Driven Scheduling of Periodic Task Systems on Multiprocessors". In: *Real-Time Syst.* 25.2-3 (2003).

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[GK07] D. Grossmann and O. Kitt. *Embedded Software for FlexRay Systems*. Technical Article. Vector Informatik GmbH, 2007. URL: `http://www.vector.com/portal/medien/cmc/press/Vector/FlexRay_EmbeddedSoftware_AutomobilElektronik_200706_PressArticle_EN.pdf` (visited on 18/01/2013).

[Gos09] Johannes Gosda. *AUTOSAR Communication Stack*. Tech. rep. Hasso-Platner-Institute für Software, 2009.

[Hag+07] Andrei Hagiescu et al. "Performance Analysis of FlexRay-based ECU Networks". In: *Proceedings of the 44th Annual Design Automation Conference*. DAC '07. ACM, 2007, pp. 284–289.

[Ham03] R. Hammett. "Flight-critical distributed systems: design considerations [avionics]". In: *Aerospace and Electronic Systems Magazine, IEEE* 18.6 (2003), pp. 30–36.

[Hea02] Steve Heath. *Embedded Systems Design*. 2nd. 2002.

[Heb09] Regina Hebig. *Methodology and Templates in AUTOSAR*. Tech. rep. Hasso-Platner-Institute für Software, 2009.

[HLH12] Yu Hua, Xue Liu and Wenbo He. "HOSA: Holistic scheduling and analysis for scalable fault-tolerant FlexRay design". In: *INFOCOM, 2012 Proceedings IEEE*. 2012.

[Hor74] W. Horn. "Some simple scheduling algorithms". In: *Naval Research Logistics Quarterly* 21 (1974).

[Izo+05] V. Izosimov, P. Pop, P. Eles and Z. Peng. "Design optimization of time- and cost-constrained fault-tolerant distributed embedded systems". In: *Proceedings of Design, Automation and Test in Europe, 2005*. 2005, pp. 864–869.

[Izo+06a] V. Izosimov, P. Pop, P. Eles and Z. Peng. "Synthesis of fault-tolerant embedded systems with checkpointing and replication". In: *Third IEEE International Workshop on Electronic Design, Test and Applications (DELTA 2006)*. 2006.

[Izo+06b] V. Izosimov, P. Pop, P. Eles and Z. Peng. "Synthesis of Fault-Tolerant Schedules with Transparency/Performance Trade-offs for Distributed Embedded Systems". In: *Proceedings of Design, Automation and Test in Europe, 2006 (DATE'06)*. Vol. 1. 2006, pp. 1–6.

[Izo+08] V. Izosimov, P. Pop, P. Eles and Z. Peng. "Scheduling of Fault-Tolerant Embedded Systems with Soft and Hard Timing Constraints". In: *Proceedings of Design, Automation and Test in Europe, 2008 (DATE '08)*. 2008, pp. 915–920.

[Jal94] Pankaj Jalote. *Fault Tolerance in Distributed Systems*. Prentice-Hall, 1994.

[Jan+11] K. Jang et al. "Design framework for FlexRay network parameter optimization". In: *International Journal of Automotive Technology* 12 (4 2011).

[Jan07] Winfried Janz. *Time Synchronization to FlexRay in OSEKtime and Autosar OS - Schedule Tables to the FlexRay Global Time*. Technical Article. Vector Informatik GmbH, 2007. URL: `http://vector.com/portal/medien/cmc/speeches/FlexRay_Symposium_2007/FRS07_09_Janz.pdf` (visited on 18/01/2013).

[KF09] Olaf Kindel and Mario Friedrich. *Softwareentwicklung mit AUTOSAR: Grundlagen, Engineering, Management in der Praxis*. 1st. dpunkt, 2009.

[KG94] Hermann Kopetz and Guenter Gruensteidl. "TTP-A Protocol for Fault-Tolerant Real-Time Systems". In: *Computer* 27 (1 1994), pp. 14–23.

