DISSERTATION

Resource Management in an Integrated Time-Triggered Architecture

ausgeführt zum Zwecke der Erlangung des
akademischen Grades eines

Doktors der technischen Wissenschaften

unter der Leitung von

o. Univ. - Prof. Dr. Hermann Kopetz
Institut für Technische Informatik 182

eingereicht an der

Technischen Universität Wien
Fakultät für Informatik

von

Dipl. Ing. Bernhard Huber
Matr. - Nr. 9926084
Edla 9, 3261 Steinakirchen/Forst

Wien, im Jänner 2008 .....

Resource Management in an Integrated Time-Triggered Architecture

Dynamic resource management is the ability of a system to dynamically modify the allocation of the system's resources to its hosted application subsystems in order to react to changing resource demands or resource availability. Dynamic resource management yields better utilization of the available resources, improved dependability of the system by exploiting spare resources or degraded service modes, and the enabling of power-aware system behavior, which has been identified as one of the grand challenges for today's and future embedded systems. This thesis examines the application of dynamic resource management for an integrated time-triggered system architecture for embedded systems.

An integrated architecture is characterized by the integration of multiple application subsystems within a single distributed system. In order to facilitate composability and robustness, as well as, modular certification of the individual subsystems, a pivotal property of an integrated architecture is to achieve encapsulation of the hosted subsystems and to provide mechanisms for fault-isolation. The Time-Triggered System-on-a-Chip (TTSoC) architecture, which builds the foundation of this thesis, provides this encapsulation for computational and communication resources and achieves fault-isolation by offering a protected, predictable time-triggered Network-on-a-Chip that interconnects physically separated cores, which interact exclusively by the exchange of messages.

The key challenge addressed in this thesis is to preserve the encapsulation and fault-isolation properties of the TTSoC architecture, despite the presence of dynamic resource allocation. Therefore, a solution for dynamic resource management in the TTSoC architecture is presented in this thesis, which unifies those, in general conflicting, properties of an architecture. To this end, we propose a resource management strategy that exploits a priori specified knowledge on the resource requirements of an application for providing its service at different Quality-of-Service (QoS) levels. This enables an off-line analysis to determine the maximum resource requirements that may emerge during the lifetime of the overall system. This guarantees that all application systems will receive a sufficient share of the available resources to execute their functionality (possibly at a degraded service level).

A key characteristic of the resource management solution presented in this thesis is its two-tiered approach: We separate the computation of a resource allocation, which is performed by the Resource Management Authority (RMA), from its verification and execution, which is in the responsibility of the Trusted Network Authority (TNA), each of them realized in physically separated components. This way, we facilitate the development of mixed-criticality systems, i.e., systems hosting applications which exhibit different criticality levels. For safety-critical applications we provide (possibly static) resources guarantees, which are protected by the TNA, while we facilitate the efficient implementation of non safety-critical applications using the services of the RMA.

An experimental validation using a prototype implementation of the TTSoC architecture evaluates the resource management solution. The performed experiments confirm that the resource protection mechanisms of the TNA preserve encapsulation and fault-isolation, even in the presence of a failure of the RMA.

Ressourcenverwaltung in einer integrierten, zeitgesteuerten Architektur

Dynamische Ressourcenverwaltung bezeichnet die Fähigkeit eines Systems, die Zuteilung der verfügbaren Systemressourcen dynamisch zu verändern, um auf eine sich ändernde Verfügbarkeit von Ressourcen beziehungsweise auf sich ändernde Anforderungen an Ressourcen reagieren zu können. Vorteile, die sich durch dynamische Ressourcenverwaltung ergeben, sind die Verbesserung der Ressourcenausnutzung, eine Steigerung der Systemzuverlässigkeit durch Aktivierung von Ersatzressourcen im Fehlerfall, sowie die Ermöglichung von energiebewusstem Systemverhalten. Diese Arbeit untersucht den Einsatz von dynamischer Ressourcenverwaltung in einer integrierten, zeitgesteuerten Architektur für eingebettete Systeme.

Eine integrierte Architektur zeichnet sich durch die Realisierung unterschiedlicher Applikationssysteme innerhalb eines einzelnen verteilten Systems aus. Um eine gute Zusammensetzbarkeit (composability) und Robustheit des Systems zu gewährleisten, sowie eine modulare Zertifizierung einzelner Systemkomponenten zu ermöglichen, ist eine starke Kapselung der einzelnen Applikationssysteme von grundlegender Bedeutung. Die Time-Triggered System-on-a-Chip (TTSoC) Architecture, welche die Grundlage dieser Arbeit bildet, ist eine integrierte Architektur, die diese starke Kapselung sowohl für Kommunikationsressourcen als auch für Rechenressourcen anbietet. Erreicht wird dies vorallem durch ein zeitgesteuertes Network-on-a-Chip (NoC), welches die Verbindung von physikalisch getrennten on-chip Komponenten (cores) herstellt, die ausschließlich durch den Austausch von Nachrichten miteinander interagieren können.

Die vorliegende Arbeit präsentiert eine Lösung für dynamische Ressourcenverwaltung in der TTSoC Architecture, welche die starke Kapselung der Applikationssysteme aufrechterhält. Zu diesem Zweck wird in dieser Arbeit eine Strategie für Ressourcenverwaltung vorgeschlagen, welche eine Analyse der zu erwartenden Ressourcenzuteilung vor der Laufzeit des Systems ermöglicht, um Garantie dafür abgegeben zu können, dass für jede Applikation die notwendigen Ressourcen zur Verfügung stehen. Dafür wird eine Repräsentation der Ressourcenanforderungen vorgeschlagen, die es ermöglicht, die benötigten Ressourcen einer Applikation in unterschiedlichen Betriebsmodi zu erfassen.

Ein wesentliches Merkmal der vorgestellten Lösung zur Ressourcenverwaltung ist ihr zweistufiger Ansatz: Die Berechnung der Ressourcenzuteilung, die in der Resource Management Authority (RMA) ausgeführt wird, ist von der Verifizierung und eigentlichen Ausführung der Ressourcenzuteilung, welche von der Trusted Network Authority (TNA) ausgeführt werden, getrennt. Dadurch wird die Realisierung von mixed-criticality Systemen, das heißt Systeme die Applikationen mit unterschiedlichen Anforderungen bezüglich Zuverlässigkeit beherbergen, erleichtert. Einerseits bietet die TNA Hardwaremechanismen, die statische Ressourcenzuteilungen für sicherheitskritische Anwendungen garantieren, andererseits unterstützt die RMA die effiziente Implementierung von nicht-sicherheitskritischen Applikationen. Im Rahmen dieser Arbeit wurde eine Implementierung und Evaluierung der dynamischen Ressourcenverwaltung durchgeführt. Anhand einer beispielhaften Anwendung aus dem Automobilbereich wird gezeigt, dass die geforderte Kapselung der Applikationssysteme erhalten bleibt.

Danksagung

Diese Arbeit entstand im Rahmen meiner Forschungs- und Lehrtätigkeit am Institut für Technische Informatik, Abteilung für Echtzeitsysteme, an der Technischen Universität Wien. Besonders danken möchte ich dem Betreuer meiner Dissertation, Prof. Dr. Hermann Kopetz, der mir die Forschungstätigkeit am Institut ermöglichte und meine Arbeit durch wertvolle Anregungen und Diskussionen unterstützte.

Ich möchte allen Kollegen am Institut für das angenehme Arbeitsklima danken, insbesondere Roman Obermaisser und Christian El Salloum für die unzähligen Diskussionen sowie für die konstruktiven und kritischen Anregungen.

Zudem danke ich meinem Bruder Wolfgang Huber, Markus Bauer, Armin Wasicek, Harald Paulitsch und Roman Obermaisser für das gewissenhafte Korrekturlesen und für die wertvollen Hinweise bei der Arbeit an dieser Dissertation.

Spezieller Dank gebührt auch meinen Eltern sowie allen Freunden für ihre Unterstützung. Ganz besonders danken möchte ich meiner Freundin Verena Katinger – für ihre Geduld, ihr Verständnis und ihre Unterstützung während der Zeit, in der ich diese Arbeit verfasst habe.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Embedded systems are ubiquitous in our everyday's life. Today, the spectrum of potential applications of embedded systems ranges from simple controllers typically used in domestic appliances like washing machines, electric cookers, or microwave heaters, over complex consumer electronics devices, which have to be capable to carry out a wide variety of functions, to ultra dependable applications where enormous costs and the safety of human lives depend on the correct operation of the embedded system.

The amazing advancements of the processing power of embedded systems in the last years have laid the foundation of embedded applications with breathtaking functionality and complexity. This evolution in embedded systems has also influenced the design paradigm of embedded systems: Until now, architectures for embedded systems are typically designed according to the *federated principle*, which means dedicated computer systems are used for implementing individual subsystems of an overall system. Nowadays, there is shift towards the design of architectures for embedded systems according to the *integrated principle* in many application domains, in the e.g., avionics domain and the automotive domain. In integrated architectures, multiple application subsystems share a single distributed computer system, which promises costs savings due to the efficient utilization of the hardware, as well as, improvements of the system's reliability.

Linked to this development, the former strict separation of embedded systems to ones that are deployed in non safety-critical applications and to others that are deployed in safety-critical applications becomes more and more softened. For the near future embedded systems are possible that act as music players, but may also monitor vital body functions and give an alarm if safe boundaries are violated. This integration of multiple application subsystems, possibly possessing different levels of criticality, into a single embedded system increases the need for system architectures that facilitate the development of integrated mixed-criticality systems.

These mixed-criticality systems have significant effects on the system design paradigm. In general, the design of a real-time system highly depends on the targeted application domain. For safety-critical applications, for instance, it is a mandatory

requirement that the resources provided by the system cover the worst-case load that may emerge throughout its entire lifetime. Apparently, this often causes an oversizing of the system compared to the resource requirements in the average case. In contrast, for non safety-critical applications basically economic forces drive the design of the system to cope with the average case of resource usage or to support dynamic resource (re)allocation in order to improve the resource utilization. Especially for mobile devices, it is of utmost importance that system architectures enable an efficient utilization of the available resources, in particular, the dynamic management of power consumption.

1.1 Problem Definition

The above stated advancements of embedded systems open up a set of, partly contradicting, challenges for future system architectures.

Non-uniform resource demands and resource availability. Many current embedded systems are dynamic, i.e., the resource demand of the embedded system may change over time, e.g., due to peaks in service demand. Hence, future system architectures need to support the dynamic modification of resource allocations to individual application systems by applying dynamic resource management. The dynamic resource management has to take into account the actual resource demands of the application system (e.g., communication resources, computational resources, power, etc.), the availability of resources in the embedded system, and—for utility-driven systems [Yeo and Buyya, 2007]—the benefit for the user arising from the service of the respective application system. Changes in the availability of resources originate, for instance, from the (permanent) failure of system components.

Power-aware system behavior. For mobile embedded systems, the required effective utilization of the available power necessitates the realization of power-aware systems. Power-aware design of a system deals with the development of techniques and algorithms that influence the system's behavior in order to meet power and energy requirements under given performance constraints or vice versa [Unsal and Koren, 2003]. Power awareness and power management are also identified as one of the grand challenges in the SIA's semiconductor road map [SIA, 2005] for future embedded systems.

Fault-isolation and composability despite dynamic resource management. For the development of complex embedded systems, it has to be ensured that upon the incremental integration of subsystems, the prior services of the already existing subsystems are not invalidated. This property, denoted as *composability* [Kopetz and Obermaisser, 2002], as well as, fault isolation provided by the architecture are required for the seamless integration of independently developed application subsystems. When combining the requirement for efficient system design

and the support for fault-isolation and composability, two conflicting requirements for resource management arise: On the one hand, the resource management solution should support resource guarantees, which are designed to meet the worst-case load of safety-critical application systems, and permit a static analysis of the resource demands of the individual application subsystems. On the other hand, it should provide flexibility with respect to resource allocation for being competitive in the realization of non safety-critical application systems.

Cost-effective development of embedded systems. A recent trend for the cost-effective development of distributed embedded systems is to decouple application development from the implementation of the hardware platform by applying *model-driven development* approaches. This enables the evolution of the hardware platform while minimizing the need for adaptation of the application. In addition, model-driven development copes with the increasing complexity of today's systems by elevating the level of abstraction for system design. Model-driven development approaches for integrated, mixed-criticality system, such as the one devised for the European project *Dependable Embedded Components and Systems* (DECOS) [Huber and Obermaisser, 2007], need to be revised in order to cope with the new requirements imposed by dynamic resource management, e.g., evolutionary resource demands of applications and variable resource availability.

1.2 Contribution

This thesis presents a solution for dynamic resource management in an integrated time-triggered architecture that addresses the above mentioned challenges. The solutions proposed in this thesis are especially tailored to support mixed-criticality systems. The major contributions of this work are:

Dynamic adaptation of the time-triggered communication schedule. For distributed real-time systems implementing safety-critical applications like steer-by-wire and break control in the automotive domain, time-triggered systems based on the *Time-Triggered Architecture* (TTA) [Kopetz and Bauer, 2003] are highly appropriate due to their high predictability and determinism. However, for applications having less stringent requirements, static allocation of resources often constitutes a too restrictive limitation. In this thesis, a dynamic resource management solution is presented that addresses the challenges of changing load patterns of applications and the variable resource availability by dynamically adapting the communication schedule of the time-triggered communication system. This resource management solution not only enables improved resource efficiency, but also preserves the predictability and determinism of the time-triggered communication system.

Separation of computation and verification of dynamic resource allocation. Resource management is considered in this thesis as an *integral part* of an

architecture rather than a feature that is added later on. Consequently, we define in this work the services and the architectural elements for dynamic resource management in an integrated time-triggered architecture. We distinguish two distinct architectural elements dealing with resource management: the *Resource Management Authority* (RMA) and the *Trusted Network Authority* (TNA). While the RMA is responsible for the computation of the resource allocation, its verification and actual execution is performed by the TNA. The two-tiered design yields the following benefits:

- **Facilitation of system certification.** Certification of the system plays a cardinal role for the development of safety-critical applications. Due to the two-tiered solution for resource management, the RMA has only to be certified to that level of criticality that is demanded by the most critical application system that requires dynamic resource management. Thus, for ultra-dependable systems for which a static resource allocation is often feasible, only the TNA has to be certified up to the highest criticality levels. In addition, by keeping the design of the TNA less complex than the design of the RMA, this separation into RMA and TNA reduces the required effort for certifying ultra-dependable systems.

- **Support for mixed-criticality systems.** Mixed-criticality systems refer to systems that run applications of different criticality classes on the same hardware. This improves the resource utilization of the hardware and is one objective for the design of the resource management solution described in this thesis. The separation of the computation of the resource allocation from its verification enables, on the one hand, (possibly static) resource guarantees protected by the TNA, which are mandatory for systems up to the highest criticality classes. On the other hand, it provides the required flexibility for an efficient implementation of systems with lower criticality by facilitating the dynamic re-allocation of resources by the RMA. Furthermore, the presented resource management solution facilitates fault-isolation and composability by using the TNA to verify and protect changing resource demands of individual applications.

- **Flexibility w.r.t. resource management strategy.** The interface between RMA and TNA cleanly decouples the computation of the resource allocation from its verification and execution. Thus, as long as the RMA adheres to this interface specification, it can be adapted without having any influence on the TNA. This is a vital property of this resource management solution, since a change within the TNA would entail a cost- and time-intensive re-certification at a higher criticality level. For instance, by extending the functionality of RMA and adding the overall power dissipation of the system to the optimization goals of the resource management strategy, power-aware system behavior can be realized.

Foundation for model-driven development methodology. We present in this thesis a model-driven approach for design and development of integrated mixed-criticality systems that is inspired by the concepts of the *Model Driven Architecture* (MDA) [OMG, 2003]. This model-driven development methodology builds the foundation for cost-efficient development of embedded systems.

Experimental validation. The practical feasibility of the devised resource management solution is demonstrated by a prototype implementation. This prototype forms a distributed system using single board computer nodes and *Time-Triggered Ethernet* (TTE) [Kopetz et al., 2005] as the time-triggered communication protocol. A case study demonstrates the on-line management of the communication resources (i.e., on-line reconfiguration of the TTE network) in an exemplary, fictive automotive application consisting of one control and one multimedia application subsystem. An experimental validation of this prototype shows that despite the presence of dynamic resource management, encapsulation of the application subsystems can be achieved. Such an encapsulation yields a reduction of complexity of the overall system, because the behavior of interfering subsystems is more difficult to understand than the behavior of cleanly encapsulated subsystems. Furthermore, it represents an important cornerstone for supporting fault-isolation and composability.

1.3 Structure of the Thesis

This thesis is structured as follows: Chapter 2 introduces the basic terms and concepts that are used throughout this thesis. Section 2.1 gives a brief introduction on integrated architectures while Section 2.2 is devoted to resource management in distributed systems. Thereafter, Section 2.3 explains the concepts of model-driven design and development focusing on the development methodology that has been devised during the European FP6 project DECOS.

Chapter 3 provides an outline of the *Time-Triggered System-on-a-Chip* (TTSoC) architecture, for which the resource management solution introduced in this thesis has been devised. Section 3.1 gives a synopsis of the architecture, whereas Section 3.2 covers the component model of the TTSoC architecture, i.e., a description of its constituting architectural elements. Subsequently, Section 3.3 is devoted to the modeling of applications within the TTSoC architecture including a description of the logical system structuring of applications and the specification of the communication topology.

Chapter 4 is concerned with a comparison of different policies for management and allocation of resources. Section 4.1 introduces an exemplary application scenario on behalf of which the different policies are analyzed. This section is followed by a discussion of different resource allocation policies, namely static resource allocation, dynamic resource allocation, and *Quality of Service* (QoS)-based resource allocation. The chapter is concluded by a comparison of these different approaches.

In Chapter 5 we describe the resource management solution for the TTSoC architecture. First, the requirements of resource management in an integrated architecture are stated in Section 5.1, while Section 5.2 explains the overall resource management strategy for the TTSoC architecture, which has been devised during this thesis. This is followed by a description of the resources that are subject to reconfiguration in Section 5.3. Thereafter, the two architectural elements of the TTSoC architecture that are devoted to resource management are introduced, namely the *Trusted Network Authority* (TNA) in Section 5.4 and the *Resource Management Authority* (RMA) in Section 5.5.

The design and implementation of a case study is outlined in Chapter 6. The case study realizes an emulation of a *System-on-a-Chip* (SoC) component by single board computers interconnected by a time-triggered communication system. The setup of the case study is outlined in Section 6.2. The subsequent sections, Section 6.3 and Section 6.4, cover the implementation of TNA and RMA, respectively.

The evaluation of the proposed resource management solution is summarized in Chapter 7. Three validation experiments are discussed: The first experiment described in Section 7.1 demonstrates the non-interference of the adaptation of the communication schedule with already ongoing communication activities. Section 7.2 gives an evaluation of the ability of the RMA to handle excessive resource requests from hosts. Section 7.3 investigates the resource protection mechanisms of the TNA.

Finally, the thesis ends with a conclusion in Chapter 8 summarizing the main contributions of the presented work and providing an outlook to future work in this research area.

Chapter 2

Basic Concepts and State-of-the-Art

The principles used throughout this thesis span several fields of research. This chapter introduces the concepts on which the work in this thesis is based. Since this thesis mainly focuses on dynamic resource management for integrated mixed-criticality systems, this chapter starts with an introduction of integrated architectures tailored to automotive, avionic, and industrial application domains. After elaborating on the differences between federated and integrated system architectures, representatives of integrated architectures for the above mentioned application domains, namely *Integrated Modular Avionics* (IMA), *Automotive Open System Architecture* (AUTOSAR), and *Dependable Embedded Components and Systems* (DECOS), are introduced. The next section is dedicated to resource management in distributed systems. The issue of resource management is viewed from different perspectives, namely resource management in real-time systems and large networked systems, as well as, the possibility to exploit resource management for building power-aware systems. The chapter is concluded by an introduction of the concepts of model-driven design. After the presentation of the general concepts of model-driven design, the focus is directed on the *Model Driven Architecture* (MDA) as a prominent representative of model-driven design methodologies and on the model-driven design and development methodology developed within the European project DECOS.

2.1 Integrated Architectures for Real-Time Systems

At present, the majority of distributed systems are designed according to the principle of *federated architectures*. In a federated system, each application subsystem (e.g., the primary flight control in avionic systems, powertrain or multimedia subsystem in a car) has its own dedicated computer system (possibly with internal redundancy), which is often specially designed for a particular application [Swanson, 1998]. In addition, federated systems have been the first choice for the realization of ultra-

dependable applications due to their natural separation of application functions, which is beneficial with respect to fault-isolation and complexity management.

The strengths of an integrated system, on the other hand, arise from the integration of multiple application subsystems within a single distributed computer system. Thereby, the resources of the distributed computer system can be exploited more efficiently than in federated systems, which promise massive cost savings (e.g., due to the reduction of the total number of required *Electronic Control Units* (ECUs) in a car). In the automotive domain, this is—in conjunction with the ability to facilitate the implementation of innovative electronic functions—the main driver for the development of today's automotive electronic systems [Fennel et al., 2006].

The following section motivates the paradigm shift towards integrated architecture by a comparison of the advantages of federated and integrated systems, which have been thoroughly analyzed by the authors of [Kopetz et al., 2004]. Afterwards, an introduction to state-of-the-art integrated architectures, namely IMA, AUTOSAR, and DECOS, is given, which have impacted the conceptual design of the *Time-Triggered System-on-a-Chip* (TTSoC) architecture [Kopetz, 2005].

2.1.1 Paradigm Shift to Integrated Architectures

A major advantage of federated systems is their inherent support for managing system complexity. Since each application subsystem is deployed on its dedicated computer system, unintended side effects—such as the impairment of the temporal predictability of intra-subsystem communication due to message transmissions originating from outside the application subsystem—are ruled out by design. Thus, it is sufficient to analyze each application subsystem on its own to reason about the overall behavior. In addition, as a direct consequence of the almost complete independence of the application subsystems in federated systems, federated systems facilitate independent development. Thus, the coordination of different subsystem vendors is reduced to a minimum.

The superior fault containment of federated systems compared to integrated systems is of particular importance, especially in the area of ultra-dependable systems. For ultra-dependable systems an accepted assumption in the scientific community states, that a hardware fault always affects an entire node computer [Lala and Harper, 1994, Kopetz, 2003]. As a consequence, a hardware fault hitting a node computer in a federated system will impair only a single application subsystem, while in an integrated system all application subsystems, which share resources of the affected node computer, might be affected.

Although these attributes of federated systems have shaped the design of distributed systems in many application domains, a paradigm shift towards integrated systems is observable (e.g., IMA in the avionics domain or AUTOSAR in the automotive domain), since they are expected to outperform federated systems mainly by the ability to reduce hardware costs, to improve the overall system dependability by

reducing wires and connectors, and to improve the coordination between application subsystems [Kopetz et al., 2004].