[KHM03] N. Kandasamy, J. P. Hayes and B. T. Murray. "Dependable Communication Synthesis for Distributed Embedded Systems". In: *Computer Safety, Reliability and Security Conf.* 2003.

[Kim+11] Junsung Kim et al. "An AUTOSAR-Compliant Automotive Platform for Meeting Reliability and Timing Constraints". In: *Society of Automotive Engineers (SAE) World Congress and Exhibition*. 2011.

[KLR10] Junsung Kim, Karthik Lakshmanan and Ragunathan (Raj) Rajkumar. "R-BATCH: Task Partitioning for Fault-tolerant Multiprocessor Real-Time Systems". In: *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*. CIT '10. 2010.

[KM97] Tei-Wei Kuo and Aloysius K. Mok. "Incremental Reconfiguration and Load Adjustment in Adaptive Real-Time Systems". In: *IEEE Transactions on Computers* 46.12 (1997).

[Kop11] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. 2nd. Springer, 2011.

[Kru09] Alfred Alexander Krupp. "A Verification Plan for Systematic Verification of Mechatronic Systems". PhD thesis. University of Paderborn, 2009.

[KSL95] Andreas Kuehlmann, Arvind Srinivasan and David P. Lapotin. "Verity – a Formal Verification Program for Custom CMOS Circuits". In: *IBM Journal of Research and Development* 39 (1995), pp. 149–165.

[Kue+02] Andreas Kuehlmann, Viresh Paruthi, Florian Krohm and Malay K. Ganai. "Robust Boolean Reasoning for Equivalence Checking and Functional Property Verification". In: *IEEE Trans. on CAD of Integrated Circuits and Systems* 21 (2002), pp. 1377–1394.

[LA90] P. A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice*. 2nd. Springer, 1990.

[LAK92] J.C. C. Laprie, A. Avizienis and H. Kopetz, eds. *Dependability: Basic Concepts and Terminology*. Springer, 1992.

[LDG04] J. M. López, J. L. Díaz and D. F. García. "Utilization Bounds for EDF Scheduling on Real-Time Multiprocessor Systems". In: *Real-Time Syst.* 28.1 (2004), pp. 39–68.

[Len90] Thomas Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley & Sons, Inc., 1990.

[Lev86] Nancy G. Leveson. "Software safety: Why, what and how". In: *ACM Computing Surveys* 18 (1986).

[Lib+99] Frank Liberato, Sylvain Lauzac, Rami G. Melhem and Daniel Mosse. "Fault tolerant real-time global scheduling on multiprocessors." In: *ECRTS'99*. 1999, pp. 252–259.

[Lie+95] Jörg Liebeherr, Almut Burchard, Yingfeng Oh and Sang H. Son. "New Strategies for Assigning Real-Time Tasks to Multiprocessor Systems". In: *IEEE Trans. Comput.* 44.12 (1995), pp. 1429–1442.

[Liu00] J.W.S. Liu. *Real-Time systems*. Prentice Hall, 2000.

[LL73] C. L. Liu and James W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment". In: *Journal of the Association for Computing Machinery* 20.1 (1973).

[LM98] Sylvain Lauzac and Rami Melhem. "Adding Fault-Tolerance to P-Fair Real-Time Scheduling". In: *Proceedings of Workshop on Embedded Fault-Tolerant Systems*. 1998, pp. 34–37.

[Luk+09] M. Lukasiewycz, M. Glaß, J. Teich and P. Milbredt. "FlexRay schedule optimization of the static segment". In: *CODES+ISSS '09: Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*. 2009.

[LW82] Joseph Y.-T. Leung and Jennifer Whitehead. "On the complexity of fixed-priority scheduling of periodic real-time tasks". In: *Performance Evaluation* 2.4 (1982).

[Mar03] P. Marwedel. *Embedded System Design*. Kluwer, 2003.

[MB09] Gary Morgan and Andrew Borg. *Multi-core Automotive ECUs: Software and Hardware Implications*. White Paper. 2009. URL: `http://www.etas.com/download-center-files/products_RTA_Software_Products/Whitepaper_Multicore_en.pdf` (visited on 29/10/2013).

[MC89] M.L. Minges and ASM International. Handbook Committee. *Electronic Materials Handbook: Packaging*. Electronic Materials Handbook, Vol 1. ASM International, 1989.

[MN08] L. Havet M. Grenier and N. Navet. "Configuring the communication on FlexRay: the case of the static segment". In: *ERTS'08*. 2008.

[Nau09] Nico Naumann. *AUTOSAR Runtime Environment and Virtual Function Bus*. Tech. rep. Hasso-Platner-Institute für Software, 2009.

[NNa+05] N.Navet, Y. Song, F. Simonot-Lion and C. Wilwert. "Trends in Automotive Communication Systems". In: *Proceedings of the IEEE*. 2005.

[NW03] P. Niewenhuis and P.E. Wells. *The Automotive Industry and the Environment: A Technical, Business and Social Future*. Woodhead Publishing in Environmental Management Series. CRC Press, 2003.

[OS95] Yingfeng Oh and Sang H. Son. "Allocating Fixed-priority Periodic Tasks on Multiprocessor Systems". In: *Real-Time Syst.* 9.3 (1995), pp. 207–239.

[OS97] Yingfeng Oh and Sang H. Son. "Scheduling Real-Time Tasks for Dependability". In: *Journal of Operational Research Society* 43.6 (1997), pp. 629–639.

[OSE05] OSEK/VDX. *Operating System Specification, Ver. 2.2.3*. 2005. URL: http://portal.osek-vdx.org/files/pdf/specs/os223.pdf (visited on 27/08/2013).

[OSE13] OSEK/VDX. *OSEK VDX Portal Homepage*. 2013. URL: http://www.osek-vdx.org/ (visited on 28/08/2013).

[Par07] Dominique Paret. *Multiplexed Networks for Embedded Systems*. Wiley, 2007.

[Par12] D. Paret. *FlexRay and its Applications: Real Time Multiplexed Network*. Wiley, 2012.

[PEP00] Paul Pop, Petru Eles and Zebo Peng. "Bus access optimization for distributed embedded systems based on schedulability analysis". In: *Design, Automation and Test in Europe (DATE) '00*. 2000.

[PEP99] Paul Pop, Petru Eles and Zebo Peng. "Scheduling with optimized communication for time-triggered embedded systems". In: *Proceedings of the seventh international workshop on Hardware/software codesign (CODES '99)*. 1999.

[Per+12a] M.-A. Peraldi-Frati, H. Blom, D. Karlsson and S. Kuntz. "Timing Modeling with AUTOSAR - Current state and future directions". In: *Design, Automation, and Test in Europe Conference Exhibition*. 2012.

[Per+12b] Marie-Agnès Peraldi-Frati et al. "The TIMMO-2-USE project: Time modeling and analysis to use". In: *ERTS2012 International Congres on Embedded Real Time Software and Systems*. Feb. 2012.

[PMH98] B. Pauli, A. Meyna and P. Heitmann. "Reliability of Electronic Components and Control Units in Motor Vehicle Applications". In: *Verein Deutscher Ingenieure (VDI)*. 1998.

[Pop+06] T. Pop et al. “Timing analysis of the FlexRay communication protocol”. In: *18th Euromicro Conference on Real-Time Systems, 2006.* 2006.

[Pop+07] T. Pop, P. Pop, P. Eles and Zebo Peng. “Bus Access Optimisation for FlexRay-based Distributed Embedded Systems”. In: *Design, Automation Test in Europe Conference Exhibition, 2007 (DATE ’07)*. 2007, pp. 1–6.

[Pra96] Dhiraj K. Pradhan, ed. *Fault-tolerant computer system design*. 1996.

[PS11] Inseok Park and Myoungho Sunwoo. “FlexRay Network Parameter Optimization Method for Automotive Applications”. In: *IEEE Transactions on Industrial Electronics* 58.4 (Apr. 2011).

[Q+00] Xiao Qin, Zongfen Han et al. “Real-time Fault-tolerant Scheduling in Heterogeneous Distributed Systems”. In: *Proceedings of the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications*. 2000, pp. 26–29.