The massive deployment of independent computer systems in today's federated systems (e.g., the BMW 7 series cars contain up to 70 ECUs [Deicke, 2002]) is more and more reaching its limits. Due to the increasing system complexity, the state-of-the-art practice of deploying a vast number of ECUs, each dedicated to a single function, becomes too costly. The potential of reducing the number of computer systems promises massive cost savings. This is even amplified, as the majority of innovations is expected to concern *Electric/Electronic* (E/E) developments. In the automotive domain, for example, up to 90% of all innovations are attributed to E/E [Scharnhorst et al., 2005].

In addition, the increasing number of node computers in federated systems has also a negative effect on the system's reliability: The increase of node computers inherently entails an increase of wires and connectors, which have been identified as a non-negligible source of electrical failures. For instance, in automotive environments more than 30% of electrical failures are ascribed to connector problems [Swingler and McBride, 1998].

The subsequent subsections introduce three state-of-the-art integrated architectures. *Integrated Modular Avionics* (IMA), which is addressed in the following section, is an integrated systems approach for avionics applications. Subsequently, an introduction to *Automotive Open System Architecture* (AUTOSAR) is given, which focuses the automotive domain. We finish by describing the *Dependable Embedded Components and Systems* (DECOS) architecture, which aims at spanning across multiple application domains, including the avionics, automotive, and industrial control domain.

2.1.2 Integrated Modular Avionics

Starting in the mid 1980s, the avionics industry addresses the emerging dominance of software used in aircrafts by devising novel design methodologies for the next generation avionics. This revolution of the design of avionics systems results in the ARINC Standard 651 [ARINC, 1991b], which is known as *Integrated Modular Avionics* (IMA).

Before the beginning of the area of IMA, avionics systems were typically designed as federated systems that comprise multiple *Line Replaceable Units* (LRUs) [Fraboul and Martin, 1998]. An LRU is a modular software/hardware component that implements a specific function of the overall avionics application. These LRUs communicate with each other with point-to-point communication protocols like ARINC 429 [ARINC, 2001]. The federated approach in combination with the increase of software complexity in avionics systems increases the number of deployed LRUs, the required cabling, as well as, costs and weight.

IMA addresses these deficiencies and represents an integrated system architecture that focuses on: *(i)* the use of shared resources for reducing unwanted resource

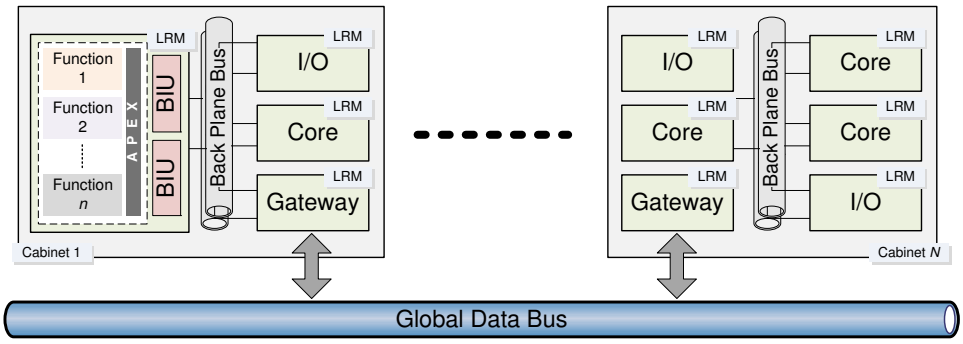


Figure 2.1: IMA avionics architecture

duplication to a minimum for lowering the acquisition costs, weight, and volume of avionics equipment, *(ii)* the support of modular interchangeable hardware components that allow a high volume production, which will positively affect the production costs, and *(iii)* the introduction of improved diagnostic techniques to improve the scheduling of maintenance actions and reduce and eliminate the unconfirmed removal of LRUs [Prisaznuk, 1992]. In the Boeing 787 Dreamliner, e.g., the use of the IMA approach enables a weight reduction of about 900 kg compared to previous aircrafts [Ramsey, 2007].

In the following subsections, we introduce the typical system structure of an avionics system according to the IMA principle and further focus on the software environment for avionics applications.

Avionics System Structure

The hardware platform of an avionics system that is designed according to the IMA is specified in the ARINC Standard 651 [ARINC, 1991b]. As depicted in Figure 2.1, the functionality of the avionics application is provided by multiple integrated *cabinets* which are interconnected by the *global data bus*. The functionality provided by one integrated cabinet is typically larger than the functionality of a single federated LRU, but smaller than the sum of all LRUs the cabinet replaces [Hoyme and Driscoll, 1993].

For example the *Airplane Information Management System* (AIMS) for the Boeing 777, which is one of the first systems that implements IMA concepts, replaces the conventional LRUs by two integrated cabinets [Driscoll and Hoyme, 1992]. The global data bus in the AIMS is realized by the ARINC 629 multi-transmitter communication bus [ARINC, 1991a]. However, systems designed according to IMA are not restricted to particular communication networks. In the Airbus A380, e.g., *Avionics Full-Duplex Switched Ethernet* (AFDX), an ARINC 664 standard network [ARINC, 2002], is deployed for the interconnection of the integrated modules [Brajou and Ricco, 2004].

A cabinet is internally further structured into multiple *Line Replaceable Modules* (LRMs), which provide the necessary computational resources for performing the

required application functionality. LRMs in an IMA platform can be classified into three categories [Fraboul and Martin, 1998] (cf. Figure 2.1):

Core module: Core modules are responsible for the execution of the applications. Typically, a single core module hosts multiple applications, which reside in dedicated encapsulated partitions ensuring that the individual applications do not interfere with each other.

I/O module: System components that are not part of the cabinet or not connected to the global data bus are usually connected by point-to-point communication protocols like ARINC 429 [ARINC, 2001] to the I/O modules of the cabinet. These I/O modules provide the functionality to perform input/output operations with system components.

Gateway module: The gateway module is a specific LRM that handles the communication between the individual cabinets over the global data bus.

Considering the example presented above: for providing the functionality of Boeing 777's AIMS (e.g., flight management, display control, communication management, etc) each cabinet of the AIMS comprises 10 active LRMs and three spare LRMs for future functionality [Morgan, 1991].

The interconnection of the LRMs within a single cabinet is established by a *backplane bus*—a fault-tolerant bus using a *Time Division Multiple Access* (TDMA) scheme for bus arbitration. The backplane bus is specified in the ARINC standard 659 [ARINC, 1993]. A commercial implementation of this standard is Honeywell's *SAFEbus* [Hoyme and Driscoll, 1993], which is deployed, e.g., in the AIMS of the Boeing 777. The SAFEbus backplane bus is accessed by an LRM via a so-called *Bus Interface Unit* (BIU). All transmission or reception operations of the BIU are a priori scheduled and stored in a memory table within the LRM that is inaccessible by the functions of the cabinet. This way, a faulty function is prevented to effect the timing behavior of the backplane bus by changing the LRM's configuration [Hoyme and Driscoll, 1993].

Avionics Software Environment

One fundamental aim of IMA is the integration of multiple avionics functions on a lower number of physical resources compared to former federated system architectures. The *Avionics Application Software Standard Interface* specified in the ARINC 653 standard [ARINC, 2003] defines the services of the avionics software environment, which serve as the basis for avionics function integration. This standard interface, which is known as *APplication EXecutive* (APEX), provides services for partition management, process management, memory management, time management, inter-partition communication, intra-partition communication, and diagnosis:

Partition Management. For the integration of multiple avionic functions on a single LRM, partition management establishes spatial and temporal partitioning [Rushby, 1999] for the individual functions. Therefore, each function is executed in a single partition. For temporal partitioning, the partition management performs a cyclic scheduling with fixed priorities [Lee et al., 1998]. Each partition has assigned two constant parameters—period and duration—that specify the amount of time at which the partition has exclusive access to the LRM's resources (e.g., processing resources) [Audsley and Wellings, 1996].

Memory Management. For spatial partitioning, each partition has assigned a constant (defined at design time) memory area that can be exploited by its hosted function. Any memory access violating these boundaries is prohibited by a *Memory Management Unit* (MMU).

Process Management. Each partition comprises one or more processes that implement its avionic function. All processes share the resources of a single partition. With respect to other partitions, the processes are executed concurrently. Based on the attributes of a process (e.g., a given period for a periodic process, the process's deadline, and the priority of the process [Audsley and Wellings, 1996]), the process management is responsible for scheduling the processes within a partition.

Time Management. Time management in APEX provides system calls for the activation (release) of periodic and aperiodic processes. Aperiodic processes are characterized by the fact, that the future instants of activation are not known a priori (e.g., aperiodic processes could be triggered after the occurrence of a specific event like the reception of a message). For example, LynxOS-178 [LynuxWorks, 2007], a real-time operating system that establishes the APEX interface to its applications, provides the system calls `TIMED_WAIT` and `PERIODIC_WAIT` for time management.

Communication. APEX supports inter-partition and intra-partition communication services. Inter-partition communication is realized via message passing over physical channels and logical ports. Logical ports represent the communication endpoints within the partition. Multiple ports can be mapped onto a single physical channel. For inter-partition communication, two variants of message passing are defined in APEX: Using *sampling ports*, the arrival of a new message overwrites the previous contents of the port, i.e., the port is realized by a single message buffer. *Queuing ports*, on the other hand, provide a message queue where incoming messages are stored in *First-In/First-Out* (FIFO) order. In APEX back pressure flow control is used to handle full message queues [Audsley and Wellings, 1996]. For intra-partition communication, standard inter-process communication mechanisms like shared memory and semaphores can be exploited.

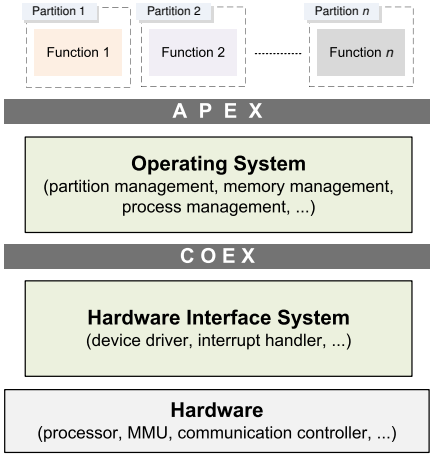


Figure 2.2: APEX avionics software structure

Diagnosis. For the support of diagnosis, the ARINC standard 653 [ARINC, 2003] defines the concept of a *health monitor*, which is responsible for monitoring faults and failures of the hardware, the operating system, and the application. The purpose of the health monitor is to help on isolating faults and preventing the propagation of failures. The response to a fault, i.e., the measures triggered by the health monitor after fault detection, can range from logging of the occurrence of faults, over responses at partition level like the restart of a partition, to a response on the LRM level like reset or shutdown of an entire LRM [Parkinson and Kinnan, 2006].

As depicted in Figure 2.2, the APEX is located between application software and operating system [Audsley and Wellings, 1996]. The operating system itself interfaces the underlying hardware via a standardized interface called *COre EXecutive* (COEX). Together with the *hardware interface system* (cf. Figure 2.2), it is the purpose of the COEX to provide a uniform interface for accessing different implementations of the LRM to the operating system. this facilitates the portability of the operating system. Operating systems that establish the APEX interface are, e.g., LynxOS 178 [LynuxWorks, 2007] or VxWorks 653 Edition [Parkinson and Kinnan, 2006].

2.1.3 Automotive Open System Architecture

The *Automotive Open System Architecture* (AUTOSAR) [Heinecke et al., 2004] is an attempt to exploit the benefits of integrated system architectures in the automotive domain. It is a joint initiative of several automotive, semiconductor, and software companies. AUTOSAR was founded in 2003 and is currently in its final phase of defining a consolidated set of specifications [Heinecke et al., 2006]. According to [Heinecke et al., 2004], the motivations behind this standardization initiative in the automotive domain are:

- Management of E/E complexity associated with growth in functional scope

- Flexibility for product modification, upgrade and update
- Scalability of solutions within and across product lines
- Improved quality and reliability of E/E systems in order to provide a higher level of abstraction

The main objective of AUTOSAR is to facilitate the reuse of AUTOSAR *Software Components* (SW-Cs)—an AUTOSAR SW-C encapsulates an application which runs on the AUTOSAR infrastructure [AUTOSAR GbR, 2006b]—between different vehicle platforms, *Original Equipment Manufacturers* (OEMs), and suppliers [Scharnhorst et al., 2005]. Furthermore, it is envisioned to improve software updates and upgrades over the entire vehicle lifetime [Heinecke et al., 2004]. For these purposes, AUTOSAR defines a standardized software architecture for each ECU in an automotive system that provides a technology-independent, i.e., independent from the ECU hardware and the underlying micro controller, infrastructure for SW-Cs. On one hand, this enables the decoupling between application development and development of the hardware platform of automotive systems. On the other hand, this will support the decoupling between the life-cycles of hardware and software [Scharnhorst et al., 2005]. In addition, AUTOSAR defines a development methodology [AUTOSAR GbR, 2006a] that supports a *distributed, function-driven* development process.

The following subsections introduce the ECU software architecture and the development methodology as defined in [AUTOSAR GbR, 2006b].

AUTOSAR ECU Software Architecture

The structure of the AUTOSAR ECU architecture is schematically depicted in Figure 2.3. According to [AUTOSAR GbR, 2006b], the software architecture of an ECU in AUTOSAR is vertically structured into *Basic Software*, the *AUTOSAR Run Time Environment* (RTE), and *AUTOSAR software*.

The Basic Software is a standardized software layer in each ECU that provides services to the SW-Cs, which are necessary to realize the actual functionality. Examples for those services are memory access, access to the communication system, operating system functionalities, etc. Basic Software itself does not provide any application specific services. It contains standardized, i.e., ECU-independent components, like *Basic Services* and *Microcontroller Abstraction*, as well as, ECU specific components like *ECU Abstraction*, and *Complex Drivers* [Heinecke et al., 2004].

Microcontroller Abstraction: The Microcontroller Abstraction decouples higher layers of the Basic Software from micro controller internals. This layer contains micro controller specific drivers like *Input/Output* (I/O) drivers, memory drivers, and *Analog-to-Digital Converter* (ADC) drivers. It represents the lowest layer of the AUTOSAR Basic Software.

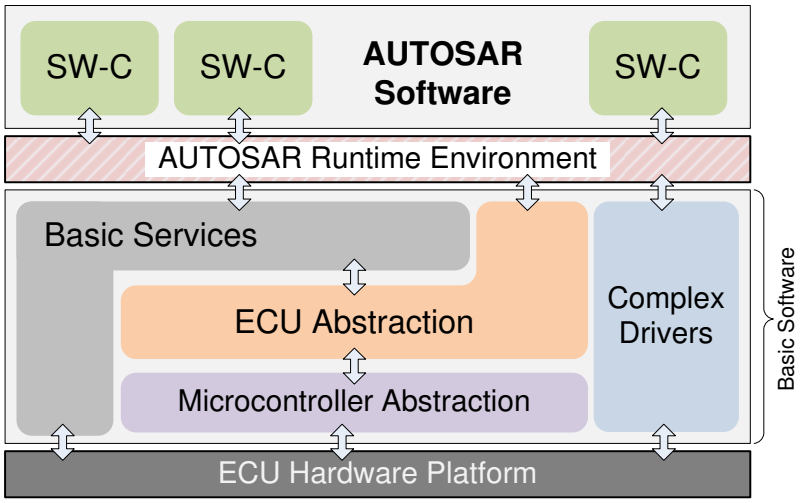


Figure 2.3: AUTOSAR ECU software architecture

ECU Abstraction: The purpose of the ECU Abstraction layer is to hide the upper layers from the layout of the ECU, i.e., to provide an *Application Programming Interface* (API) to access the ECU's peripherals, regardless whether they are micro controller internal or external devices. Since it is built on top of the Microcontroller Abstraction, the implementation of the ECU Abstraction is micro controller independent.

Basic Services: The Basic Services represent the highest layer of Basic Software and are used from the layers above the Basic Software to abstract from ECU and micro controller hardware. The Basic Services include operating system services, vehicle network communication services, memory management services, diagnostic services, and ECU state management.

Complex Drivers: The concept of Complex Drivers is introduced to handle complex sensors and actuators with strong real-time requirements or electromechanical hardware requirements, which cannot be directly mapped to a single layer of the AUTOSAR Basic Software [AUTOSAR GbR, 2006b]. The implementation of a Complex Driver is highly dependent on the micro controller and the ECU hardware. However, to an upper layer—the AUTOSAR *Run Time Environment* (RTE)—Complex Drivers provide a standardized AUTOSAR interface.

Between the Basic Software and the application software resides the AUTOSAR RTE. The purpose of the RTE is to provide a uniform environment to all SW-Cs, i.e., to abstract from any implementation details of the Basic Software and from hardware aspects [Heinecke et al., 2006]. The RTE can be seen as the runtime representation of the *Virtual Function Bus* (VFB) on a specific ECU [AUTOSAR GbR, 2006b]. The VFB provides standardized communication services to the application software, which are defined independently whether the communication manifests af-

ter the integration of the system in inter-ECU or intra-ECU information exchange [Heinecke et al., 2004]. This way, the VFB decouples the application from the system infrastructure [Scharnhorst et al., 2005].

An application in AUTOSAR consists of interconnected *AUTOSAR SW-Cs*. SW-Cs are located in the ECU's application layer in the AUTOSAR stack. By introducing the VFB and standardized interfaces, the implementation of such a SW-C is independent from the ECU and the underlying micro controller. In addition, the VFB realizes location independence for the implementation of SW-C, i.e., the implementation of a SW-C has not to be aware of its physical location and the physical location of other SW-Cs.

An AUTOSAR SW-C is an *atomic* component, which means that each instantiation of a SW-C is allocated to exactly one ECU and cannot be distributed over several ECUs. In general, the implementation of an AUTOSAR SW-C is independent from the infrastructure in terms of the type of the micro controller and the ECU the SW-C is located on (due to the Microcontroller Abstraction layer and the ECU Abstraction layer of the Basic Software). In addition, it is in general also independent from the physical location of the SW-C, because of the abstraction provided by the VFB. However, in typical automotive applications there exist SW-Cs which are designed for a specific sensor or actuator (e.g., a car velocity sensor). By the use of a specialized class of SW-Cs—the *Sensor/Actuator Software Components*—such dependencies can be expressed within the AUTOSAR standard.

AUTOSAR Methodology

The AUTOSAR standard [AUTOSAR GbR, 2006a] specifies a development methodology that leads from system design to implementation. The workflow of this methodology is schematically depicted in Figure 2.4. The entire workflow is logically structured into four phases which are explained in the following: *system configuration*, *ECU information extraction*, *ECU configuration*, and *executable generation*.

System Configuration. The purpose of the system configuration phase is mainly to map the SW-Cs of the application, defined in the input *SW-C Description*, to the available ECUs of the hardware platform (defined in the input *ECU Resource Description*) taking resources and timing requirements into account (defined in the input *System Constraints Description*).

The SW-C Description specifies the logical functions of the application that are visible to the environment independent of the available ECUs, networks, network topology, etc. Furthermore, the representation of the SW-C within the SW-C Description is decoupled of any implementation details.

The ECU Resource Description provides a detailed representation of the relevant physical and electronic characteristics of all ECUs in the system. This includes a description of the computational resources provided by the ECU (e.g., processor type,

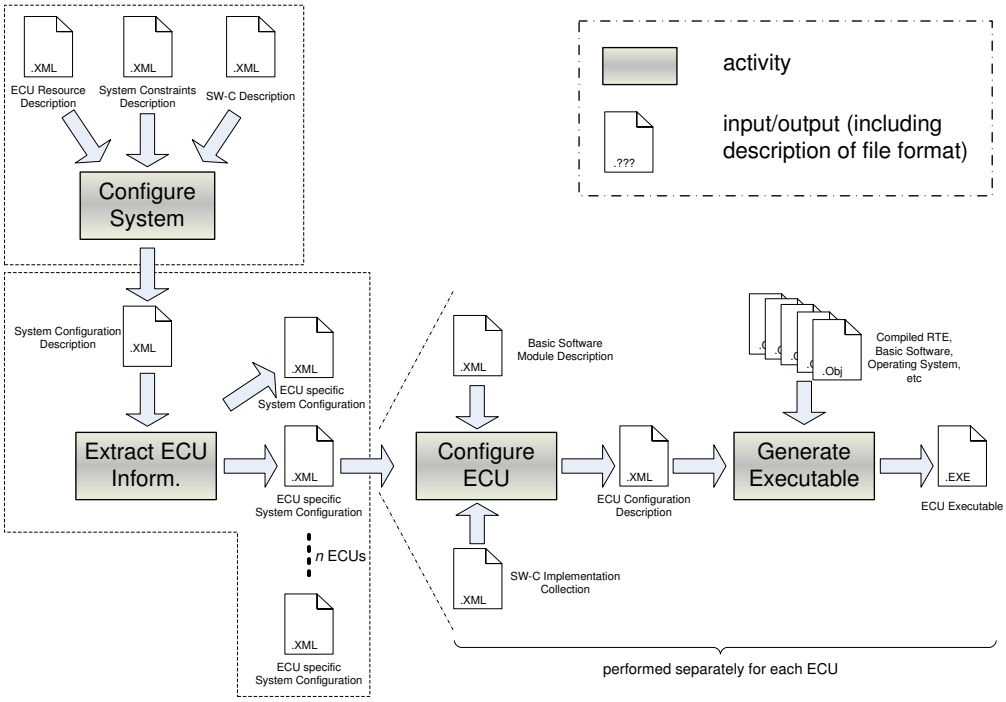


Figure 2.4: AUTOSAR development methodology

amount and type of memory), the connected peripherals (e.g., sensors and actuators), as well as, hardware interfaces (e.g., communication interfaces to *Controller Area Network* (CAN) [Bosch, 1991], *Local Interconnect Network* (LIN) [LIN, 2003], or FlexRay [FlexRay Consortium, 2005]) and physical connectors.

The third input to the system configuration is a description of system constraints, which comprises system wide information that have an effect on the mapping of SW-Cs to ECUs. Examples for such constraints are attributes of deployed communication systems (e.g., bandwidth or latency) or distribution constraints (e.g., distributing replicated SW-Cs of a *Triple Modular Redundancy* (TMR) configuration to separate ECUs).

The output of the system configuration is the *System Configuration Description*, which includes a mapping of SW-Cs to ECUs, a mapping of communication activities over the VFB to physical messages on the deployed communication bus, and finally a description of the bus topology in form of a communication matrix.

ECU Information Extraction. The AUTOSAR RTE of the final system is adapted for each ECU to comply to the needs of the SW-Cs the ECU hosts and further to the resources provided by the ECU's hardware platform. Hence, those parts of the *System Configuration Description* that are relevant for the individual ECUs are extracted and separate *ECU-specific System Configurations* are generated in the ECU information extraction phase. Since no additional information is added in this step of the development methodology, ECU information extraction can be

typically performed completely automatically. The remaining two activities have to be carried out for each ECU individually.