[Q+99] Xiao Qin, Liping Pang et al. “Efficient Scheduling Algorithm with Fault-tolerance for Real-time Tasks in Distributed Systems”. In: *Proceedings of the Proceedings of the 5th International Conference for Young Computer Scientists*. Vol. 2. 1999, pp. 721–725.

[QJS02] Xiao Qin, Hong Jiang and David R. Swanson. “An Efficient Fault-tolerant Scheduling Algorithm for Real-time Tasks with Precedence Constraints”. In: *Proceedings of the 31st International Conference on Parallel Processing in Heterogeneous Systems*. 2002, pp. 360–368.

[Ram+09] Franz Rammig et al. “Basic Concepts of Real Time Operating Systems”. In: *Hardware-dependent Software*. Ed. by Wolfgang Ecker, Wolfang Mueller and Rainer Doemer. Springer, 2009. Chap. 2.

[Rau08] Matthias Rausch. *FlexRay – Grundlagen, Funktionsweise, Anwendung*. Hanser, 2008.

[Rei09] K. Reif. *Automobilelektronik: Eine Einführung für Ingenieure*. Vieweg + Teubner, 2009.

[Ric09] Harald Richter. *Elektronik und Datenkommunikation im Automobil*. Analytical Report. Institut fuer Informatik, Technische Universitaet Clausthal, 2009. URL: http://www.in.tu-clausthal.de/fileadmin/homes/techreports/ifi0905richter.pdf (visited on 04/12/2012).

[RLT78] B. Randell, P. Lee and P. C. Treleaven. “Reliability Issues in Computing System Design”. In: *ACM Computing Surveys* 10.2 (1978).

[Rob03] Robert Bosch GmbH. *(ACC) Adaptive Cruise Control: Bosch Technical Instruction*. BENTLEY ROBERT Incorporated, 2003.

[Rob06] Robert Bosch GmbH. *Safety, Comfort and Convenience Systems: Bosch Technical Instruction*. Wiley, 2006.

[RS06] Bill Rogers and Stefan Schmechting. *FlexRay Message Buffers - The Host View of a FlexRay System*. White Paper. 2006. URL: http://www.ip-extreme.com/downloads/flexray_mb_wp.pdf (visited on 20/11/2012).

[RS94] K. Ramamritham and J.A. Stankovic. “Scheduling algorithms and operating systems support for real-time systems”. In: *Proceedings of the IEEE* 82.1 (1994), pp. 55–67.

[SB02] Anand Srinivasan and Sanjoy Baruah. “Deadline-based Scheduling of Periodic Task Systems on Multiprocessors”. In: *Inf. Process. Lett.* 84.2 (2002).

[SCR09] G.B. Shelly, T.J. Cashman and H.J. Rosenblatt. *Systems Analysis and Design*. Ahelly Cashman Series. Thomson Course Technology, 2009.

[Sem12] NXP Semiconductors. *TJA1080A FlexRay transceiver*. Data Sheet. 2012. URL: `http://www.nxp.com/documents/data_sheet/TJA1080A.pdf` (visited on 18/03/2013).

[Siv+03] Hakan Sivencrona, Per Johannessen, Mattias Persson and Jan Torin. “Heavy-Ion Fault Injections in the Time-Triggered Communication Protocol”. In: *Proceedings of the 1st Latin American Symposium on Dependable Computing*. Vol. 2847. Lecture Notes in Computer Science. 2003.

[SJ08] Robert Shaw and Brendan Jackman. “An Introduction to FlexRay as an Industrial Network”. In: *Proceedings of the 2008 IEEE International Symposium on Industrial Electronics*. July 2008.

[SJ99] Santhanam Srinivasan and Niraj K. Jha. “Safety and Reliability Driven Task Allocation in Distributed Systems”. In: *IEEE Trans. on Parallel and Distributed Systems* 10.3 (1999), pp. 238–251.