ECU Configuration. During the ECU configuration the necessary information for the final implementation is added. This is typically a non-trivial engineering task. Besides the extracted ECU-specific System Configuration, the *Basic Software Module Description* and a *SW-C Implementation Collection* are used as inputs for this task. The AUTOSAR standard defines the Basic Software Module Description as a definition of all possible configuration parameters of a Basic Software module, which is assumed to be delivered by the vendor of the respective Basic Software Module. The SW-C Implementation Collection is used to decide, which (possibly alternative) implementation of a SW-C is deployed on the ECU, and to configure the RTE according to the requirements of the chosen SW-Cs. The output of the ECU configuration phase is called *ECU Configuration Description* and acts as starting point for the generation of executables.

Executable Generation. The last step includes the generation of code for the RTE and particular Basic Software modules, as well as, the compilation of SW-Cs that are available as source code. This step concludes by linking all objects into an executable.

2.1.4 Dependable Embedded Components and Systems

According to Hammett in [Hammett, 2003, p. 33], an ideal future system architecture *would combine the complexity management advantages of the federated approach, but would also realize the functional integration and hardware benefits of an integrated system.* Addressing this challenge, the major aim of the DECOS architecture [Obermaisser et al., 2006] is to devise a framework for integrating multiple application subsystems within a single, distributed computer system, while retaining the error containment and complexity management benefits of federated systems.

DECOS is an European FP6 integrated project developing the basic enabling technology to move from current domain-specific federated distributed architectures to an integrated distributed architecture. Whereas IMA and AUTOSAR are focused on a single application domain, DECOS spans across multiple application domains, including the avionics, automotive, and industrial control domain. The key intent of DECOS is to develop technology invariant software interfaces and encapsulated virtual networks with predictable temporal properties such that application software can be transferred to a new hardware and communication base with minimal effort.

The DECOS architecture offers a framework for the development of distributed embedded real-time systems and supports the integration of multiple application subsystems with different levels of criticality and different requirements concerning the underlying platform. Furthermore, structuring rules guide the designer in the decomposition of the overall system at a functional level and for the transformation

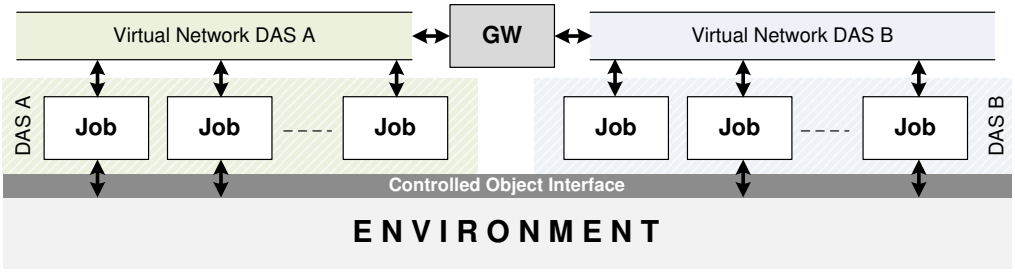


Figure 2.5: Functional structure of a DECOS system

to the physical level. In addition, the DECOS integrated architecture offers generic architectural services to system designers, which provide a validated stable baseline for the development of applications.

Functional System Structuring

The design principle of divide-and-conquer is well suited for sub-dividing a large problem into smaller parts where the mental effort of understanding the individual parts in isolation is reduced. According to this principle, in DECOS the overall system is divided into a set of nearly-independent subsystems - denoted as *Distributed Application Subsystems* (DASs) (cf. Figure 2.5). A DAS is a distributed subsystem of a large distributed real-time system that provides a well-specified application service [Kopetz et al., 2004]. The identification of DASs is guided by the functional coherence and common criticality of subsystems. Examples of DASs in present day automotive applications are comfort electronics, the power-train system, or the multimedia system.

In analogy, each DAS is further decomposed into smaller units called *jobs*. A job is the basic unit of work and exploits a virtual network in order to exchange messages with other jobs to reach a common goal. A virtual network is the encapsulated communication system of a DAS (see Section *Architectural Services* for more details). All communication activities of a virtual network are restricted to the particular DAS, i.e., messages are exchanged only by jobs of the same DAS, unless a message is explicitly exported or imported by a gateway. Furthermore, a virtual network exhibits predefined temporal properties that are independent from other virtual networks.

The access point of a job to a virtual network of its DAS is denoted as *port*. Each port is assigned to exactly one job. Depending on the data direction of the port, input and output ports are distinguished in the DECOS architecture. Furthermore, ports are classified into state ports and event ports, respectively, depending on the information semantics of sent (in case of output ports) or received (in case of input ports) messages.

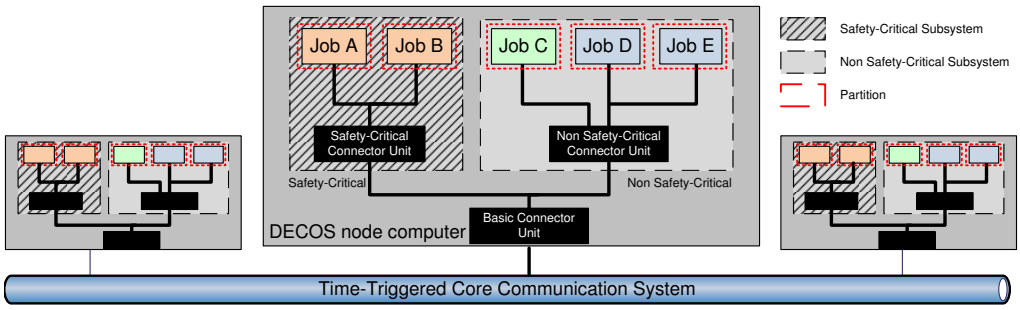


Figure 2.6: Physical structure of a DECOS system

Physical System Structuring

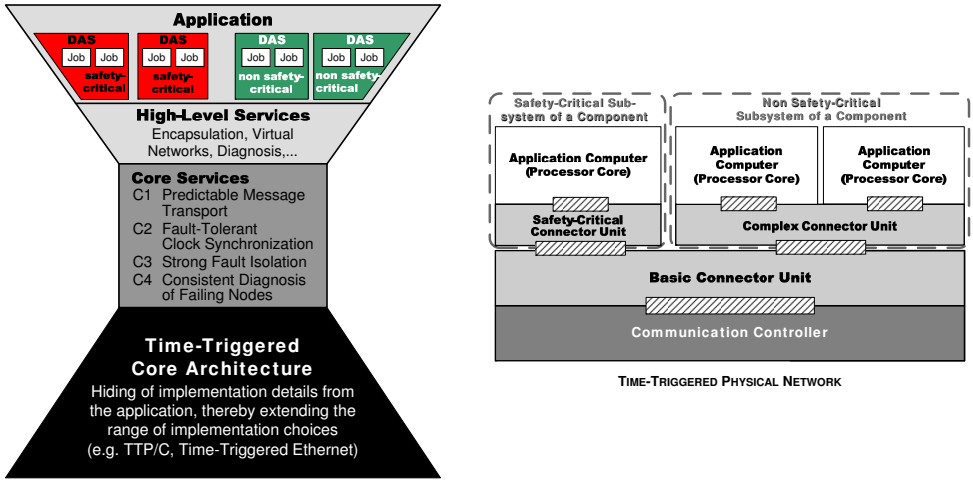
During the development of an integrated system the functional elements, i.e., the jobs, must be mapped onto the physical building blocks provided by the hardware platform. As depicted in Figure 2.6, these building blocks are the time-triggered core communication system, node computers and partitions.

A node computer is a self-contained computational element with its own hardware (processor, memory, communication interface, and interface to the controlled object) and software (application programs, operating system) [Kopetz and Suri, 2003]. The node computers are part of a distributed computer system interconnected by the time-triggered core communication system. Node computers are the target of job allocation and provide encapsulated execution environments denoted as partitions for the execution of jobs. Each partition prevents temporal interference (e.g., stealing processor time) and spatial interference [Rushby, 1999, Huber et al., 2005] (e.g., overwriting data structures) between jobs.

Since the DECOS architecture is targeted at the realization of mixed-criticality systems, i.e., application systems with different dependability requirements share a single integrated node computer, a DECOS node computer entails two classes of partitions: Partitions residing in the safety-critical subsystem for jobs of safety-critical DASs and partitions residing in the non safety-critical subsystem of the node computer for jobs with less stringent dependability requirements.

Architectural Services

The concept of platform-based design [Sangiovanni-Vincentelli and Martin, 2001] proposes the introduction of abstraction layers to separate the application functionality from the underlying platform technology in order to facilitate reuse and reduce design complexity. The architectural services of the DECOS architecture are such an abstraction layer. The specification of the architectural services hides the details of the underlying platform, while providing all information required for ensuring functional and meta-functional (dependability, timeliness) requirements for the design of a safety-critical real-time application. The architectural services serve as a validated



(a) DECOS Integrated System Architecture (b) DECOS Node Computer Model

Figure 2.7: DECOS Integrated Architecture (taken from [Obermaisser et al., 2006])

stable baseline that reduces application development efforts and facilitates reuse, because applications build on an architectural service interface that can be established on top of numerous platform technologies.

In order to maximize the number of potential platforms and applications, the DECOS architectural service interface distinguishes a minimal set of core services and an open-ended number of high-level services which are built on top of the core services (depicted by the waistline in Figure 2.7(a). The core services include predictable time-triggered message transport, fault tolerant clock synchronization, strong fault isolation, and consistent diagnosis of failing components through a membership service. Any architecture that provides these core services can be used as a core architecture [Rushby, 2001] for an integrated distributed architecture. An example of a suitable base architecture providing those core services is the TTA [Kopetz and Bauer, 2003].

Based on those four core services, the DECOS integrated architecture realizes high-level architectural services, which are specific for particular DASs and constitute the interface for the jobs to the underlying platform. This interface is denoted as *platform interface*. These high-level services include gateway services [Obermaisser et al., 2005a] for the exchange of information across DAS boundaries, virtual network services [Obermaisser et al., 2005b], and encapsulation services [Huber et al., 2005]. The virtual network service is the encapsulated communication infrastructure tailored to the needs of a particular DAS. It is built on top of the time-triggered physical network. In one instantiation of the DECOS integrated architecture, different kinds of virtual networks are established and each type of virtual network can exhibit multiple instantiations.

The purpose of gateway services is to selectively redirect messages between virtual networks of different DASs improving quality of service and eliminate resource

duplication. In addition, the gateway service is responsible to resolve conflicts with respect to operational properties and naming. The encapsulation services control the visibility of exchanged messages and ensure spatial and temporal partitioning for virtual networks and jobs in order to obtain error containment.

The DECOS Node Computer

A DECOS node computer as depicted in Figure 2.7(b) is vertically structured into two subsystems. The *safety-critical subsystem* serves as an encapsulated execution environment for ultra-dependable applications, while the *non safety-critical subsystem* offers an environment for those applications having less stringent dependability requirements. For the latter applications, the emphasis lies on low-cost, flexibility, and resource efficiency instead of predictability and support for certification as it is the case for applications within the safety-critical subsystem.

Besides vertical structuring, DECOS node computers are also horizontally structured into *application computers*, *connector units*, and the *communication controller*. The purpose of the application computer is to establish encapsulated partitions for the execution of jobs. As depicted in Figure 2.7(b) connector units interconnect the application computers and the communication controller, which provides access to the time-triggered core network. The connector unit ensures that each subsystem of the node computer obtains a predefined share of the overall network resources by controlling the application computer's access to the state message interface provided by the communication controller. This way, the connector unit enables each subsystem to exchange messages with guaranteed temporal properties (e.g., maximum latency and latency jitter of message transmissions) and data integrity and ensures temporal and spatial partitioning of communication resources.

We distinguish between three types of connector units (cf. Figure 2.7(b)). The *Basic Connector Unit* (BCU) performs the primary allocation of the physical network resources, as required for the separation of safety-critical and non safety-critical subsystems. Within the safety-critical subsystem, the *Safety-Critical Connector Unit* (SCU) allocates network resources to jobs and realizes the safety-critical high-level services (e.g., voting functionality). The *Complex Connector Unit* (XCU), on the other hand, performs the allocation of network resources for the non safety-critical subsystem. As it can be seen in Figure 2.7(b) the XCU does not directly access the communication controller, but it is stacked on top of the BCU. This way, the XCU is not involved in the fault isolation and error containment between the safety-critical and non safety-critical subsystem within the node computer. Hence, the XCU and the non safety-critical subsystems of a component need not to be certified to same criticality level as BCU and SCU. Thus, the XCU can provide increased functionality at the cost of increased complexity.

The DECOS architecture does not restrict the choice of implementations of a particular node computer. Although the provision of separate processors (or processor cores) for each partition has significant advantages with respect to certification, a

solution with only one processor shared among the jobs is also an alternative choice. However, this would require from the operating system to provide partitioning mechanisms that can be certified up to the required criticality class. An analysis of the component model with respect to certifiability, encapsulation and independent development aspects is given in [Kopetz et al., 2004].

2.2 Dynamic Resource Management

Many current and future real-time systems are dynamic, i.e., the environmental demands on the service of the real-time system may change over time, e.g., either due to peaks in service demand or to evolutionary changes in external conditions [Bihari and Schwan, 1991]. Therefore, a real-time system that must be able to dynamically change its configuration in order to cope with the changing environment, i.e., it has to support dynamic resource management.

In addition, dynamic resource management allows a more efficient utilization of mutually exclusive resource demands in complex systems, e.g., when it is a priori known that the worst-case resource consumption in different subsystems cannot occur simultaneously. In such systems, the available resources can be alternately allocated to the different subsystems, circumventing the need for oversizing the design of the system, i.e., to calculate with the worst case requirements of both subsystems. Furthermore, if a permanent fault affects only individual computer nodes in a distributed system, dynamic resource management can be used to relocate the application functionality to spare nodes to preserve the specified service.

There exists no universal definition of resource management in the literature, since the role of resource management in a system depends highly on the structure and characteristic of the system, i.e., the resources that are managed, as well as, the requirements of the application realizing the service of the system. A high-level definition of resource management in the context of distributed systems is given in [Veríssimo and Rodrigues, 2001, p. 517] as *the issue of ensuring that distributed systems are configured correctly in order to provide adequate service, and that they remain correctly configured and providing adequate service throughout their life.* In the context of this definition, we understand dynamic resource management as the distribution of the *shared resources* of the system among the hosted application subsystems in such a way that each application subsystem is able to provide its specified service.

In the following two subsections we elaborate on the concepts of resource management in two highly diverse fields of computer science, namely resource management in *(hard) real-time systems* and resource management in *(best effort) large networked systems*, which have inspired the resource management solution of the TTSoC architecture. The section ends by giving an introduction of power-awareness and dynamic power management—one of the grand challenges in future embedded systems [SIA, 2005]—which is made possible by means of resource management techniques.

2.2.1 Classic Resource Management in Real-Time Systems

In the context of real-time systems, the authors of [Murthy and Manimaran, 2001] identify four fundamental issues on resource management, namely task scheduling, resource reclaiming, fault tolerance, and real-time communication.

Task Scheduling. In real-time systems a failure in meeting a task's deadline may result in severe consequences. The challenge of task scheduling is defined in [Ramamritham and Stankovic, 1994] as *the allocation of resources and time to tasks in such a way that certain performance requirements are met.* Scheduling algorithms are typically classified into *preemptive* algorithms (e.g., rate-monotonic and earliest-deadline-first scheduling [Liu and Layland, 1973]) and *non-preemptive* algorithms (e.g., first in first out (FIFO) and shortest process next (SPN) [Stallings, 1998, p. 389ff]). The latter are featured by the fact that once a task is executed, the processor cannot be reallocated before the task has finished. In preemptive algorithms, on the other hand, an executing task can be disrupted by a task with higher priority and is resumed later on.

Resource Reclaiming. For many scheduling algorithms, the worst case computation time of each task forms the basis for calculating the task schedules. Resource reclaiming deals with the problem of utilizing resources which are left unused by a task [Shen et al., 1993]. That allocated resources are not fully used by task may have several reasons: *(i)* a task executes faster than its worst case computation time, or *(ii)* it is removed from the current schedule (e.g., a backup task after the successful completion of the primary task in a primary-backup based fault-tolerant approach [Ghosh et al., 1997]).

Fault Tolerance. In real-time systems—especially in safety-critical systems—fault-tolerance is a vital system characteristic. Otherwise, the failure of a single system component may lead to a failure of the entire system with the potentiality of a catastrophic consequence [Kopetz, 1997]. A system is said to behave fault-tolerant, if it incorporates *additional components and abnormal algorithms which attempt to ensure that occurrences of erroneous states do not result in later system failures* [Randell et al., 1978, p. 6]. One basic principle of fault-tolerant design is *redundancy* (e.g., in the domains of time and/or space). The allocation of sufficient resources to redundant components (e.g., in a TMR configuration), is one important challenge of resource management in order to facilitate fault-tolerant distributed real-time systems.

Real-Time Communication. The adherence of a distributed real-time system to its temporal specifications can only be ensured, if the communication between distributed tasks occurs over a dedicated *real-time communication system.* As elaborated in [Kopetz, 1997] the requirements on real-time communication systems include: low protocol latency with minimal jitter, support for composability, and the need for error detection at the receiver. With respect to resource management, in distributed systems the challenge of task scheduling is

extended by the problem of finding a communication schedule satisfying the system's temporal constraints.

Resource management for uni- or multi-processor systems is mostly concerned with task scheduling and memory access scheduling. In distributed systems these challenges are extended by the fact that spatial distribution of tasks and the additional complexity introduced by the requisite of a communication system has to be tackled. The issue of scheduling in real-time systems has been extensively researched in the past and deals according to [Ramamritham and Stankovic, 1994] with *the allocation of resources and time to tasks in such a way that certain performance requirements are met.* The authors of [Ramamritham and Stankovic, 1994] classify the multitude of scheduling algorithms into the following four paradigms:

Static table-driven approaches. These approaches have been originally developed for hard real-time systems. In these systems the resources required for meeting the deadlines of the tasks have to be preallocated and guaranteed a priori. The static table-driven scheduling algorithms are characterized by a static schedulability analysis that is performed off-line during the design phase of the system. The arising schedules are stored in tables and denote the points in time when a task must start its execution. A limitation of those approaches is that they require periodic tasks. In addition, those approaches are highly inflexible, since changes to the task characteristics (e.g., period, execution time, resource requirements, etc) may invalidate the off-line generated schedule table.

Priority-driven preemptive approaches. The fundamental concept of priority-driven preemptive scheduling algorithms is that a task of higher priority is able to disrupt the execution of tasks of lower priority and takes over the shared resource (e.g., the processor). There exists no scheduling table as described before, but scheduling decisions are made at runtime. The most prominent representatives are the *Rate-Monotonic* (RM) and the *Earliest-Deadline-First* (EDF) scheduling algorithms [Liu and Layland, 1973]. One can distinguish static (e.g., the RM algorithm) and dynamic priority-driven preemptive scheduling algorithms. While in the RM algorithm each task is assigned a static priority based on its period (shorter period results in higher priority), a dynamic priority assignment is performed in the EDF algorithm—the task with the closest deadline gets assigned the highest priority.

Dynamic planning-based approaches. While for the previously mentioned approaches a schedulability analysis of a given task set is performed off-line, dynamic planning-based algorithms are characterized by performing feasibility checks dynamically during run-time. A task is called *guaranteed* and thus accepted for execution, if a task execution plan can be generated that ensures that all guaranteed tasks will meet their timing constraints [Ramamritham and Stankovic, 1994]. The basis for an algorithm checking the feasibility of a task set is formed by a set of

assumptions on the tasks itself and the system. This includes assumptions about the tasks' resource requirements (e.g., worst case execution time, memory access), as well as, the nature and frequency of faults in the system. Dynamic planning-based scheduling is used for instance in the kernel of the hard real-time operating system *Spring* [Stankovic and Ramamritham, 1991].

Dynamic best effort approaches. As the name implies, in dynamic best effort approaches no feasibility checks are performed. They accept a task for execution and try to do the best for meeting all deadlines. Typically, best-effort approaches are simulated using various load scenarios in order to obtain estimations on the probability of a given task set for meeting all deadlines. Best-effort algorithms—a typical example is CPU time sharing as it is performed by most desktop operating systems—are usually applied to such applications where the effort for determining the worst-case execution time is not feasible. An approach for improving the performance of soft real-time applications on systems using best-effort scheduling is described in [Banachowski and Brandt, 2003].

2.2.2 Resource Management in Large Networked System

Unlike most classical real-time applications (e.g., a flight controller in the avionic domain), which are usually *closed world systems* [Kopetz et al., 2005], large networked systems such as *Grid computing systems* [Foster et al., 2001] are often realized as *open world systems* where *an (unknown) number of uncoordinated clients compete for the services of a server* [Kopetz et al., 2005]. In this section we give an outline of the requirements on resource management in open world system by taking resource management in Grid computing systems as an example.

The term *Grid* denotes a very large-scale distributed computing infrastructure which is typically used for solving advanced scientific or engineering problems like the factorization of large integers or the modeling of the earth's climate [Bernholdt et al., 2005]. The common challenge among the different types of Grid computer systems (e.g., computational Grids, data Grids, or service Grids [Krauter et al., 2002])—also referred to as the *Grid problem*—is the *flexible, secure, coordinated resource sharing among dynamic collections of individuals, institutions, and resources* [Foster et al., 2001]. Thus, resource management for Grid computing systems is a highly active research topic.

A Grid computing system is characterized by the interconnection of a vast number of cooperating *machines*, which are distributed across different organizations and administrative domains and communicate via high-speed communication links. The constituting elements of a Grid computing system can be categorized as *processing elements* (e.g., parallel computers, personal computers, personal digital assistants, etc), *network elements* (e.g., routers, switches, virtual private network devices, etc), and *storage elements* (e.g., network attached storage devices, automated tape libraries, etc) [Krauter et al., 2002]. A central function of each Grid computing system is the *Resource Management System* (RMS). The main responsibility of a RMS

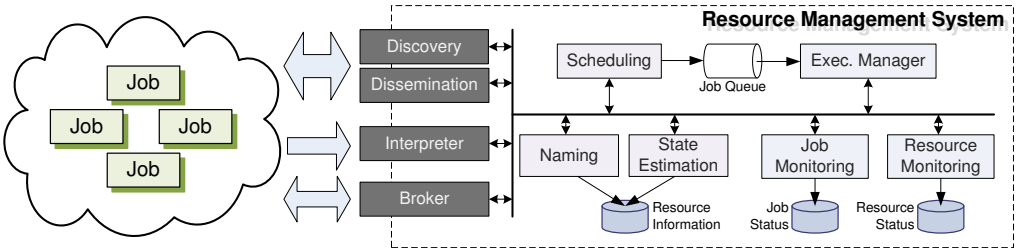


Figure 2.8: Abstract model of a Resource Management System (RMS) (according to [Krauter et al., 2002])

is to accept requests for resources from machines within the Grid, match and assign the requests to available resources, schedule the matched resources, and execute the request, i.e., to actually grant the requesting machine access to a specific available resource in the Grid [Krauter et al., 2002]. The term *resource* refers to all elements of the Grid that are managed by the RMS. An application that utilizes resources of the Grid is called *job* in the following.

The concrete characteristic of a RMS highly depends on both, the nature of the Grid computing system (e.g., computational grid, data grid, or service grid) and the required services. An abstract model of showing the responsibilities of a general RMS is depicted in Figure 2.8. Widely following [Krauter et al., 2002], the services that have to be provided by a RMS can be classified into three classes, namely services provided to jobs, services provided to the native operating system or hardware environments for executing resource management, and internal services.

Service Interface towards Jobs

As indicated in Figure 2.8, the service interface towards jobs includes a resource discovery and dissemination service, a resource interpreter service, and a resource broker service.

It is the purpose of the resource dissemination and discovery service to provide the means by which a job is able to establish a view on the existence of resources and their availability. The mechanism which enables a job to find resource information (e.g., a query to a *Lightweight Directory Access Protocol* (LDAP) compliant directory service) is called *resource discovery*. The *resource dissemination* service is the mechanism to make the resource information available to the jobs (e.g., replication of the database for local access for jobs, or publishing of network addresses of the nearest network directory).

The *resource interpreter* service enables jobs to initiate resource requests. Resource requests are described using a resource description language or protocol, which is consistent in the entire Grid computing system (e.g., the *Grid Resource Access and Management* (GRAM) protocol [Chervenak et al., 2001] in the *Globus* Grid architecture).

The *resource broker* service is used in Grid computing systems which employ market/economy-based resource management mechanisms as the one described in [Buyya et al., 2000]. Market/economy-based resource management mechanisms aim at *encouraging resource owners to contribute their resource(s) for the construction of a Grid and compensate them based on the resource usage or value of work done* [Buyya et al., 2000]. This would enable on demand formation of cooperating groups within the Grid computing system to solve a given problem by having their own private mechanisms for sharing resources and calculating profits among themselves.

Services for Executing Resource Management

The services of the RMS provided for the actual execution of resource management are depicted in Figure 2.8 and include the execution manager, as well as, job monitoring and resource monitoring services.

The execution manager service is responsible for controlling the life time of jobs (i.e., creating, managing, and destroying jobs), as well as, for controlling the execution of the jobs on the machine (e.g., migrate an executable to a particular machine and initiate the job execution by the use of native operating system calls).

The job monitoring service acquires the actual state of the job. This information can be exploited by the RMS to initiate *resource reclamation*, which is the mechanism of the RMS to reclaim a resource hold by a job that has already finished. In addition, the job monitoring service provides the basis for the realization of the *policing* functionality of a RMS. Policing is the process of ensuring that a given contract for resource utilization between the RMS and a particular job is not violated. The resource monitoring service acquires the actual state of all resources in order to enable *resource accrediting* within the RMS, i.e., the ability to keep track of the usage of the individual resources [Krauter et al., 2002].

Internal Services

As the name implies, the internal services do not interact with elements of the Grid computing system outside of the RMS. These services, which include resource naming, scheduling, and state estimation (cf. Figure 2.8), exploit the information provided by the previously described service classes.

The purpose of the naming service is to interact with the resource dissemination and discovery service, as well as, with the request interpreter service and to maintain a database containing the resource information. The design of the naming service determines the organization (e.g., flat or hierarchical) and content of this database, which influences the performance of the resource dissemination and discovery service [Krauter et al., 2002].

The task of the scheduling service is to allocate the available resources to requesting jobs. A central part of this task is *admission control*, i.e., to determine whether

a resource request can be granted or has to be rejected. Therefore, the resource scheduling service exploits the current information provided by the job monitoring service, the resource monitoring service, and the state estimation service to make its scheduling decisions. The state estimation service estimates the future states of resources and jobs by exploiting history information on resource usage and knowledge about the job's resource usage characteristic (e.g., included in the resource request).

Needless to say that not all real-world Grid computing systems implement all mentioned services. A detailed taxonomy of different RMS can be found in [Krauter et al., 2002].

2.2.3 Power-Aware Systems

Dynamic resource management builds an important cornerstone for the realization of power-aware systems. According to the SIA's semiconductor road map [SIA, 2005], power awareness and power management are identified as one of the grand challenges for future embedded systems. In [Unsal and Koren, 2003] the authors define a power-aware system as a system which *modifies its behavior based on current power/energy availability*. Hence, power-aware design deals with the development of techniques and algorithms that influence the system's behavior in order to meet power and energy goals under given performance constraints or vice versa. This is in contrast to low-power design, where the minimization of the power consumption of single gates up to complex SoCs, as well as, the communication among the individual cores on the chip is the primary concern and techniques and methodologies are investigated that optimize the design of those chips.

Sources of Power Dissipation in CMOS Technology

The power dissipation of *Complementary Metal Oxide Semiconductor* (CMOS) devices consists of three major parts contributing to the total power dissipation of the device, namely *dynamic power dissipation*, *short-circuit power dissipation*, and *static power dissipation*. The following expression shows the constituting elements of the total power dissipation of a CMOS device [Benini et al., 2001], which are discussed in detail in the following:

$$P_{tot} = P_{Dynamic} + P_{Short-Circuit} + P_{Static} \tag{2.1}$$

Dynamic Power Dissipation. The first term in Equation 2.1 represents the contribution of dynamic power dissipation. Dynamic power dissipation is issued from charging and discharging of capacitors during switching of gates in CMOS devices. In the literature, it is therefore often referred to as *switching power* [Chandrakasan et al., 1992] or *capacitive power dissipation* [Pedram, 1996].

Equation 2.2 describes the influence of supply voltage, clock frequency, and switching activity on dynamic power dissipation [Chandrakasan et al., 1992].

$$P_{Dynamic} = p_t \cdot (C_L \cdot V \cdot V_{dd} \cdot f_{clk}) \tag{2.2}$$

The dissipated dynamic power is influenced by the load capacitance C_L, the supply voltage V_{dd}, the voltage swing V (usually equal to V_{dd} except to some specialized logic circuit implementations where it my be slightly less than V_{dd} [Yano et al., 1990]), and the clock frequency f_{clk}. The term p_t represents the activity factor, i.e., the probability that a power consuming switching operation occurs. In [Pedram, 1996, Devadas and Malik, 1995, Unsal and Koren, 2003] the switching activity is described by the average number of output transitions in $1/f_{clk}$ time.

Short-Circuit Power Dissipation. The short-circuit power dissipation is caused by the short-circuit current I_{sc} that flows when both, the NMOS and PMOS transistor, are conducting simultaneously (see Equation 2.3 [Chandrakasan et al., 1992]).

$$P_{Short-Circuit} = V_{dd} \cdot I_{sc} \tag{2.3}$$

An alternative representation of the short-circuit power is introduced by the authors of [Devadas and Malik, 1995]. In this work, I_{sc} is modeled as the product of the quantity of charge, the number of transitions per clock cycle, and the clock frequency. With careful design the short-circuit power dissipation can be kept less than 15 % of the dynamic power dissipation [Kang, 2003].

Static Power Dissipation. Dynamic power dissipation was the predominant source of power dissipation in CMOS circuits for fabrication processes of $0.8\,\mu m$ and above [Benini et al., 2001]. Therefore, research in low-power and power-aware design of integrated-circuits was mainly focused on reducing dynamic power dissipation. However, for deep-submicron processes the static power dissipation becomes more important. The static power dissipation results mainly from leakage current between the power supply and the ground (see Equation 2.4).

$$P_{Static} = V_{dd} \cdot I_{Leakage} \tag{2.4}$$

There are three dominant components of leakage current, namely reverse-bias p-n junction leakage, subthreshold leakage, and the gate oxide tunneling current [Johnson et al., 1999]. The primary source of leakage current is typically reverse-bias p-n junction leakage; however, due to ongoing miniaturization of the feature size and reduced supply voltage, sub-threshold leakage and leakage caused by gate oxide tunneling become more important [Keshavarzi et al., 1997]. In [Kang, 2003] various techniques are presented to reduce the leakage current $I_{Leakage}$ by tackling the subthreshold and gate leakage problem.

Power-Aware System Design Techniques

Although for deep-submicron devices the reduction of static power dissipation gains more importance, dynamic power dissipation still contributes a major part to the total dissipated power of a CMOS device. Therefore, in the following we give a short outline on the three degrees of freedom for taking influence on the dynamic power dissipation of a system [Pedram, 1996].

Voltage: The reduction of the supply voltage offers an effective potentiality due its quadratic impact on power. However, reducing the supply voltage implies a speed penalty since the delay of the circuits increases when the supply voltage approaches the threshold voltage of the device. Furthermore, increasing the size of a transistor reduces its delay. Optimal transistor sizing [Kakumu and Kinugaw, 1990, Chandrakasan et al., 1992] is the challenge to find the optimum between the required delay improvements and the dissipated power by exploiting the slack time of the circuit [Devadas and Malik, 1995].

Capacitance: The physical capacitance seen from the individual gates in the circuit has a linear influence on the dissipated power. Thus, reducing this capacitance e.g., by shrinking the devices, using fewer and shorter wires, or in general using less logic, positively influence the dynamic power dissipation [Pedram, 1996]. However, as for voltage scaling, reducing the capacitance may also impact the performance of the circuit (e.g., increases the delay) [Pedram, 1996] and thus results in a trade-off between reducing the physical capacitance and voltage scaling.

Activity: As the name implies, dynamic power or switching power is only dissipated when the device performs actions that cause switching in a circuit. The switching activity is influenced by two components, cf. Equation 2.2: The operating frequency f_{clk} and the activity factor p_t. Slowing down the operating frequency directly influences the performance of the device. Especially in real-time systems, this performance loss may not be tolerated. The switching activity of a circuit depends on a multitude of effects (e.g., input-pattern, delays, path lengths etc.) and is thus hard to estimate. An overview about techniques for estimating the switching activity can be found in [Pedram, 1996].

There are two prominent approaches which exploit the above described degrees of freedom for selectively controlling the execution of tasks according to required performance constraints or available power/energy levels, namely *Dynamic Power Management* (DPM) and *Dynamic Voltage and Frequency Scaling* (DVFS). In general, using DPM techniques means to put a processor in a more power-efficient execution mode whenever it is not required for the execution of tasks, while DVFS techniques try to find an optimal trade-off between power-saving by supply-voltage reduction and performance (due to the lowered execution frequency implied by the

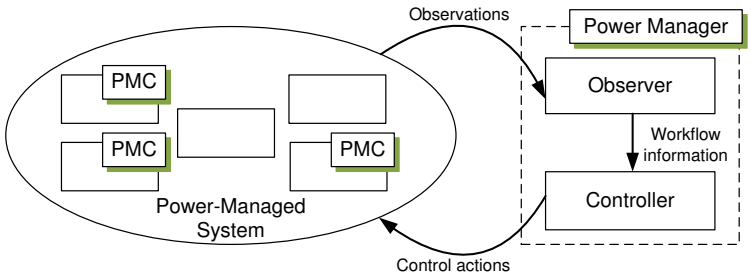


Figure 2.9: Structure of a Power-Managed System (PMS) [Benini et al., 2000])

reduced supply-voltage). DPM techniques are often also applied for I/O devices like hard disks.

Dynamic Power Management

The term *Dynamic Power Management* (DPM) is used to describe a design methodology of systems that perform *energy-efficient computations* by exploiting idle times of system components [Benini et al., 2000, Lorch and Smith, 1998]. Thereby, it is tried to shut-off (or reduce the performance) of system components that are currently not in use or partially unexploited. Systems applying DPM—so-called *Power-Managed Systems* (PMS) [Benini et al., 2000]—strive for providing the requested service with minimum system power by using a minimum number of active components (or minimum load on such components).

For being able to apply DPM, it is mandatory that the individual *Power Manageable Components* (PMCs)—system components that support multiple modes of operation spanning the power-performance trade-off [Benini et al., 2000]—are subject to non-uniform workloads. In addition, it must be possible to predict, up to a certain degree of confidentiality, this non-uniformity of the workload. A typical example for a PMC is a hard disk. Read/write accesses to hard disks are typical workloads with non-uniform characteristics. Furthermore, hard disks exhibiting an idle time longer than a given threshold (e.g., 30 minutes) can be put into standby, since it is assumed that for a sufficiently long time no further accesses will occur.

As depicted in Figure 2.9, a PMS typically comprises a set of interacting PMCs and a *power manager* [Benini et al., 2000]. The power manager is responsible for controlling the mode of operation of the PMCs by applying a control strategy—often called policy—based on observation and assumptions on the workload. The challenge to find such a control strategy that optimally trades off power and performance is elaborated in [Benini et al., 1999].

DPM is supported by several hardware and software vendors (e.g., Hewlett-Packard, Intel, Microsoft, and Toshiba) which have established a standardization initiative that is known as the *Advanced Configuration and Power Interface* (ACPI) [Hewlett-Packard Corp. et al., 2006]. ACPI is an operating system independent power management and configuration standard with the aim of providing a

specification for a flexible interface between power manageable devices (like hard disks) and the power manager. However, ACPI does not provide any strategies for optimal dynamic power management.

Dynamic Voltage and Frequency Scaling

Equation (2.2) clearly shows that lowering the operating voltage of a CMOS circuit has a high potential for reducing the power consumption of the circuit due to the quadratic relationship of voltage and dynamic power dissipation. Even more, by lowering the supply voltage of a CMOS circuit and at the same time lowering the operating frequency—as it is aimed at in DVFS techniques—the power requirement can be reduced by the third power of the relative frequency reduction [Flynn and Hung, 2005]. Reducing the operating voltage and the frequency saves substantial power; however, in the same way the execution of programs is slowed down. Therefore, DVFS heuristics usually trade off power savings against delay [Unsal and Koren, 2003]. For example, a scheduling approach for hard real-time tasks that aims at combining DPM and DVFS techniques is presented in [Rong and Pedram, 2006].

Nowadays, several processor families, as for example the Intel® XScale™ processor family [Intel, 2000], support the optimization of the power dissipation of a processor for the completion of a given workload by adapting the performance per mWatt ratio by DVFS techniques. For instance, the Intel® XScale™ architecture supports on-the-fly scaling of supply voltage and clock frequency covering a performance range from 40 mW/185 MIPS at 150 MHz up to 900 mW/1000 MIPS at 800 MHz [Intel, 2000].

2.3 Model-Driven Design and Development

The current state-of-the-art system development methodologies for distributed embedded systems are heavily imposed to be reviewed, because of the requirement to continuously improve functionality with stringent time-to-market constraints and to cope with changing deployment platforms. In today's development cycles of embedded systems, the validation of the system usually occurs after the integration of the software onto the chosen hardware platform. However, in this late phase of the development process, any changes to the system would result in significant costs.

To circumvent this deficiency, a recent trend in the design and development of distributed embedded systems is to decouple application development from the implementation of the hardware platform and to apply *model-driven* development approaches. In such approaches, the entire system is defined by models with an adequate level of detail that facilitates the generation of the system's implementation. This entails a change of the role of models from a *descriptive* characteristic to a *prescriptive* characteristic in the entire development process [Gruhn et al., 2006, p. 20].

The main benefits of those *model-driven development* approaches are the potential for reducing initial development costs and costs for maintenance, as well as, to facilitate the evolution of the hardware platform by minimizing the need for adaptation of the application by separating *platform-specific* and *platform-independent* aspects of applications and capture them in separate models. A platform-independent model of an application can be used in model-driven development as input to different model transformations resulting in alternative realizations based on diverse platforms [Almeida, 2006]. In addition, by using models of the software as well as of the hardware platform a *virtual system integration* [Giusto et al., 2002] can be applied, which enables the identification of design faults earlier in the development process, thus lowering the costs induced by required changes to the system.

The following section gives an introduction to the *Model Driven Architecture* (MDA) [OMG, 2003], which is a prominent representative of model-driven development. The MDA serves as the inspiration for the model-driven development methodology of DECOS, which is presented in the subsequent section.

2.3.1 Model-Driven Architecture

The MDA is a software development method proposed by the *Object Management Group* (OMG) that aims at the *separation of the specification of the operation of a system from the details of the way that system uses the capabilities of its platform* [OMG, 2003, p. 2–2]. The motivation for model-driven approaches is that platforms are subject to change over time. This entails that software systems are eventually requiring to be deployed on one or more different platforms [Mellor et al., 2002]. Thus, the primary goals of the MDA are *portability*, *interoperability*, and *reusability* of applications achieved by architectural separation of concerns [OMG, 2003, Gruhn et al., 2006]. Therefore, in the MDA the specification of the operation of a system is expressed by different models at various levels of abstraction, with each level emphasizing certain aspects or viewpoints of the system.

Besides the definition of standardized viewpoints (e.g., computation independent, platform-independent, or platform-specific viewpoints, which will be described in the following section), the MDA defines also a process for software development. The key aspects of this process are the specification of a platform-independent representation of the software and the transformation to a platform-specific representation by using transformation rules, which are derived from the model of the platform.

MDA Models at Different Levels of Abstraction

A model of a system is a formal specification of the system and provides an abstraction, i.e., the model includes certain classes of information while suppressing other ones. The selection of the classes of information to include or to suppress depends on the purpose and the focus of the model. A particular selection of such information classes is denoted as a viewpoint. Widely used viewpoints in model-driven development methodologies are *platform-independent* and *platforms-specific*

viewpoints. The MDA proposes such viewpoints denoted as *Computation Independent Model* (CIM), *Platform Independent Model* (PIM), *Platform Model* (PM), and *Platform Specific Model* (PSM) [OMG, 2003].

Computation Independent Model (CIM): The CIM is a view of the system focusing on the computation-independent viewpoint, which describes the environment of the system and requirements for the system while abstracting from the details of its structure. A CIM is typically a domain or business model that provides domain specific information on the system which is independent of the implementation [Zhao, 2005].

Platform Independent Model (PIM): The purpose of the PIM is to describe the operation of a system while still abstracting from the details of a particular platform. The challenge of creating the PIM is to identify and model those parts of the system that do not change from one platform to another. The quality of platform independence of a model is always a matter of degree [OMG, 2003, p. 2–5]: Considering a virtual machine that defines a set of services (e.g., communications, scheduling, naming, etc.). The virtual machine itself is a platform. A model describing an application by the means of the virtual machine's services is platform-specific w.r.t. this virtual machine. However, the same model is platform independent w.r.t. the class of different platforms realizing this virtual machine, because the platform-independent models are unaffected by changes of the underlying platform.

Platform Model (PM): The PM describes the technical concepts that make up the platform and specifies the services the platform provides. In the MDA, the concept of a platform is defined as *a set of subsystems and technologies that provide a coherent set of functionality [. . .], which any application [. . .] can use without concern for the details of how the functionality provided by the platform is implemented* [OMG, 2003, p 2–3].

Platform Specific Model (PSM): The PSM is an extension of the PIM that includes further details on how services of a specific platform are exploited by the system. It is constructed from the PIM by applying a model transformation, which is the process of converting one model to another model of the same system. The details of the model transformation in the MDA are the focus of the next section.

The concepts introduced above are only meaningful w.r.t. a particular point of view [Brown, 2004] on the system. In this sense, a model that is a PSM in one development step serves as the PIM for a further refinement of the system. An illustrative example (taken from [Brown, 2004]) is depicted in Figure 2.10: With respect to the choice of a communication middleware, a model serves as a PIM in the MDA, as long as the model does not dictate a particular middleware technology. After choosing a technology (e.g., CORBA [OMG, 2004]), a transformation to a PSM (a CORBA-specific PSM) is performed. However, with respect to the target

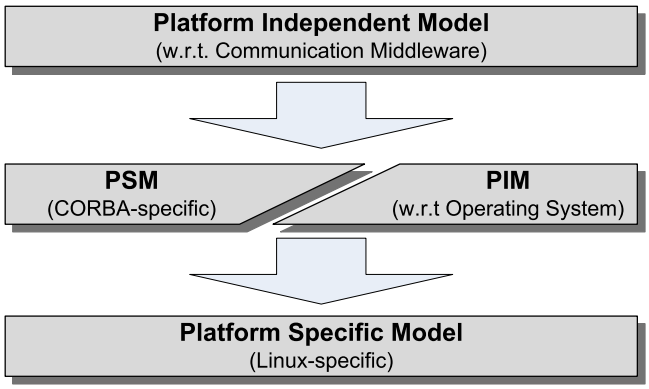


Figure 2.10: Different perceptions of PIM and PSM

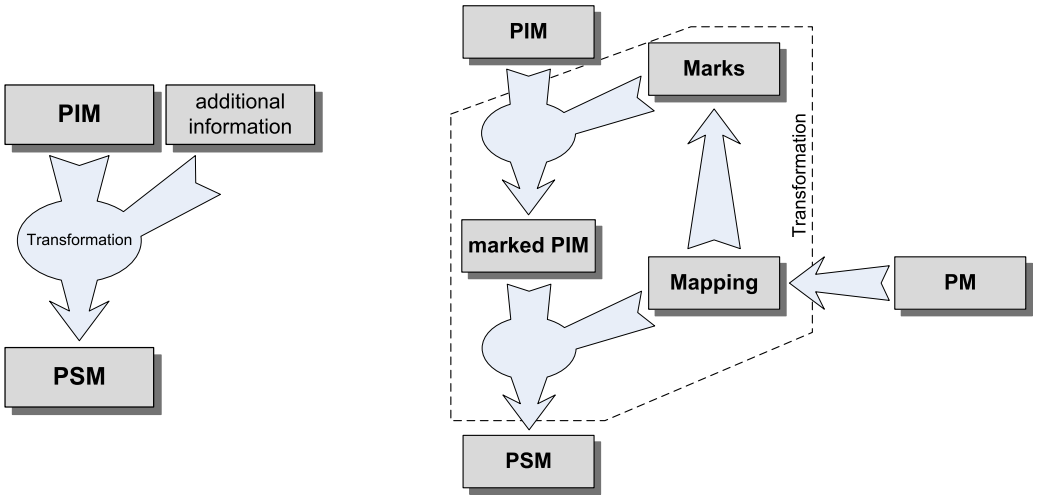
operating system and the hardware, this CORBA-specific PSM may still serve as PIM, if this is still of interest to the system development. For this purpose, after choosing a particular operating system a transformation to a PSM, e.g., a Linux-specific PSM, can be performed.

PIM to PSM Transformation

One of the key activities in the MDA is the transformation of the platform independent representation of the system (contained in the PIM) into a representation that is specific w.r.t. a particular platform—called the PSM. Figure 2.11(a) illustrates the generic transformation pattern for this purpose as it is defined in the MDA [OMG, 2003, p. 2–7]. The PIM together with additional information, e.g., the utilization of the services of a particular platform by the application, serves as the input to a transformation process, which results in the PSM of the system. A typical instantiation of this generic pattern is to realize the transformation process by the use of a *marked PIM* as depicted in Figure 2.11(b).

After a PIM of the system is built, a platform has to be chosen that enables the realization of the system with the desired functional and non-functional properties (e.g., dependability). For each platform a *mapping* has to be provided. The mapping specifies the rules for the transformation of a PIM into the PSM for a dedicated platform. For this purpose, for each element in the PIM the rule to be applied has to be identified, i.e., in which particular way this part of the PIM is transformed to the PSM. One solution therefore is the use of *marks* and the creation of a marked PIM. A mark represents a concept in the PSM and is always specific for a particular platform. It indicates how the marked model element of the PIM has to be transformed to the PSM.