[Spe12] Robert N. Charette (IEEE Spectrum). *Nissan Moves to Steer-by-Wire for Select Infiniti Models*. 2012. URL: `http://spectrum.ieee.org/riskfactor/greentech/advanced-cars/nissan-moves-to-steerbywire-for-select-infiniti-models` (visited on 20/11/2012).

[Spu+95] Marco Spuri et al. “Robust Aperiodic Scheduling under Dynamic Priority Systems”. In: *Proceedings of the IEEE Real-Time Systems Symposium*. 1995.

[SR08] O. Scheickl and M. Rudorfer. “Automotive Real Time Development Using a Timing-augmented AUTOSAR Specification”. In: *Proceedings of the 4th European Congress ERTS*. 2008.

[SS09] K. Schmidt and E.G. Schmidt. “Message Scheduling for the FlexRay Protocol: The Static Segment”. In: *IEEE Transactions on Vehicular Technology* 58.5 (2009).

[SS10] K. Schmidt and E.G. Schmidt. “Optimal Message Scheduling for the Static Segment of FlexRay”. In: *IEEE 72nd Vehicular Technology Conference Fall 2010 (VTC 2010-Fall)*. 2010.

[Sun+10] Zheng Sun, Hong Li, Min Yao and Nan Li. “Scheduling Optimization Techniques for FlexRay Using Constraint-Programming”. In: *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int’l Conference on Int’l Conference on Cyber, Physical and Social Computing (CPSCom)*. 2010.

[SZ06] Joerg Schaeuffele and Thomas Zurawka. *Automotive Software Engineering*. 3. Vieweg, 2006.

[Tan+10] B. Tanasa, U.D. Bordoloi, P. Eles and Zebo Peng. "Scheduling for Fault-Tolerant Communication on the Static Segment of FlexRay". In: *IEEE 31st Real-Time Systems Symposium (RTSS2010)*. 2010, pp. 385–394.

[Tan+11] B. Tanasa, U.D. Bordoloi, P. Eles and Z. Peng. "Reliability-Aware Frame Packing for the static segment of FlexRay". In: *Proceedings of the International Conference on Embedded Software (EMSOFT) 2011*. 2011, pp. 175–184.

[Tan95] A.S. Tanenbaum. *Distributed Operating Systems*. Pearson Education, 1995.

[Tea98] X-By-Wire Team. *X-By-Wire - Safety Related Fault Tolerant Systems in Vehicles*. Final Project Report. Version 2.0. Nov. 1998.

[The13] The MathWorks, Inc. *MATLAB/Simulink Homepage*. 2013. URL: `http://www.mathworks.com/products/simulink/` (visited on 07/08/2013).

[VBK10] K. Venkatesh Prasad, M. Broy and I. Krueger. "Scanning Advances in Aerospace & Automobile Software Technology". In: *Proceedings of the IEEE* 98.4 (2010), pp. 510–514.

[Vec12] Vector Informatik GmbH. *Bus Transceiver Overview*. Data Sheet. 2012. URL: `http://www.vector.com/portal/medien/cmc/datasheets/CANcabsCAN-piggy_DATASHEET_EN.pdf` (visited on 18/03/2013).

[War09] Robert Warschofsky. *AUTOSAR Software Architecture*. Tech. rep. Hasso-Platner-Institute für Software, 2009.

[Zen+09] Haibo Zeng et al. "Scheduling the FlexRay bus using optimization techniques". In: *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*. 2009.

[Zen+11] Haibo Zeng, M. Di Natale, A. Ghosal and A. Sangiovanni-Vincentelli. "Schedule Optimization of Time-Triggered Systems Communicating Over the FlexRay Static Segment". In: *IEEE Transactions on Industrial Informatics* 7.1 (2011), pp. 1–17.

[ZMM03] D. Zhu, D. Mosse and R. Melhem. "Multiple-resource periodic scheduling problem: how much fairness is necessary?" In: *24th IEEE Real-Time Systems Symposium (RTSS 2003)*. 2003, pp. 142–151.

[ZS07] Werner Zimmermann and Ralf Schmidgall. *Bussysteme in der Fahrzeugtechnik: Protokolle und Standards*. 2nd ed. Vieweg, 2007.