The final step in the MDA pattern is to perform a model transformation of the marked PIM into a PSM. The MDA defines a model transformation as *the process of converting one model to another model of the same system* [OMG, 2003, p. 2–7]. This model transformation can be performed manually, with computer assistance, or automatically. The results of the transformation are the PSM and a record of the



(a) Generic PIM to PSM transformation

(b) Extended PIM to Marked PIM and Marked PIM to PSM transformation

Figure 2.11: MDA model transformation approaches (according to [OMG, 2003])

transformation. This record enables the traceability of the transformation by comprising information that describes which elements of the PIM correspond to elements of the PSM and which parts of the mapping were involved in the transformation. By the use of the transformation record, changes to PIM or PSM could be synchronized without the need to perform the entire transformation again.

2.3.2 Model-Driven Development in DECOS

The development methodology of DECOS follows the separation of application development from the development of the hardware platform, as it is characteristic for model-driven development approaches like the MDA, by distinguishing *platform-independent* and *platform-specific* viewpoints of the system. Figure 2.12 depicts, on a high abstraction level, the development methodology in DECOS. It is structured into three phases, namely system design, code generation, and deployment. System design itself is decomposed into requirement analysis, subsystem design, and the system integration [Giusto et al., 2002].

As depicted in Figure 2.12, the DECOS development methodology starts with a requirements analysis that is followed by the modeling of the platform independent viewpoint of the system—the DECOS PIM. In the PIM, the overall application is structured into smaller logical elements denoted as DASs and jobs, which exchange messages via virtual networks (see Figure 2.13(a)). For each virtual network, the PIM provides an operational specification that denotes the syntax of exchanged messages, the timing of the messages, and their error conditions (e.g., input and output assertions). Furthermore, the PIM contains a formal specification of the dependability and performance characteristics of the entire DAS. Within the scope of the DECOS

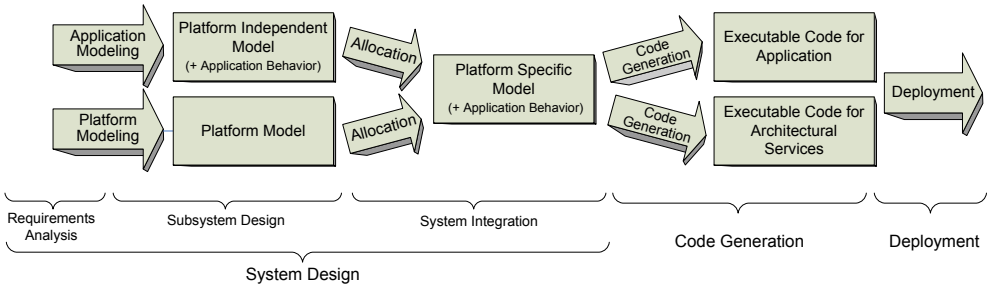


Figure 2.12: Development methodology in DECOS

project, a formal meta-model has been devised [DECOS, 2004] that enables the specification of the PIM by means of the *Unified Modeling Language* (UML) [OMG, 2007].

In addition to devising the logical structure, the behavior has to be formally captured. Although it is possible to directly capture the behavior of the application by using a specific programming language, it is recommended to elevate the level of abstraction of the behavior specification. This can be achieved for instance by using formalisms provided by Matlab-Simulink [Hanselman and Littlefield, 2001] or SCADE [Esterel Technologies, 2005].

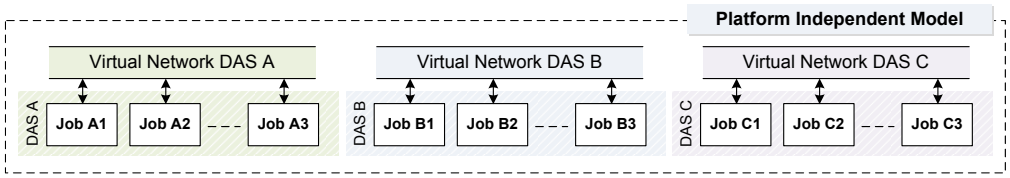
In parallel to the creation of the PIM, a *Platform Model* (PM) is constructed that captures the resources of a DECOS execution platform by describing its physical building blocks. According to the formal meta-model as specified in [Huber et al., 2006], a DECOS execution platform constitutes clusters, node computers, partitions, and (physical) networks.

Cluster: In DECOS, a cluster is a distributed computer system that consists of a set of node computers interconnected by a network.

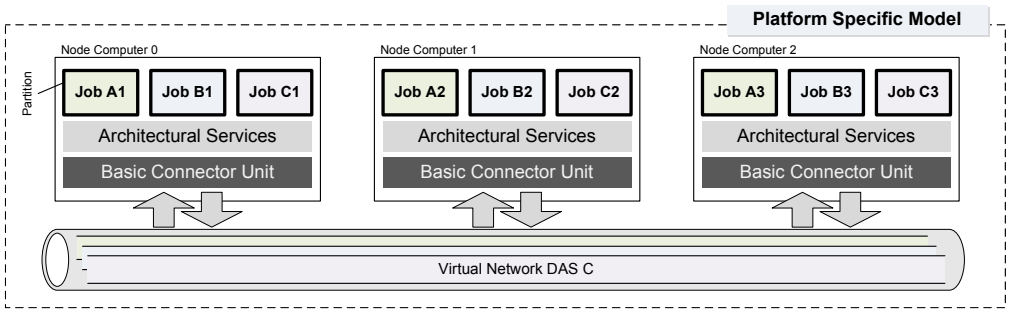
Node Computer: A node computer is a self-contained computational element with its own hardware (processor, memory, communication interface, and interface to the controlled object) and software (application programs, operating system), which interacts with its environment by exchanging messages. A node contains one or more partitions.

Partition: A partition is an encapsulated execution space within a node computer with a priori assigned computational resources (e.g., processor, memory, I/O) and network resources (e.g., network bandwidth) that can host a job. Partitions are the target of job allocation and each job is always assigned entirely onto a single partition, i.e., a job is never fragmented onto multiple partitions.

Physical Network: At cluster comprises at least the time-triggered core network, which establishes the interconnection between node computers. In case a node computer provides separate processors for establishing the safety-critical and non safety-critical subsystems (cf. Section 2.1.4), additional physical networks for intra-node communication are captured in the PM.



(a) Exemplary PIM of a DECOS system



(b) Exemplary PSM of a DECOS system

Figure 2.13: System representation using Platform Independent Models and Platform Specific Models

After PIM and PM are formally defined, the next step is a tool-supported transformation towards the PSM. The PSM is a refinement of the PIM (see Figure 2.13(b)), which extends the PIM with information concerning the mapping of jobs to appropriate node computers and hardware partitions, the configuration of the communication system, and the parametrization of the DECOS architectural services (cf. Section 2.1.4). The transformation of the PIM to the PSM is constrained by the following requirements:

- **Dependability requirements.** Replicated jobs (e.g., replicas of a TMR configuration) must be assigned to partitions of independent fault-containment regions.
- **Resource constraints of nodes.** The required computational resources of jobs allocated to a node computer must not exceed the available resources of the partitions located in this node computer.
- **Resource constraints of the physical network.** The resources of the physical network constrain the number of virtual networks that are supported simultaneously and the temporal performance of these virtual networks.

Finally, the generated source code—by using appropriate formalisms for the specification of the application behavior like Matlab-Simulink or SCADE, source code can be automatically generated by tools—together with the PSM forms the input to a deployment step in which the final executables for a specific instance of the DECOS execution platform are created.

The DECOS Tool-Chain

Within the course of the DECOS project, an integrated, model-driven tool-chain has been devised, which accompanies the system development process from design to deployment. Relying on model-driven concepts, the tool-chain targets at minimizing the risks for human coding mistakes by exploiting domain-oriented modeling, model transformation, and code generation [Herzner et al., 2007]. The tool-chain (depicted in Figure 2.14) comprises tools for the entire development methodology in DECOS as defined in the previous section, that is tools for system design, code generation, and deployment.

System Design

The central concept in the DECOS tool-chain for supporting the system design phase is the *VIATRA DECOS model store* (see Figure 2.14). The model store serves as the backbone for the integration of the individual tools involved in the system design. The tool integration is facilitated by the specification of data interchange formats based on *Extensible Markup Language* (XML) for importing and exporting data to the model store.

The first step in the system design phase is the generation of the PIM, which is supported in the tool-chain by two solutions: First, commercial UML tools like *Rational Rose 2003* and the *Rational Software Modeler* can be used for creating UML representations of the PIM, which have to be checked for its compliance to meta-model defined in [DECOS, 2004]. Furthermore, this representation has to be transformed into the XML-based representation, which allows the import into the DECOS model store. The second solution is the use of a *Domain-Specific Editor* (DSE). The benefit of a domain-specific editor is that it only provides model entities, which are defined in the underlying meta-model. Thus, compliance checks can be omitted. Such a DSE has been implemented under Eclipse technology that directly runs inside the *Visual Automated Transformations* (VIATRA) framework [Csertán et al., 2002], which also implements the model store, thus rendering the import to the model store becomes unnecessary.

For modeling the application behavior, SCADE [Esterel Technologies, 2005] is chosen as the primary tool in the DECOS tool-chain. SCADE is based on the formally defined data flow notation called *Lustre* [Halbwachs et al., 1991] and offers strong typing, explicit time management (delays, clocks, etc), and simple expression of concurrency [Herzner et al., 2007]. SCADE offers support for modeling and simulation of the application behavior, as well as, formal proof techniques and the generation of qualified code using a code generator, which has been certified against DO178B level A [RTCA, 1992].

Modeling of the hardware platform is simplified by a graphical DSE based on the *Generic Modeling Environment* (GME). GME is a configurable framework for the creation of domain-specific modeling environments [Ledeczi et al., 2001]. The major objective of the DSE is to facilitate hierarchical composition of the PM and

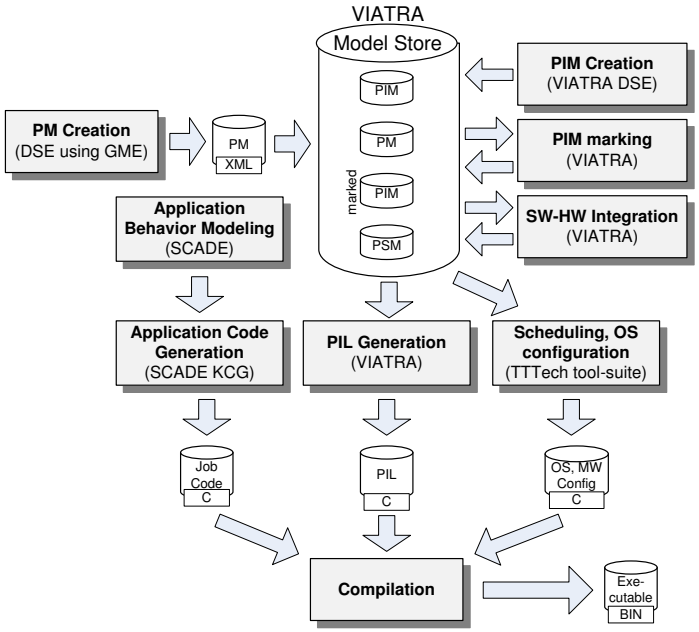


Figure 2.14: DECOS Tool-Chain – tool integration

to reuse existing PMs or parts of existing PMs to speed up the modeling process [Huber and Obermaisser, 2007]. The compliance of the PM to its meta-model is automatically checked during the modeling, ensuring that only valid PMs are imported in the model store. For performing the import, the DSE provides an automated transformation of the PM into an XML data format that is accepted by the model store.

The concluding activity in the system design phase is the *software-hardware integration* yielding in the PSM of the system. This activity is the integration of the individual application functions described in the PIM on the execution platform described by the PM. The software-hardware integration process comprises three major steps [Islam et al., 2006]: *(i)* the generation of a marked PIM which includes extended information on jobs and their requirements (e.g., for facilitating legacy integration, a job can be marked as *CAN job* in order to indicate that this job depends on an application middleware emulating a CAN environment [El Salloum et al., 2006]), *(ii)* the execution of feasibility checks eliminating unachievable requirements in an early design phase, and *(iii)* the allocation of jobs to dedicated partitions of the hardware platform. This allocation step has to consider the functional and non-functional constraints, which are given in the (marked) PIM. Examples for these constraints are resource constraints (e.g., available memory and computational performance, as well as, the availability of special purpose hardware like sensors and actuators) and dependability constraints (e.g., according to the fault-hypothesis of the DECOS architecture [Kopetz et al., 2004], replicated jobs forming a TMR configuration have to be allocated to separate node computers).

Code Generation and Deployment
Using SCADE for modeling the application functionality, the qualified code generator KCG from the SCADE tool-suite can be deployed for generating the code for the individual DECOS jobs. The typical operation of an application described with SCADE is to fetch all input variables, execute the application whereas it is assumed that the input parameters do not change during the execution time, and finally write the output variables. For being compliant with this execution semantic, the generated application code is compiled together with a dedicated wrapper code that performs the input and output operations. Both together form the code of a DECOS job.

In addition to generating application code, the configuration code for the time-triggered communication service and the partitions of the DECOS *Encapsulated Execution Environment* (EEE) [Leiner et al., 2007] have to created. For this purpose, the TTTech tool suite[1] (e.g., TTP-Plan, TTP-Build) has been integrated into the DECOS tool-chain. Besides the generation of a valid message schedule, an execution schedule of the partitions established by the EEE is created. The partition schedule determines the instant of execution of the individual jobs assigned to a dedicated partition.

Furthermore, to realize the *Platform Interface Layer* (PIL), which provides to the jobs a technology invariant interface to the DECOS architectural services, a dedicated API in the programming language C is generated for each job. This API fits exactly to the needs of the individual jobs, i.e., only access to those services (e.g., time-triggered or event-triggered ports for message transmission, global time service) is provided that are actually required by the job.

The final activity supported by the tool-chain is the deployment of the software on the target platform. Currently, the TriCore TC1796 manufactured by Infineon [Infineon, 2005] is supported by the tool-chain for the DECOS hardware platform. The deployment is performed by loading a single executable into the flash memory of the TC1796-based DECOS node computer. The executable contains the application software, middleware modules realizing the DECOS high-level architectural services (e.g., virtual networks and the EEE), as well as, configuration data of the node computer.

[1] `http://www.tttech.com/products/software.htm`

Chapter 3

The Time-Triggered SoC Architecture

We describe in this thesis the resource management solution for a novel integrated system architecture—the TTSoC architecture [Kopetz, 2005]—that aims at enabling the implementation of mixed-criticality embedded systems. This chapter outlines the concepts of this architecture and provides the conceptual background for the description of the resource management solution in the subsequent chapters. This chapter is structured as follows: The next section outlines the motivation and aims for devising the TTSoC architecture. Section 3.2 introduces the physical building blocks of the architecture by describing the component model of a single SoC implemented according to the TTSoC architecture, while Section 3.3 addresses the design of applications by summarizing a model-based design as devised in [El Salloum, 2007].

3.1 Motivation and Aims

In the recent years, there are mainly two large trends in the field of embedded systems: Firstly, aided by the enormous improvements of the functionality of semiconductor devices for embedded computing systems (e.g., the advent of chips with a number of transistors that is close to a billion), which is at the same time accompanied with a cost reduction of those devices, the complexity of embedded systems has highly increased. According to the 2005 semiconductor industry roadmap [SIA, 2005], the management of this increased complexity for system design is one of the key challenges for future embedded systems.

Secondly, the field of *Multi-Processor System-on-a-Chips* (MPSoCs) has become a highly active research area for embedded systems. While early MPSoCs where mainly targeted for super-computing applications and not for embedded applications, the decreasing size of MPSoCs, as well as, the reduction of cost and power consumption have rendered them also interesting for embedded applications [Wolf, 2004]. Further, the inherent concurrency in many typical embedded applications (e.g., automotive

electronics or avionics) in conjunction with the deployment of MPSoCs could be used to circumvent Pollack's rule, which states that the increase in performance of a sequential computer is only about the square root of the increase in the number of devices [Gelsinger, 2001]: By partitioning an application into a set of nearly autonomous concurrent functions, each of them assigned to a dedicated processing element, a nearly linear performance improvement could be achieved.

With the TTSoC architecture presented in this chapter, we address these challenges by offering a component-based design methodology for managing the complexity of billion-of-transistors SoCs through the consequent decoupling of the computational components from the communication infrastructure. It further supports *composability* [Kopetz and Obermaisser, 2002, Sifakis, 2005], i.e., the side-effect-free composition of component services to form larger systems-of-systems. As stated in [TT-SoC, 2007b], the contributions and key properties of the TTSoC architecture are as follows:

Elevation of the level of design abstractions. In order to tackle the increasing complexity of future embedded systems, we have to lift the design process to a higher level of abstraction. According to [TT-SoC, 2007b], this is proposed by conceptualizing components that form stable intermediate forms and exhibit aggregate properties, which are specified solely by an adequate interface model. Thereby, it is possible to consider the individual components as *black boxes*, i.e., to reason about interactions among components and the emerging system properties without the need to understand the internals of the component. It further enables the enhancement of the component's implementation without the need for redesigning the entire system at the higher abstraction level.
In the TTSoC architecture those stable intermediate forms are realized by the concept of *micro components*: micro components clearly separate the processing within the micro component from the interaction among the micro components and provide their functionality at a well-defined message-based interface [Gaudel et al., 2002].

Predictability and determinism through encapsulation. A key mechanism for reducing the cognitive complexity of a design is identified by the authors of [Feltovich et al., 2001] as the avoidance of system mechanisms that increase the cognitive load for understanding the system. One such system mechanism for complexity reduction is encapsulation of subsystems, because the behavior of interfering subsystems is more difficult to understand than the behavior of cleanly encapsulated subsystems. In addition, the effort required for test and validation of an encapsulated system is smaller than that for interfering systems [Owens, 2004].
In order to achieve encapsulation, the TTSoC architecture comprises a predictable on-chip interconnect that is based on a time-triggered communication schedule. Each subsystem implemented on the TTSoC architecture has assigned a number of dedicated, conflict-free sending slots, which are protected by hardware mechanisms. This way, non-interference between the subsystems is ensured and composability at the system level is facilitated.

Global time base. The major benefit of the existence of a system-wide global time base is the possibility to temporally coordinate different actions (e.g., computations, I/O interactions, etc) within a single SoC or even across an ensemble of interconnected SoCs. Consequently, timestamps of events in different subsystems (either within a single SoC or across SoC boundaries) can be related to each other. For instance, this feature is utilized for performing diagnostic actions on the health state of the SoC by correlating diagnostic information (e.g., failure indication messages disseminated from the architectural elements of the SoC) in the time domain.
The TTSoC architecture provides such a system-wide global time base through internal clock synchronization (i.e., within the SoC) and external clock synchronization w.r.t. an SoC-external reference time. The internal clock synchronization on a single chip is necessary, since it cannot be assumed that a single clock signal can be provided for the entire chip. The reason for establishing multiple clock domains include the handling of clock skew, the reduction of the operation frequency of individual IP blocks as part of power management, or the support for heterogeneous IP blocks with different speeds.

Integrated resource management. We understand integrated resource management as the simultaneous management of multiple resources in order to globally optimize the resource utilization. For this purpose, the design of the TTSoC architecture supports integrated resource management taking into account the following requirements: communication resources (e.g., bandwidth, latency, latency jitter), computational resources (e.g., dynamic allocation of computational resources to application subsystems), and power.
The integrated resource management builds an important cornerstone for the realization of power-aware systems [Unsal and Koren, 2003]. Furthermore, in case a permanent fault affects only parts of the SoC (e.g., individual micro components), dynamic reconfiguration is able to relocate the application functionality to free resources (e.g., spare micro components) in order to preserve the specified service of the SoC.

3.2 Architectural Elements

The architectural elements which form the structure of a single SoC that is built according to the TTSoC architecture are depicted in Figure 3.1. The central element of the TTSoC architecture is a *time-triggered Network-on-a-Chip* (NoC), which interconnects multiple, possibly heterogeneous *Intellectual Property* (IP) blocks called *micro components*. Any interactions between micro components occur solely by the exchange of messages over the time-triggered NoC. These messages are specified in both, the time and the value domain, by the interface specification of the micro components.

A micro component is subdivided into two parts, the *Trusted Interface Subsystem* (TISS) and the *host*. While the TISS belongs to the *trusted subsystem* of the SoC (the

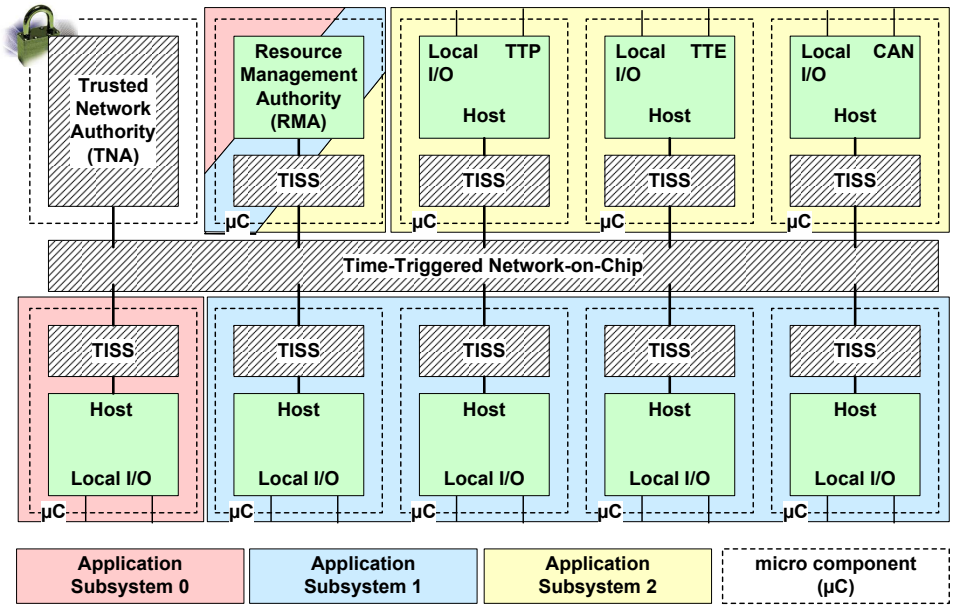


Figure 3.1: Architectural elements of the TTSoC architecture: *trusted subsystem (shaded) and non-trusted subsystem (hosts of micro components)* (figure taken from [Obermaisser et al., 2007])

constituting elements of the trusted subsystem are depicted as shaded components in Figure 3.1), the host is part of the *non-trusted subsystem*. The trusted subsystem is assumed to be free of design faults and ensures that a fault (e.g., a software fault or a design fault) within the non-trusted subsystem (e.g., a host of a micro component) cannot lead to a violation of the micro component's temporal interface specification and a subsequent disruption of the communication between other micro components.

The SoC architecture provides two dedicated architectural elements to support integrated resource management, namely the *Trusted Network Authority* (TNA) and the *Resource Management Authority* (RMA). These two architectural elements accept resource allocation requests from application subsystems and, if required, change the configuration of the SoC, e.g., by dynamically updating the communication schedule of the time-triggered NoC.

3.2.1 Micro Components

One objective of the TTSoC architecture is to provide a platform for the implementation of integrated (distributed) systems hosting multiple application subsystems (possibly possessing different criticality levels). The computational resources for hosting those application subsystems are provided by nearly autonomous and possibly heterogeneous *micro components*. A micro component is a self-contained computing element that is exclusively used by a particular application subsystem. An application subsystem can be implemented on a single micro component or by using a group of possibly heterogeneous micro components (either on one or on multiple,

interconnected SoCs).

A key mechanism of the TTSoC architecture for facilitating the implementation of mixed criticality systems is the encapsulation of micro components, which ensures that a failure of micro components of a non safety-critical application subsystem must not cause the failure of application subsystems of higher criticality. Encapsulation prevents by design temporal interference (e.g., delaying messages or computations in another micro component) and spatial interference (e.g., overwriting a message produced by another micro component).

For this purpose, a micro component comprises two parts: a *host* and a *Trusted Interface Subsystem* (TISS). The host implements the application services using a stable set of *core platform services* provided by the TISS, while the TISS protects the access of the host to the time-triggered NoC. This way, the behavior of a host can neither disrupt the computations nor the communication performed by other hosts. The only possibility that a faulty host can affect other hosts, is by providing faulty input values via the exchanged messages.

Trusted Interface Subsystem

To protect the access of the host to the NoC, each TISS manages a table (denoted as *Message Descriptor List* (MEDL)) that stores the a priori known points in time of all message receptions and transmissions of the respective micro component. Hardware protection mechanisms ensure that this table cannot be modified by the host. Thus, a fault within the host of a micro component cannot affect the exchange of messages of other micro components. This ensures spatial and temporal partitioning [Rushby, 1999] at the level of the NoC.

The design and the structuring of a micro component including the definition of the core platform services provided by the TISS, as well as, its interfaces has been part of the work of Christian El Salloum in the course of his PhD thesis [El Salloum, 2007]. According to this work, the TISS provides a stable interface with a defined set of generic services to the host for the development of application services. This interface is denoted *Uniform Network Interface* (UNI). The provided generic services, denoted as core platform services, are discussed in the following.

Communication service. The communication service establishes encapsulated communication channels for the message exchange between micro components. An encapsulated communication channel provides an unidirectional interface that transports messages at a priori defined points in time from a single source micro component to one or more destination micro components. In case of multi-cast communication, the communication service ensures that a consistent message ordering among the different receivers of the message is preserved. The host accesses this communication service via ports. A port is a communication endpoint in the UNI from which a message can be read or to which a message can be written. The TISS supports two basic types of ports: *State ports* used for the periodic transmission of messages

with state semantics and *event ports*, which are used for the sporadic transmission of messages with event semantics.

Global time service. The global time service permits to establish a temporal relationship between events that have been timestamped by different micro components within the SoC component, despite the existence of possibly multiple clock domains on a single chip. If multiple SoCs are interconnected over a gateway (see Section 3.2.3) via an appropriate external network that is synchronized with the global time service of individual SoCs (e.g., TTE [Kopetz et al., 2005]), the timestamps remain meaningful even across SoC components.

Programmable timer interrupt service. The TISS implements a programmable timer that provides two kinds of timer interrupts to the host: A one-shot timer interrupt that is configured to occur once at a specified instant of the global time base and a periodic timer interrupt. The recurring trigger instant of the periodic timer interrupt is configured by specifying period and phase offset using the same time format as for the specification of dissemination instants of messages over the NoC (see Section 3.2.2).

Watchdog service. The TISS exploits the watchdog service for determining whether a host is still operable or not, e.g., in case the host exhibits a fail-silent failure. For this purpose, the host is required to periodically update a dedicated memory location within the TISS. The update period of this life sign event, as well as, the activation state of the watchdog is not configurable by the host, in order to prevent that a faulty host is able to deactivate or impair the watchdog service. The resulting information can be either forwarded to the *Diagnostic Unit* (DU) (see Section 3.2.4) or can be directly exploited by the TISS to reset the host if a host failure is detected.

Power control service. The time-triggered NoC of the TTSoC architecture is a resources that is shared between all micro components. If a host is able to monopolize the access to the NoC, it would directly impair the services provided by all other hosts. This scenario is prevented by design by the introduction of a TISS for each micro component. Similarly, power is also such a shared resource. A faulty micro component that consumes more power than permitted can directly influence the correct operation of the entire chip. Therefore, the TISS has physical control over the power lines and the clock lines of the host in order to be able to turn off or deactivate the host in case it violates its specification.

Diagnostic dissemination service. In the TTSoC architecture, diagnosis is seen as an integral part of the architecture rather than an add-on that is implemented later on. For this purpose, the TISS implements a diagnostic dissemination service,

which is used to send failure indication messages (e.g., a missing life sign from a host, a queue overflow of an event port) to the DU (see Section 3.2.4).

Host

A host provides the computational resources for the execution of the application software. The concrete realization of the host is not determined by the design of the TTSoC architecture. A host can be realized as a general purpose processor, including system and application software, as well as, a custom hardware unit. The only way for the host to access the services of the architecture is via the UNI established by the TISS.

The host itself is horizontally structured into two layers: *frontend* and *application computer*. The main purpose of the frontend is to refine the core platform services. The core platform services are provided by the TISS at the UNI and are independent of any particular *DAS*. They separate the application functionality from the underlying platform technology to reduce design complexity and to enable design reuse. The refinement of the core platform services enables the implementation of higher-level services (e.g., voting in a TMR configuration), which can be reused in different applications. This refinement is achieved by incorporating middleware modules within the frontend. In addition, the frontend is used to provide a *temporal firewall interface* [Kopetz, 1997] for the host by implementing the micro components' *Communication Network Interface* (CNI). The application computer provides the computational resources of a micro component and controls the micro component's local I/O interfaces. An elaboration of the benefits obtained by this separation of frontend and application computer accompanied with a detailed interface specification of the UNI can be found in [El Salloum, 2007].

3.2.2 Time-Triggered Network-on-Chip

The time-triggered NoC interconnects the micro components of an SoC; thus, it is the central architectural element in the time-triggered SoC architecture. The NoC performs clock synchronization in order to establish a global time base on the SoC, as well as, to ensures the predictable transport of periodic and sporadic messages.

Clock Synchronization

The time-triggered NoC performs clock synchronization to provide a global time base to all micro components despite the existence of multiple clock domains. The resulting system-wide global time base allows the temporal coordination of actions on the distributed micro components within an SoC and in an ensemble of different SoCs. It is based on a 64 bit wide binary time format, which has been standardized by the OMG in the smart transducer interface standard [OMG, 2002].

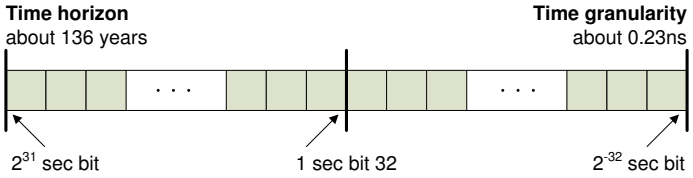


Figure 3.2: Time format of the Time-Triggered NoC

A digital time format is typically characterized by three parameters: *granularity*, *horizon* and *epoch*. The granularity determines the minimum interval between two adjacent ticks of a clock, i.e., the smallest interval that can be measured with this time format. The reasonable granularity can be derived from the achieved precision of the clock synchronization [Kopetz and Ochsenreiter, 1987]. The horizon determines the instant when the time will wrap around. The epoch determines the earliest representable instant.

The time format of the NoC is a binary time-format that is based on the physical second (see Figure 3.2). Fractions of a second are represented as 32 negative powers of two resulting in a granularity of about 230 picoseconds. Full seconds are presented in 32 positive powers of two, which yields a horizon of about 136 years. This time format is closely related to the time-format of the *Global Positioning System* (GPS) time and takes the epoch from GPS. The representation as negative and positive powers of two of the full second eases the synchronization with the full second signal of GPS. In case no external synchronization is available, the epoch starts with the power-up instant of the SoC.

Predictable Transport of Messages

Using TDMA, the available bandwidth of the NoC is divided into periodic conflict-free sending slots. We distinguish between two utilizations of a periodic time-triggered sending slot by a micro component. A sending slot can be used for the *periodic transmission of messages* or the *sporadic transmission of messages*. In the latter case, a message is only sent if the sender has to transmit a new event to the receiver. A message exhibits predefined temporal properties, which are specified by the period of the message, a start offset from the beginning of the period, and the message duration, i.e., the time it takes to transmit the entire message over the channel. The resulting exact knowledge of the transmission instant at the sender and the reception instant at the receiver permits the *temporal alignment* of sender and receiver jobs.

The temporal alignment of sender and receiver is required in applications demanding a short latency between sender and receiver. This is typical for many real-time systems. For example, consider a control loop realized by three micro components performing sensor data acquisition (A), processing of the control algorithm (C), and actuator operating (E) as it is schematically depicted in Figure 3.3. In this application, temporal alignment between sensor data transmission (B) and the start of

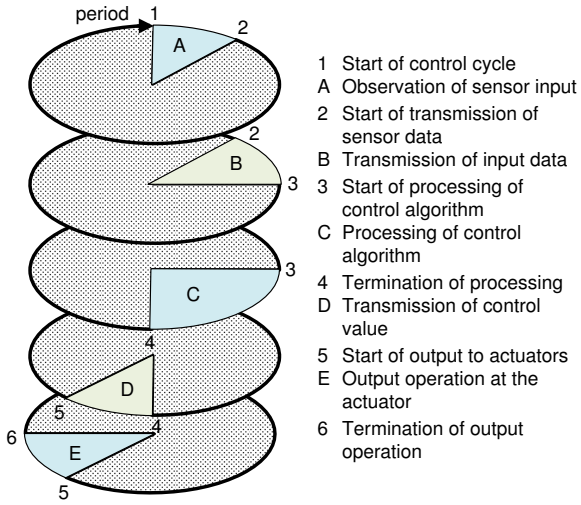


Figure 3.3: Temporal alignment in control loops. *In this cyclic model of time, the perimeter represents the period of the control application.*

the processing of the control algorithm (cf. instant 3 in Figure 3.3), as well as, the alignment between the transmission of the control value (D) and the start of actuator output (cf. instant 5) is vital to reduce the end-to-end latency of the control loop—an important quality characteristic of many real-time systems. By specifying two pulsed data streams corresponding to (B) and (D) in Figure 3.3, efficient temporally aligned data transmission can be achieved.

Contrary to the on-chip interconnect of the TTSoC architecture, many existing NoCs, e.g., the Sonics SiliconBackplane uNetwork [Sonics, 2002] or Æthereal [Goossens et al., 2005], provide only a guaranteed bandwidth to the individual senders without support for temporal alignment. The resulting consequences are: (i) the short latency cannot be guaranteed, (ii) a high bandwidth has to be granted to the sender throughout the entire period of the control cycle, although it is only required for a short interval, or (iii) the communication system has to be periodically reconfigured in order to free and re-allocate the non-used communication resources.

The allocation of the periodic sending slots of the time-triggered NoC to the micro components occurs by the use of a communication primitive called *pulsed data stream* [Kopetz, 2006]. A pulsed data stream is a *time-triggered periodic unidirectional data stream* that transports data in pulses with a defined length from *one* sender to n a priori identified receivers at a specified phase of every cycle of a periodic control system.

A pulsed data stream is specified by three parameters: *pulse period*, *pulse phase*, and *pulse duration*. It is made up of periodic pulses of data, which recur with the defined pulse period at the specified offset to the start instant of the pulse's period, defined by pulse phase. Our design restricts the pulse periods to 32 different periods corresponding to negative powers of two of the second, i.e., a period can be 1 second, 1/2 second, 1/4 second, 1/8 second and so forth. This restriction is introduced in

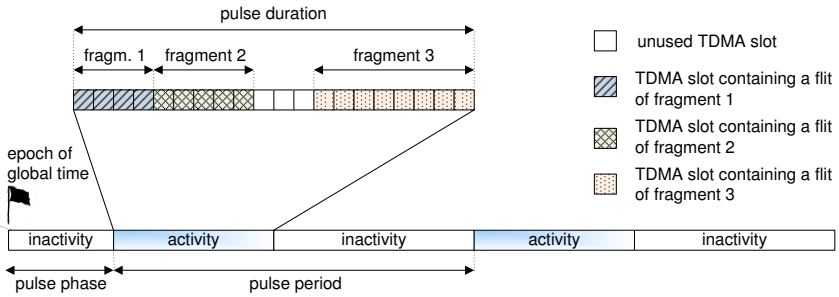


Figure 3.4: Representation of a Pulsed Data Stream

order to reduce the complexity of the NoC and the computation of the time-triggered schedule significantly.

A pulse of a pulsed data stream consists of at least one *fragment*. Each fragment is further decomposed into a set of atomic, fix-sized *flits*. A flit is the smallest unit of data that is transmitted over the time-triggered NoC and occupies one TDMA slot. Thus, the schedule of flits on the NoC determines the allocation of TDMA slots to the micro component that sends the pulsed data stream. Successive fragments of one pulse are not required to be transmitted in a dense sequence on the NoC, i.e., fragments of one pulse can be interleaved by fragments of other pulses. This way, pulsed data streams enable the efficient transmission of large data in applications requiring a temporal alignment of sender and receiver(s). The time between the instant of transmission of the first and the last fragment of a pulse is denoted the *duration* of the pulse (cf. Figure 3.4).

It is important to note that the decomposition of a message transmitted as pulsed data stream into fragments and flits is not visible to the host. The TISS abstracts from this fragmentation in order to simplify the complexity of the application design, as only the periodic/sporadic transmission/reception of entire messages are of relevance.

3.2.3 Gateways

The TTSoC architecture supports *gateways* for accessing chip-external networks, e.g., TTP [Kopetz and Grünsteidl, 1994], TTE [Kopetz et al., 2005], standard (802.3 compliant) Ethernet [IEEE, 2000], or CAN [Bosch, 1991] as depicted in Figure 3.1. The benefits of gateways include the interoperability with public networks, such as the Internet, and the ability to interconnect multiple SoCs to a distributed system. The realization of a distributed system enables applications for ultra-dependable systems based on the SoC architecture. In ultra-dependable systems, a maximum failure rate of 10^{-9} critical failures per hour is demanded [Suri et al., 1995]. Today's technologies do not support the manufacturing of chips with failure rates low enough to meet these reliability requirements. Since component failure rates are usually in the order of 10^{-5} to 10^{-6} (e.g., for ECUs this is statistically analyzed in [Pauli et al., 1998]), ultra-dependable applications require the system as a whole

to be more reliable than any one of its components. This can only be achieved by utilizing fault-tolerant strategies such as TMR [Lyons and Vanderkulk, 1962] that enable the continued operation of the system in the presence of component failures.

In case a time-triggered network (e.g., TTP or TTE) is used for the chip-external network, the TDMA scheme of the NoC can be synchronized with the TDMA scheme of the chip-external network. The periods and phases of the relayed pulsed data streams on the NoC can be aligned with the transmission start instants of the messages on the time-triggered chip-external network. Consequently, a message that is sent on the chip-external network is delivered to the micro components within a bounded delay and with minimum jitter (only depending on the granularity of the global time base).

The alignment between pulsed data streams and messages on time-triggered networks ensures that replicated SoCs perceive a message at the same time, i.e., within the same inactivity interval of the global sparse time base [Kopetz, 1992]. This property is significant for achieving replica determinism [Poledna, 1994] as required for active redundancy based on exact voting. Without synchronization between the NoC and the chip-external network, a scenario could occur in which one SoC forwards the message to the micro components in one period of the pulsed data stream, while another SoC would forward the message in the next period.

Furthermore, the gateways provide an SoC with an externally synchronized time base. For example, the global time base of the SoC can be synchronized to GPS time. Consequently, a timestamp assigned to an event is also meaningful outside the SoC. Furthermore, the global time base enables a global coordination of activities spanning multiple SoCs, e.g., output to actuators at the same global point in time.

3.2.4 Diagnostic Unit

The support for diagnosis constitutes an integral part of the TTSoC architecture. Therefore, all architectural elements are involved in failure detection. For instance, the watchdog service provided by the TISS is exploited to determine whether a host crashed. Additionally, the detection of queue overflows of event ports permits the detection of violations of message inter-arrival and service times [Kleinrock, 1975] in case of sporadic message transmissions. At the level of the NoC, data integrity during message transmission is protected and verified by a checksum at the end of each fragment. Furthermore, resource requests leading to a violation of the system specification (e.g., exceeding predefined resource quotas of particular application subsystems) are detected at the TNA and might indicate a failure within the requesting micro component.

All of these detection events result in failure detection or failure indication messages, which are disseminated to a dedicated core on the SoC for further analysis, namely the *Diagnostic Unit* (DU). Each of those messages is structured into three attributes describing the detection event. The *type* attribute provides information about the type of the observed failure (e.g., crash failure of a host, illegal resource

request). A *timestamp* is included in each message for recording the time of detection w.r.t. the system-wide global time and the *location* attribute identifies the architectural element that causes the observed failure.

It is the purpose of the DU to accept those failure indication messages and perform additional failure detection by means of message classification. Therefore, the DU executes assertions on the syntactic, temporal, and semantic correctness of messages according to the DSoS message classification [Gaudel et al., 2002]. The DU correlates different failure detection messages in space and time in order to identify the faulty architectural element. By the use of the timestamp that is part of each message, correlation in the time domain is made possible. The location attribute of those messages in combinations with the inherent fault-isolation of the TTSoC architecture permits the correlation of failure detection messages in the space domain. The identification of the faulty architectural element is a prerequisite for triggering an appropriate corrective action such as restarting a host that suffers from a failure caused by a transient fault or updating the software in the host to eliminate a design fault.

3.2.5 Architectural Elements for Resource Management

The support for developing mixed-criticality systems, has significant impact on the design of the resource management solution of the TTSoC architecture: On the one hand, the resource allocation should be flexible and nearly optimal w.r.t. resource utilization, in order to be competitive in the implementation of non safety-critical applications. On the other hand, predictability, determinism, and fault isolation are vital characteristics for resource management regarding safety-critical systems.

To this end, the architecture provides two distinct architectural elements for resource management, namely the *Trusted Network Authority* (TNA) and the *Resource Management Authority* (RMA). The RMA computes new resource allocations for the non safety-critical application systems, while the TNA verifies and actually executes the resource reallocation and ensures that the new resource allocations have no negative effect on the behavior of all hosted application systems (in particular the safety-critical application systems). In the following, a short introduction of those architectural elements is given. A detailed description of the resource management services provided by TNA and RMA is given in Chapter 5, which is dedicated to resource management in the TTSoC architecture.

Trusted Network Authority. The task of the TNA is to update the configuration of the SoC (e.g., the time-triggered schedule stored within each TISS) according to configuration proposals generated by the RMA. As depicted in Figure 3.1, in contrast to the RMA, the TNA is part of the trusted subsystem of the SoC. Thus, the TNA acts as a guard for computations performed by the RMA and is empowered to reject erroneous configuration proposals. A configuration proposal is classified as erroneous, if potential collisions in the communication schedule for the time-triggered NoC are

detected or violations of resource guarantees are found. In such a case, the current configuration remains unchanged and the RMA is notified about the reject.

Resource Management Authority. The RMA is responsible for scheduling the available resources to allocate them among the micro components according to *resource requests* received from the individual application subsystems. For this purpose, the RMA exploits application-specific knowledge (e.g., communication topology) and system knowledge, e.g., temporal properties of the time-triggered NoC. However, the RMA is not permitted to change the configuration of the SoC directly, i.e., to update the affected TISSs. This is ensured by the design of the TTSoC architecture. Therefore, it has to pass on the configuration proposal to the TNA where it is verified for its validity. The reason for this procedure is that the RMA is not part of the trusted subsystem. Thus, it is not required for an implementation of the RMA to pass a certification process for a level of criticality that is as high as the criticality of the trusted subsystem. This split-up of the calculation and the authorization of new resource schedules into two separate parts (and also two separate *Fault Containment Regions* (FCRs)) significantly simplifies the certification of the entire SoC, which is in particular important for safety-related or safety-critical applications.

3.3 Application Modeling

We have introduced in Section 2.1.4 a methodology to structure a large system into smaller parts that are manageable with less mental effort. This approach, which entails the subdivision of the entire system into a set of nearly independent *DASs*, which are further decomposed into *jobs*, forms the basis for modeling applications in the TTSoC architecture. The issues of application modeling and naming have been elaborated in [El Salloum, 2007]. This section summarizes only the concepts that are relevant for the further understanding of the following chapters.

In the TTSoC architecture, a DAS denotes a part of the entire system that provides a self-contained application service and is implemented on a single SoC or on a set of interacting SoCs. A job is a constituting part of a DAS that represents a basic unit of work within the DAS and is atomic w.r.t. distribution. This means that a job is always implemented on a single micro component. Any interaction among jobs occurs solely via the exchange of messages over *channels*. A channel defines the communication topology of the message exchange and is associated with temporal and dependability properties. An endpoint of a channel used by a job to access the channel is denoted *port*.

For modeling a system by means of DASs executed on the TTSoC architecture, we propose to use a model-based design methodology similar to the MDA, using platform-independent and platform-specific representations of the system. As explained in Section 2.3.1, the perception of a model as PIM or as PSM is not static, but changes dynamically during the modeling process depending on the current viewpoint on the model. In the model-driven design approach for the TTSoC architecture,

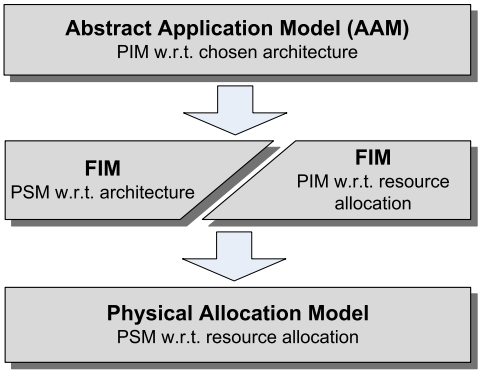


Figure 3.5: Models deployed in the design process of the TTSoC architecture

we have defined three models representing three distinct phases in the design process (see Figure 3.5), namely the *Abstract Application Model* (AAM), the *Fully-specified Interface Model* (FIM), and the *Physical Allocation Model* (PAM) [El Salloum, 2007].

The AAM is an abstract representation of the system, where the interfaces of the individual subsystems are not fully specified. This means that some design decisions are still left open to be solved in a subsequent step of the design process. For instance, the selection of an adequate security protocol in order to achieve the desired security properties of a communication channel. The reason is to provide a model that enables the conceptualization of the problem and which provides the means for describing its solution with a formalism at a higher-level of abstraction. Kopetz states that: *The major challenge of design is the building of a software/hardware artifact (an embedded computer system) that provides the specified services under given constraints and where relevant properties of this artifact can be modeled at different levels of abstraction by models of adequate simplicity* [Kopetz, 2007].

As depicted in Figure 3.5 we denote the AAM as PIM in our design process, since it is independent w.r.t. the chosen architecture. At this high-level of abstraction, we do not restrict the choice of the final architecture to the TTSoC architecture. Consequently, we denote the FIM as PSM in our design process, since it is specific to a particular architecture—the TTSoC architecture—and permits the modeling of the application service only by means of core platform services provided at the UNI. The transformation of the AAM to the FIM is typically a non-trivial task that cannot be done fully automatically. During this transformation plenty of design decisions have to be performed. For instance, the selection of appropriate means for achieving the dependability of a channel as specified in the AAM.

The FIM captures the entire structure of a distributed system. It describes the decomposition of the overall system into DASs, as well as, the behavior, the interaction, and the non-functional properties (e.g., dependability properties) of their constituting jobs and channels. As described in [El Salloum, 2007] we distinguish two kinds of representation of the FIM at two different levels of abstraction, namely the *Macro FIM* (MFIM) and the *Uniform FIM* (UFIM). While the MFIM enables the specification of DASs by means of higher-level modeling constructs (e.g., request/reply

communication channels), the UFIM restricts the specification of DASs to constructs that represent services, which are natively provided at the UNI (e.g., unidirectional communication channels).

In the following, we consider only the modeling constructs that are expressed by the UFIM and their counterparts in the PAM, since they form the application and system knowledge that is exploited by RMA and TNA to perform resource management. In the UFIM, the communication topology is expressed by means of UFIM-Channels and UFIM-Ports. A UFIM-Channel represents an encapsulated, unidirectional communication channel that transports messages with defined temporal properties (e.g., period, phase, and duration) from a single source job to one or more destination jobs. A UFIM-Port is an endpoint of a UFIM-Channel and is exploited by a job to access, i.e., read messages from or write messages to, a UFIM-Channel. A job can access multiple UFIM-Ports, since it can be attached to multiple UFIM-Channels.

According to [El Salloum, 2007], a UFIM-Port is uniquely identified in the entire system by its fully-qualified *UFIM-Port identifier*. This is a location independent identifier, which means that it describes the role of the port in the system, but abstracts from the physical component (e.g., a particular micro component on an SoC) that implements the port. The fully-qualified UFIM-Port identifier comprises the *DAS context* (e.g., an avionics system or an automotive system), a unique *DAS identifier* within the DAS context (e.g., a brake-by-wire system within a car), a unique *local job identifier* within the scope of the DAS (e.g., a job processing the sensor data of the left front wheel), and a unique *local UFIM-Port identifier* for a particular UFIM-Port of the job (e.g., a UFIM port, which holds state messages containing the actual revolutions per minute of the wheel).

The UFIM (as one kind of representation of the FIM) includes the full specification of the *Linking Interface* (LIF) of each job of a DAS, but abstracts from the mapping of DASs to the physical resources of the platform. Thus, w.r.t. the resource allocation it is also a platform-independent representation in our design process (cf. Figure 3.5 where the FIM is denoted as PIM w.r.t. resource allocation). This additional information is contained in the PAM of the system. The PAM is a platform-specific representation adding the mapping of jobs to a particular SoC and micro component to the FIM. In contrast to the definition of a job in the FIM, a job's definition in the PAM is specific to the characteristics of the micro component on which a job should be executed. Nevertheless, a jobs's LIF in the PAM is exactly the same as the jobs's LIF in the UFIM. Similar to the AAM-to-FIM transformation, the FIM-to-PAM transformation is a non-trivial engineering task.

The PAM describes the mapping of the platform-independent structure described by the FIM to the physical system structure of the distributed system. The physical system structure of a distributed system comprises one or more clusters. A cluster consists of a set of SoCs interconnected by a physical network. The interconnection of SoCs is implemented via dedicated components, denoted as gateways, and is established via *Gateway-Channels* (G-Channels). A G-Channel is a unidirectional

communication channel that is used for message transmission from one source SoC to at least one destination SoC. Analogous to UFIM-Channels, the endpoint of a G-Channel is denoted as *G-Port* and an SoC can be attached to multiple G-Channels; thus requiring multiple G-Ports. Each G-Port is uniquely identified in the physical system structure by a fully-qualified G-Port identifier that is composed of a unique *cluster identifier*, a unique *SoC component identifier* within the cluster, and a *local G-Port identifier* on a particular SoC.

Within a single SoC, the physical system structure describes the interaction of micro components. The communication between micro components is established by *SoC-Channels*. The SoC-Channels denotes an encapsulated unidirectional communication channel for message transport within a single SoC. Thus, an SoC-Channel transports messages from one source micro component to one or more destination micro components, but is not able to cross the SoC boundaries. *SoC-Ports*, which denote the endpoints of SoC-Channels are identified by a fully-qualified *SoC-Port identifier*. This identifier is formed by the composition of a *micro component identifier* that is unique within the particular SoC, as well as, a *local SoC-Port identifier* that is unique within the sender/receiver micro component.

Since SoC-Channels cannot cross the boundaries of a single SoC, the case of two interacting jobs which are allocated to micro components located on different SoCs has to be treated separately. Such an allocation requires an SoC-Channel from the micro component hosting the sender job to the gateway of the respective SoC. Additionally, a dedicated G-Channel to each SoC hosting a receiver job has to be established. Within the SoCs of the receiver jobs, an SoC-Channel from the gateway to the respective micro component of the receiver job is required.

It is the purpose of resource management w.r.t. communication resources, which is the focus of the following chapters, to take the specification of UFIM-Channels as input and to produce a mapping between those UFIM-Channels to SoC-Channels. In addition, it generates a configuration of the SoC-Channels in such a way that the temporal constraints of the UFIM-Channels (e.g., bandwidth, latency, or phase alignment) are fulfilled.

Chapter 4

Resource Allocation Policies

An architecture that supports on-line reconfiguration and an operating system or middleware layers that can be configured during runtime are prerequisites to enable resource management in a distributed system. In addition, in order to coordinate dynamic resource management, it is required to define a policy for allocating resources among the architectural elements of the system.

It is the purpose of this section to discuss several resource management and allocation policies. As a first step, we present a simplified, fictive example of an integrated system from the automotive domain, which is used to elaborate on the strengths and weaknesses of the different approaches. Subsequently, we give arguments for the choice of the resource management policy in the TTSoC architecture.

4.1 Exemplary Application Scenario

The purpose of this section is to set the stage for an analysis of different resource management policies. Therefore, we introduce a fictive example of an integrated system on which we will elaborate on the various strategies for (re)allocating resources to *Distributed Application Subsystems* (DASs).

Nowadays, a vehicle houses a multitude of different application subsystems (e.g., powertrain, comfort electronics, infotainment) interconnected by diverse communication networks (e.g., CAN [Bosch, 1991], *Media Orientated Systems Transport* (MOST) [MOST Cooperation, 2002], or LIN [LIN, 2003]). This trend leads to an enormous number of *Electronic Control Units* (ECUs) and cabling in a state-of-the-art vehicle. For instance, the number of micro controllers inside a vehicle has increased from 20 in the year 2000 to 40–60 in 2003 [Murray, 2003], the BMW 7 series cars even contain up to 70 ECUs [Deicke, 2002]. As a consequence, the reduction of the number of ECUs on-board is one major goal in system design for automotive system, which is addressed by integrated system architectures as AUTOSAR [Heinecke et al., 2004] or DECOS [Obermaisser et al., 2006].

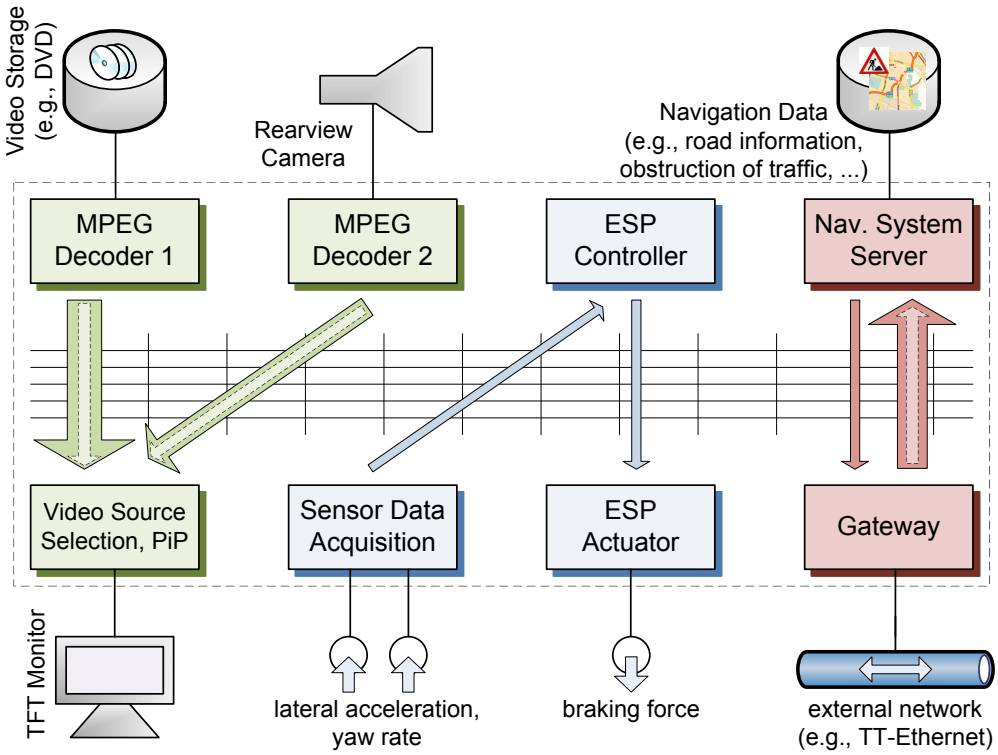


Figure 4.1: Simplified example of an Integrated System. *The dashed arrows indicate channels with variable bandwidth.*

A fictive integrated node computer, i.e., an instance of the time-triggered SoC architecture, of a future automotive system is schematically depicted in Figure 4.1. The SoC houses eight micro components dedicated to three diverse DASs. For sake of simplification, the architectural elements for resource management and diagnosis (cf. Section 3.1) are not shown in this schematic representation. In addition, in order to keep the example and the analysis feasible, only communication resources are considered in this chapter. Table 4.1 summarizes the communication resources assumed to be provided by the SoC-internal and the SoC-external networks.

Name	**Bandwidth**	**Comment**
Internal Network	1 Gbps	overall bandwidth of the internal network
External Network	100 Mbps	overall bandwidth of the external network

Table 4.1: Available communication resources in the exemplary application

4.1.1 Multimedia Application Subsystem

The first DAS shown in Figure 4.1 is a multimedia subsystem which streams a video stream from two different sources—either a stream from a video storage like a DVD player or from a rear view camera that acts as a parking assistant—to a single sink,

e.g., a TFT monitor connected to the *video source selection job*. This job is able to select the source(s) to be displayed. There are five possible combinations:

1. No source (e.g., if the video sink is switched off)
2. DVD in full size
3. Rear view camera in full size
4. *Picture-in-Picture* (PiP) mode; DVD in full size and rear view camera small
5. PiP mode; rear view camera in full size and DVD small

The purpose of the micro components connected to the video sources is to decode the video stream and transmit a raw data stream to the video sink. Depending on the selected display mode, individual streams are transmitted in full resolution, low resolution, or not at all. The assumptions for the requirements on communication bandwidth for different display modes are listed in Table 4.2.

Name	**Sender**	**Bandwidth***	**Comment**
DVD full	MPEG 1	∼944 Mbit/s	1280x1024 pixel @ 24 bit
DVD small	MPEG 1	∼55 Mbit/s	320x240 pixel @ 24 bit
Rear view cam full	MPEG 2	∼221 Mbit/s	640x480 pixel @ 24 bit
Rear view cam small	MPEG 2	∼55 Mbit/s	320x240 pixel @ 24 bit

* assuming a frame rate of 30 frames/sec

Table 4.2: Communication requirements of the exemplary multimedia application

4.1.2 ESP Application Subsystem

The second DAS represents a simplified *Electronic Stability Program* (ESP) braking system. It consists of one micro component responsible for the acquisition and preprocessing of sensor data, e.g., lateral acceleration and yaw rate of a vehicle, one micro component that executes the control job for calculating the braking force on the wheel, and a third micro component that operates the actuator (e.g., an electronically controlled brake).

The assumed communication requirements of the ESP DAS are represented in Table 4.3. Compared to the multimedia DAS, the communication requirements of this DAS are very low. The sensor micro component disseminates two 32 bit values which are sampled with a frequency of 100 Hz to the controller micro component. The control value for the actuator is again disseminated by a single 32 bit value. Thus, the overall bandwidth requirements of the ESP subsystem results in approximately 10 kbit/s.

Unlike the multimedia DAS, however, this subsystem realizes a part of the vehicle functionality that is related to the safety of the entire system. Therefore, the

Name	Sender	Bandwidth	Comment
Lateral acceleration	Sensor	3.2 kbit/s	32 bit value @ 100 Hz
Yaw rate	Sensor	3.2 kbit/s	32 bit value @ 100 Hz
Brake force	Controller	3.2 kbit/s	32 bit value @ 100 Hz

Table 4.3: Communication requirements of the exemplary ESP application

resource management system, i.e., TNA and RMA, have to ensure that under any circumstances the communication bandwidth granted to this application subsystem is not withdrawn.

4.1.3 Infotainment Application Subsystem

The last DAS that is hosted on the time-triggered SoC presented in Figure 4.1 is an infotainment application subsystem. The only part of this DAS located on this SoC is a navigation system server, which responds to route information requests. The origin of these requests is outside the boundaries of the time-triggered SoC; thus, the gateway is used to forward the requests to the respective micro component.

For the exchange of control messages to request route information, e.g., by a dedicated visualization and user interface system located on a separate SoC, a dedicated channel with a bandwidth of 10 kbit/s is reserved for this messages. Depending on the nature of the request, the response of the navigation system server contains the route information solely, enhanced information regarding points of interest, or additional graphical information that enables a 3D representation at the visualization system. We list the required communication bandwidth for the different detail levels of the route information in Table 4.4.

Apart from accepting route information requests, this dedicated channel is also exploited by the navigation system server to initiate the update of the stored map material, e.g., to refresh traffic jam information, at the navigation system storage. In our example the download rate of this data varies in the range of 100 kbit/s and 500 kbit/s (cf. Table 4.4).

4.2 Static Resource Allocation

In an integrated system using static resource allocation without on-line resource management, the allocation of the available resources (e.g., the communication bandwidth provided by the internal and external network in the example presented above) to the DASs is performed off-line, e.g., during the integration of the individual DASs into a single system. A static resource allocation strategy, however, has to assume the worst case demand for all DASs in its offline planning phase, even for DASs with dynamic resource usage like the multimedia application subsystem that is able to change the QoS of the streamed video data. The result is a highly inefficient utilization of the available communication resources.

Name	Sender	Bandwidth	Comment
Navi control	Navi/GW	10 kbit/s	initiating download of updated map material or the transfer of route information
Data Update	GW	100 kbit/s - 90 Mbit/s	download of updated map material/road information to the navigation system storage
Route information	Navi	100 kbit/s	streaming route information to the visualization system
Points of interest	Navi	100 kbit/s	enhancement of route information with points of interest
3D data	Navi	300 kbit/s	enabling 3D visualization of route information

Table 4.4: Communication requirements of the exemplary infotainment application

Considering the multimedia example mentioned before, the worst case demand of communication bandwidth results in ∼944 Mbit/s for streaming the DVD source in full quality, plus additional ∼221 Mbit/s for streaming the data of the rear view camera in full quality. Although in the specification of the system, these two streams do not occur simultaneously, bandwidth for both streams has to be reserved, since no online resource management is performed: the streams are only activated or deactivated according to the user request. Nevertheless, since no additional complexity for resource management is introduced to the system, this static approach is in widespread use for resource allocation in safety-critical systems, e.g., in avionics systems using a time-triggered communication system [Nilsson et al., 1998, Scheidler et al., 2000].

Role of the Resource Management System:
We do not need an RMS in case resource allocation is performed off-line, since all communication requirements of every DAS are taken into account during the planning phase and appropriate bandwidth is reserved. In the example described before, a static resource allocation would entail that the maximum bandwidth for all defined communication channels have to be reserved. This would lead to an overall demand for communication bandwidth of 1.256 Gbps, which could not be provided by the internal network.

Resource Management Procedure:
The DASs, respectively the jobs, are not empowered to emit requests for resources and all communication channels are statically configured.

4.3 Dynamic Resource Allocation

Compared to static resource allocation, a fully dynamic allocation strategy represent the other extreme. In such a system, each job competes for the available commu-

nication resources against all other jobs hosted on the same SoC. The allocation of resources is not calculated beforehand and therefore not static during runtime, but each job requests for the resources it actually requires. A minimalistic implementation of such a behavior would be the use of a *Carrier Sense Multiple Access / Collision Avoidance* (CSMA/CA) technique for arbitrating the access to the communication medium at the sending micro components, as done for instance in CAN [Bosch, 1991]. In the scope of the TTSoC architecture, this would require a RMS that accepts resource requests from all jobs of the individual DASs and that is empowered to alter the configuration of the time-triggered NoC according to an on-line calculated resource allocation.

Role of the Resource Management System:
Given the individual job resource requests, the RMS allocates the available resources in a fine-grained manner to the jobs of the individual DASs. In case resource conflicts occur, e.g., there is not sufficient bandwidth to fulfill the communication demands of all jobs, the RMS has to ensure a conflict-free resource allocation. For this purpose, the RMS has to contain a detailed application knowledge of the individual subsystem:

- *Priority/Utility for the overall system of the individual DAS:* In case of conflicts the RMS has to decide, whether a requested share of the resources is granted to a job or other jobs are preferred due to their higher priority and/or utility for the system
- *Granularity of resource allocation:* It is often not possible to continuously increase or decrease the resources allocated to a particular job. For instance, while the bandwidth of the *data update* communication channel of the infotainment application subsystem can be more or less continuously varied within the given bounds (cf. Table 4.4 in Section 4.1.3), this is not possible for the DVD video stream in the multimedia subsystem. The DVD video can either be streamed in full-resolution (1280x1024 pixel requiring a bandwidth of 944 Mbit/s) or with a resolution of 320x240 pixel in PiP mode (requiring a bandwidth of 55 Mbit/s). Thus, when full resolution is demanded by the user, the bandwidth allocated to this job cannot be reduced, e.g., to 100 Mbit/s.
- *Interdependencies of resource allocations:* Many applications rely on pairs of communication channels that are directly linked together, such as request/reply or client/server interactions patters. For such applications it is often not possible to reduce the bandwidth of one part of the pair without affecting the other one.

Resource Management Procedure:
Each job requires a dedicated communication channel to the RMS to transmit its resource requests. The RMS processes the incoming requests, either in a regular periodic manner or immediately after the reception of the request. By exploiting the application knowledge stored in the RMS, decisions are taken whether resource

requests are granted or not and whether these decisions entail interventions in the resource allocation of other DASs. For instance, to enable a PiP display of the DVD stream in full resolution and the rear view camera with small resolution, the overall bandwidth granted to the infotainment application subsystem has to be reduced to approximately 1 Mbit/s to ensure that sufficient resources are available on the internal network (note that the bandwidth of the ESP DAS must not be reduced). Afterwards, all affected application subsystems are informed of the conducted changes and the configuration of the communication system is updated.

4.3.1 Restricted Dynamic Resource Allocation

As stated in Section 2.1, integrated system architectures aim at combining the functional integration and hardware benefits of an integrated system with the error containment and complexity management advantages of the federated approach. For this, it is of utmost importance to facilitate the composability of application subsystems by the system architecture. Especially the second principle of composability, as defined in [Kopetz and Obermaisser, 2002], the *stability of prior service* is not supported by unrestricted dynamic resource allocation, since the effects on resource allocation for the integrated system cannot be analyzed separately for each application subsystem.

A prevention of these adverse effect on composability can be achieved by restricting the degrees of freedom of the RMS for performing resource allocation to strict boundaries for each DAS. This means, each DAS has assigned a statically allocated and guaranteed share of the overall available resources and the RMS is only empowered to grant requested resources as long as the allotted resource share of the respective DAS is not exceeded. This way, the effects of the resource allocation performed by the RMS can be analyzed for each subsystem in isolation, which facilitates composability.

Compared to the static resource allocation approach, a better utilization of the available resources can be achieved, since application knowledge can be exploited for dimensioning the resource share that is statically allocated to each DAS. For instance, instead of requiring a statically allocated bandwidth of 1.165 Gbps for the multimedia application subsystem, as it would be the case for a static resource allocation, the statically allocated share of the bandwidth for this DAS could be reduced to 999 Mbit/s (944 Mbit/s for the full-resolution of the DVD video stream and 55 Mbit/s for the stream of the rear view camera stream in small resolution). This is possible since the RMS is now able to dynamically (re)allocate the communication resources to the DVD decoder job and the decoder job of the rear view camera, respectively. However, compared to the unrestricted resource allocation approach, a less efficient resource utilization is achievable, since free resources cannot be exploited across DAS boundaries.

4.4 QoS-based Resource Allocation

Similar to unrestricted dynamic resource allocation, the *Quality of Service* (QoS)-based resource allocation approach shares the available resources across all DASs hosted on the SoC. In this approach, however, the RMS is supported in its decision on resource allocation, by having knowledge of a range of QoS levels for each DAS. For instance, for the infotainment application subsystem it can be specified that, depending on the available bandwidth, the *navigation system server* can operate with a bandwidth of 100 kbit/s, 1 Mbit/s, 50 Mbit/s, or 90 Mbit/s for its data download stream. In case a resource allocation conflict arises, this knowledge is exploited in order to decide, which DAS has to be operated at a lower QoS in order to free the required amount of resources. A representative of QoS-based resource allocation is introduced in [Agrawal et al., 2003]. The authors describe how they exploit QoS-based allocation of computational resources for advanced avionics systems.

Role of the Resource Management System:
The application knowledge required at the RMS is comparable to the unrestricted dynamic resource allocation technique. However, the explicit specification of granularity and interdependencies of resource allocations is rendered unnecessary, since this information is subsumed in the specification of the QoS levels of the individual DAS. One possibility to provide this information to the RMS is via resource requests themselves. For instance, a request from the navigation system server could contain the requirement for a download channel with a bandwidth of 90 Mbit/s, but including also additional degraded QoS levels (e.g., 50 Mbit/s, or 100 kbit/s) in case the actually available resources would be exceeded.
Another approach is to explicitly specify a finite set of operation modes for each application subsystem, each augmented with a specification of the resource requirements at different QoS levels. This way, an off-line analysis can be performed, to investigate whether the resources are sufficient to enable for each DAS the provision of its service (at least at a degraded QoS level). An example for such a specification for the infotainment application subsystem is presented in Table 4.5.

Resource Management Procedure:
In case a finite set of operation modes with its QoS levels is specified for each DAS, each job can directly request one of the specified modes. The RMS is responsible for summing up the resource demands required for operating each DAS in its requested mode. When a resource allocation conflict occurs, the RMS is empowered to change the QoS level of particular DASs, again according to the priority and utility of the respective subsystem with respect to the entire system. By changing the QoS level, a DAS is still able to provide the requested service, potentially of degraded quality. Considering the example presented in Table 4.5, when the operating mode *Update* is requested, the RMS is only empowered to change to a lower QoS level (from level 4 to 0), but not to the operating mode *Route*. If the DAS permits the deactivation of a service in case of resource allocation conflicts, this has to be explicitly specified

Mode	QoS	Bandwidth	Comment
Update	4	90 Mbit/s + 10 kbit/s	download of map or road information updates with maximum download rate
	3	50 Mbit/s + 10 kbit/s	download map or road information updates with reduced download rate
	2	1 Mbit/s + 10 kbit/s	— ” —
	1	100 kbit/s + 10 kbit/s	— ” —
	0	10 kbit/s	download deactivated
Route	3	510 kbit/s	route information (100 kbit/s), information on POIs (100 kbit/s), 3D visualization data (300 kbit/s), and control data (10 kbit/s)
	2	210 kbit/s	route information, information on POIs, and control data
	1	410 kbit/s	route information, 3D visualization data, and control data

Table 4.5: Exemplary operation modes and QoS levels for the infotainment DAS

(e.g., as it is the case for QoS level 0 in the example). After the resource allocation, all affected application subsystems are informed whether the requests have been granted or a QoS level has been changed and the configuration of the communication system is updated.

4.5 Discussion

In this section we discuss the various advantages and disadvantages of the different resource allocation policies.

Static Resource Allocation. The completely static approach is an approved allocation policy, which has been successfully applied for years especially in the domain of safety-critical embedded systems. The main benefit of the static approach is that resource allocations can be analyzed off-line, identifying potential design failures or eliminating resource conflicts.

However, the resource utilization achieved by this approach is inefficient compared to the other presented policies. The system is designed to cope with the maximum resource needs of all application subsystems—in case of the example introduced in Section 4.1 this means the sum of the maximum bandwidth requirements of all application subsystems. Although it can be shown that the communication resources are sufficient for implementing the presented DASs according to their specification (e.g., by using a QoS-based resource allocation policy), the available communication resources would be exceeded with the static resource allocation approach. The main reason for this is that many application subsystems inherently comprise various mutually exclusive modes of operation. To disregard this application knowledge

entails a resource allocation with lower complexity at the expense of a worse resource utilization.

Dynamic Resource Allocation. As long as enough resources are present in the system to fulfill the requirements of all DASs, the unrestricted dynamic allocation approach is very flexible and provides the possibility to achieve the most fine-grained allocation of resources to the individual micro components. In this case, it facilitates to strive for an optimal resource utilization. For instance, as soon as a request for communication resources is received, the RMS updates the communication schedule accordingly and notifies the requesting job about the change. This is a straight forward operation in case enough resources are available.

For a dynamic resource allocation policy, it is important that mechanisms are realized that permit to free unused resources. One way to accomplish this is using the assumption of cooperating jobs, i.e., individual jobs notify the RMS in case resources are not used by them any longer. A further approach would be to detect unused resources by the RMS (e.g., similar to the functionality of a garbage collector in Java) for instance by applying a *least recently used* policy on resource usage. Alternatively, an expiration time for granted resources could be defined. This way, jobs bear the responsibility to refresh resource requests before the expiration time elapses, otherwise the resources would be deallocated.

However, the biggest challenge for the unrestricted dynamic allocation policy emerges in case the overall resources of the system are not sufficient for all DASs (as it is the case in the presented example). The RMS has then to decide which DAS is of higher importance, e.g., has more utility for the overall system, than the others in order to determine which resource requests are granted and which are refused. For those decisions a vast amount of application knowledge has to be stored at the RMS, e.g., utility of the application subsystem for the overall system or granularity and interdependencies of the resource allocations. To achieve an efficient and valid resource allocation using a unrestricted dynamic allocation policy is therefore a challenging and complex task.

Consider the case in the exemplary application of Section 4.1, in which the DVD stream is already activated and at the same time the navigation system server requests a map update with the highest download rate, i.e., 90 Mbit/s. In this simple case, the available bandwidth of the internal network would be exceeded and the RMS has to decide which one of the following responses to the request it will choose:

1. The request is declined and the infotainment DAS is informed that requested bandwidth for the download is not allocated to the micro component.

2. The resources are granted to the infotainment DAS and the resources allocated to another DAS are reduced. The affected application subsystems are informed about this change.

3. The amount of communication bandwidth requested by the infotainment application subsystem is decreased to an amount for which sufficient resources would be available (e.g., from 90 Mbit/s to 50 Mbit/s).

Alternative *(1)* is unsatisfying, specifically because the still available bandwidth is enough for enabling the download of updated data with a reduced download rate, which would still satisfy the specification of the DAS. For alternative *(2)* the RMS has to decide, for which application subsystem it is possible to reduce the allocated resources (e.g., according to the description of the ESP application subsystem it has to be assured that no resources are withdrawn at any circumstances). In addition, the RMS has to determine the amount of which the resource allocation is decreased for the selected DAS, which is also necessary for alternative *(3)*.

This is a very complex challenge. While it is possible to reduce the communication bandwidth allocated to the data update stream more or less continuously (cf. Table 4.4), all other messages defined for the infotainment subsystem demand a fixed amount of bandwidth. This applies to the multimedia subsystem as well. Furthermore, this bandwidth reduction might have severe consequences on the functionality of the entire application subsystem. Consider, for instance, the reserved bandwidth for the control channel in the infotainment DAS. If this communication channel is removed from the internal network by means of resource management, no route information requests can be transmitted to the navigation system server on the one hand, and no data updates can be initiated by the navigation server itself—resulting in a failure of the entire application subsystem.

Restricted Dynamic Resource Allocation. The main difference between the unrestricted and the restricted dynamic resource allocation policy is that in the latter one, the distribution of resources between DASs is calculated off-line and remains stable during the lifetime of the system. Thus, the RMS is only empowered to (re)allocate resources within the boundaries of a DAS, which is similar to the concept of federated architectures where each subsystem is implemented on its own dedicated computer system. This way, the workload of the RMS is reduced and the allocation decisions are simplified, e.g., each application subsystem can specify an allocation strategy that is tailored exactly to its needs.

However, the static resource distribution between DASs hinders the RMS to exploit resources that are left unused (e.g., in case only a single video stream is selected rather than the PiP mode). This would result in a worse resource utilization than using the unrestricted dynamic allocation policy. For instance, with the present example, similar to static resource allocation, it is not possible to find a static resource distribution among application subsystems, as required for the restricted dynamic approach, that would fulfill the requirements of all DASs.

QoS-Based Resource Allocation. The QoS-based resource allocation policy is a trade-off between a fully flexible resource allocation as provided by the unrestricted

dynamic approach and the low complexity and low overhead induced by resource management as in the static approach. Obviously, the introduction of modes, possibly augmented with different levels of QoS, reduce the flexibility w.r.t. resource allocation compared to the unrestricted dynamic approach. Furthermore, the specification of modes is an efficient way to provide application knowledge to the RMS. The specification of degraded QoS levels for individual modes of operation of DASs empowers the RMS to resolve resource conflicts without changing the intended behavior of an application subsystem. For instance, by the use of degraded QoS levels it could be specified in the infotainment subsystem that the communication requirements for the download of updated information is permitted to vary from 100 kbit/s, 1 Mbit/s, 50 Mbit/s to 90 Mbit/s. However, in the concrete example given in Section 4.1 changing the resolution of the rear view camera stream from 640x480 pixel per frame to 320x240 pixel is not an allowed specification of a QoS level, since the intended behavior would change, i.e., the full quality display of the rear view camera stream will be reduced to a presentation of the stream with the resolution of the PiP mode.

Furthermore, at design time the use of modes for capturing all resource requirements of DASs allows for a simple specification of resource allocations that must not be changed during the systems' lifetime. For instance, for the ESP application subsystem only a single mode of operation without degraded QoS levels would be specified, which captures the communication requirements of all jobs. Since no degraded QoS levels are specified, the RMS is only able to grant either all resources to the application subsystem or no resources at all. This would represent a failure of the DAS in the given example.

Since for each application subsystem a finite number of modes is specified, an off-line analysis can be performed in order to determine the maximum resource requirements that may emerge during the lifetime of the system. With such an analysis, it could be guaranteed that all application subsystems, especially those requiring statically assigned resources like the ESP subsystem, will receive a sufficient share of the available resources to execute their functionality. In particular, w.r.t. integrated systems the ability to evaluate the affects of adding an additional application subsystem, possibly originating from a different application vendor, to an existing system, is of high importance.

The resource management approach followed in the TTSoC architecture aspires to combine advantages of the static resource allocation policy, as well as, the flexibility provided by the QoS-based resource allocation policy. Therefore, the RMS of the TTSoC architecture comprises two architectural elements. A dedicated architectural element, the *Trusted Network Authority* (TNA), assures that design-time allocations and resource guarantees are always preserved, while the so called *Resource Management Authority* (RMA) is concerned with processing resource requests according to a QoS-based resource allocation policy. The details of this policy and the executed resource management procedure are the focus of the next chapter.

Chapter 5

Resource Management in the TTSoC Architecture

In the previous chapters we have introduced the TTSoC architecture and have given an overview on different approaches to perform resource allocation. It is now the purpose of this chapter to present a resource management solution for the TTSoC architecture. The chapter starts with an introduction of the requirements on the resource management solution in Section 5.1, which is followed by a description of the devised resource management strategy in Section 5.2. Section 5.3 elaborates on the particular resources of the TTSoC architecture that are subject to resource management. This is followed by an explanation of the architectural elements for resource management. Subsequently, Section 5.4 is dedicated to the *Trusted Network Authority* (TNA) and Section 5.5 describes the *Resource Management Authority* (RMA).

5.1 Requirements on Resource Management

The concrete design of a real-time system and its dimensioning w.r.t. the amount of resources provided to the hosted application subsystems depend highly on the targeted domain of the application, e.g., automotive applications, multimedia applications, control applications, etc., and its demanded functional and non-functional (e.g., dependability) characteristics. For instance, in safety-critical applications, it is a mandatory requirement that the resources provided by the real-time system meet their worst-case load. In contrast, the design of non safety-critical applications, basically driven by economic forces, has typically to cope only with the average case of the anticipated resource usage and has to support dynamic resource allocation, respectively.

Dynamic resource management enables the system to react to changing resource demands of application subsystems by modifying the allocation of the available resources to the hosted application subsystems over time. This modification is dependent on (i) the actual resource demands of the application subsystem (e.g., communication resources, computational resources, power), (ii) the availability of resources

on the SoC, and (iii) the benefit for the user arising from the respective application subsystem. This permits a more efficient dimensioning of the system, since it is not necessary to consider the worst-case resource requirements of all application subsystems.

The SoC architecture aims at being deployed in various application domains supporting the integration of application subsystems (i.e., DASs) of mixed criticality. Therefore, it arises, on the one hand, the demand for the resource management solution to support resource guarantees, which are designed for the worst-case resource demand of safety-critical application systems. On the other hand, flexibility w.r.t. resource allocation becomes important, in order to be competitive in the realization of non safety-critical application subsystems.

Besides an improved utilization of the available resources, dynamic resource management can be used to improve the dependability of the SoC in case a micro component develops a permanent fault. Consider an SoC containing many micro components of identical hardware where functional differentiation is solely provided by application software. In case one of those micro components fails permanently, dynamic resource management can be exploited to relocate the affected function to a spare micro component. This would relax the requirement for 100% correctness for devices and interconnections, which may dramatically reduce the cost of manufacturing, verification, and test of SoCs based on the TTSoC architecture.

In order to cope with the functionality of the different targeted application domains of the TTSoC architecture, the following generic requirements on dynamic resource management have been identified:

Correctness of Resource Allocation: All DASs hosted on the SoC have to rely on dynamically created resource allocations. Thus, it has to be guaranteed that only correct resource allocations are generated, respectively, already correct resource allocation are not violated.

Guaranteed Resource Shares: Since the SoC architecture aims at being deployed for mixed criticality systems, resource management must ensure that for dedicated DASs (in particular safety-critical DASs) a certain amount of the shared resources is reserved and that this amount is protected by hardware mechanisms to ensure that it is never affected by any resource (re)allocation.

Bounded Reconfiguration Time: We demand from the resource management solution of the TTSoC architecture that after a (re)allocation of resources is requested, e.g., due to a change in the load of an application or by a diagnostic service that detects a permanent fault of one of the micro components, a new assignment of resources is calculated in bounded time. This is required in order to enable a discrimination between correct and defective behavior of the resource management.

To cope with these requirements, the SoC architecture provides the *Trusted Network Authority* (TNA) and the *Resource Management Authority* (RMA)—two separated architectural elements for realizing dynamic resource management. Figure 5.1

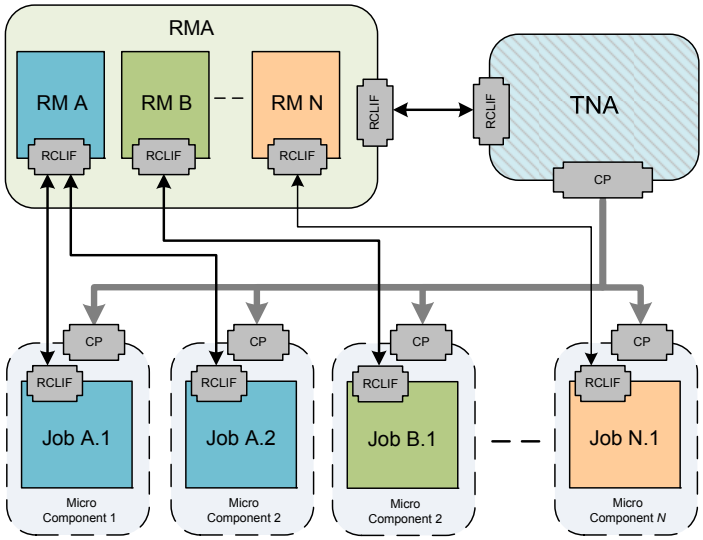


Figure 5.1: Dedicated architectural elements for resource management in the TTSoC architecture. The interfaces depicted in this figure are: Configuration and Planning Interface (CP), and Resource Coordination Linking Interface (RCLIF)

depicts this separation between TNA and RMA and shows the information flow from and to the individual interfaces of TNA, RMA, and each of the micro components. The *Configuration and Planning* (CP) *interface* depicted in Figure 5.1 is used by the TNA to assign resources to micro components, e.g., communication resources are assigned by updating the MEDL at each TISS accordingly. The interface of a job towards the RMA is denoted *Resource Coordination Linking Interface* (RCLIF). We consider this interface as part of the LIF of the job, because knowledge of reconfiguration capabilities of a job's service, e.g., different existing primary modes of its provided service, are required to completely understand a job's behavior. The RCLIF of a job towards the RMA is used by the job to disseminate reconfiguration requests to the RMA or to the respective *Resource Manager* (RM) within the RMA (see Section 5.5 for more details). The RCLIF between RMA and TNA is used to pass on resource allocations computed by the RMA to the TNA, which checks their validity and responds whether they are accepted or not.

5.2 Resource Management Strategy

As a consequence of the discussion in Section 4.5, we decided in favor QoS-based resource management policy. This approach requires capturing of application knowledge, which is performed by the explicit specification of modes of operations for each DAS hosted on the SoC. This specification is the focus of the next subsection. The interaction of jobs and RMA, as well as, RMA and TNA are the topics that are discussed subsequently.

5.2.1 Specification of Applications and Modes

As explained in Section 3.3, we consider a DAS as a functional part of the entire system, e.g., body electronics or multimedia subsystem in the automotive domain, which provides a well specified service that might change over time. In order to facilitate the capturing of those service changes, we restrict them to occur only within a priori known bounds, which are called *primary modes* of an application subsystem.

The change between primary modes is initiated by the DAS itself (e.g., indirectly triggered by a request from the environment) and not caused by the resource management solution of the SoC. Examples for primary modes in the previously given automotive example are different playback modes (e.g., audio or video playback) in the multimedia subsystem. A switch between primary modes represents always *a substantial change in the service of an application subsystem.* This change may affect temporal parameters of communication channels, the communication topology between cooperating jobs, or (in)activates a subset of the jobs of an application subsystem.

Additionally, for each primary mode a set of *degradation levels* is defined. Different degradation levels represent the same service of one primary mode, but with different QoS levels. The complete deactivation of a service represents a special case of a QoS level. Continuing the example mentioned beforehand, degradation levels of the primary mode *video playback* can be considered as different frame sizes and frame rates of the video stream in a multimedia system. Also the inactivation of the video stream can be considered as a degraded mode. Each degradation level is characterized by resource demands of the individual jobs realizing the application subsystem, as well as, a penalty. The penalty indicates loss of utility for the *entire system* when switching to a particular degradation level.

The specification of those modes of operation, i.e., primary modes and related degradation levels, has to comprise all information about the resource requirements of the entire DAS in the given mode. This information is exploited by the RMA for generating the resource schedule. Therefore, the definition of operation modes includes at least the following for every DAS and every mode:

Active jobs. In different primary modes a differing subset of constituting jobs of a DAS may be active. Consider for instance the multimedia example introduced in Section 4.1. A typical change of a primary mode would be to switch from displaying the DVD stream to the rear camera stream at the monitor. This entails a switch of the active jobs.

Job allocation. During the generation of the PAM (cf. Section 3.3), an allocation of jobs to micro components is generated. This information is exploited by the RMA to instantiate the required SoC-Channels and SoC-Ports for implementing the interconnection of cooperating jobs.

Micro component configuration. As elaborated in Section 3.2.1, the TISS provides a set of core platform services via the UNI to its host. Some of those services can be parameterized by the host itself (e.g., the programmable timer interrupt service), others require to be configured solely by the TNA, e.g., the watchdog service and the power control service. The actual configuration parameters of the latter services are part of the specification of every mode. For instance, consider a job for which the watchdog service is activated. When this job is deactivated in a particular mode, the watchdog service has to be deactivated as well. Otherwise, misleading diagnostic information would be generated by the respective TISS.

Communication topology. The UFIM of each DAS (cf. Section 3.3) comprises the specification of the entire communication requirements of the DAS in a platform-independent way. Some of the specified communication channels are mandatory for every mode, others are not. Therefore, a list of active communication channels has to be specified for each mode of operation. Furthermore, for each communication channel, a sender job, receiver job(s), and temporal constraints (e.g., bandwidth, phase alignment, etc.) have to be specified.

Using this system model, the task of dynamic resource management is to obtain that set of degradation levels for all DASs in a specific primary mode that yields the minimum penalty and for which the sum of the demanded resources is available. Since the specification of the primary modes and their degradation levels is done at design time of the application subsystems, it can be verified off-line that all possibly simultaneously occurring primary modes do not exceed the overall available resources. Furthermore, it is important that it is not possible for the resource management solution (i.e., the RMA in our case) to change the primary mode of a DAS, but only its degradation level.

For facilitating the composition of independently developed DASs, it is important that the penalty value denotes an absolute (in the meaning of system-wide) instead of an relative (i.e., only in the context of the single DAS) loss of utility. Otherwise, a prioritization of particular application subsystems may occur. Therefore, the penalty has to be determined in the context of the integration of all application subsystems to the overall system, i.e., during the transformation of the FIM to the PAM.

5.2.2 Interaction Pattern

The interactions between architectural elements of the SoC, as well as, interactions between the jobs with the RMA can be classified into three distinct stages. The first stage deals with the acquisition of resource requests at the RMA by interacting with the hosted jobs. This is followed by the computation and verification stage which is performed jointly by RMA and TNA. The final stage deals with the execution of the computed resource (re)allocation, which includes a configuration of the micro components by the TNA, as well as, a notification of the jobs about the conducted

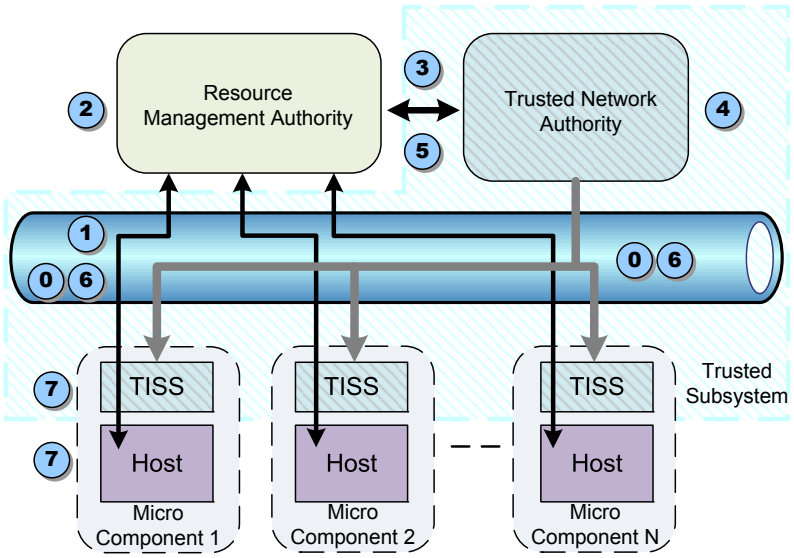


Figure 5.2: Different phases of resource management

changes by the RMA. Taking a closer look, these three stages can be subdivided into eight phases (cf. Figure 5.2), which are explained in the following. Please note that *Phase 0* is executed only once at the startup or after reset of the SoC, while the other phases (*Phases 1* to *Phase 7*) are executed periodically for every SoC reconfiguration.

Phase 0: The TNA writes the configuration of the TISS of each micro component according to an initial schedule, which is statically stored in the memory of the TNA. The purpose of this initial schedule is to pre-configure the SoC in order to enable an initial startup and further reconfiguration activities, i.e., at least the SoC-Channels between the RMA and all micro components that are requesting for resources have to be established. Optionally, statically configured *RMA-to-job* messages containing an initial configuration of (parts of) the jobs can be disseminated from the RMA to the respective jobs.

Phase 1: The jobs of the individual DASs hosted on the SoC send their resource requests to the RMA. As explained in the previous section, we decided in favor a QoS-based resource management strategy by using explicitly defined modes of operation for each DAS. Thus, a resource request is initiated by the job by requesting for a particular primary mode of its DAS.

Phase 2: The RMA processes the resource requests and performs a (re)allocation of the available resources according to the specification of the requested modes. This phase results in an adapted resource schedule describing a proposal for a new configuration of the SoC.

Phase 3: The newly generated resource schedule is sent from the RMA to the TNA for further verification.

Phase 4: In this phase of the reconfiguration procedure, the TNA performs a verification of the configuration proposal received from the RMA. The TNA acts as guard for the system configuration by checking the communication schedule for its correctness and by ensuring that the allocation of protected resources is not violated.

Phase 5: Based on the results of the previous phase, the RMA is informed, whether the configuration proposal is accepted, or not.

Phase 6: In Phase 6 the results of the reconfiguration procedure are made visible to the architectural elements of the SoC, as well as, to the hosted jobs. If the TNA considers the configuration proposal as correct, the TISSs of all micro components are updated accordingly and the RMA informs the respective jobs whether their requests have yield to any changes in the resource allocation or have any effect on the operational mode of a job. If the configuration proposal is rejected, the currently active TISS configuration remains active and the jobs are notified about the refusal.

Phase 7: The last phase of the reconfiguration process is concerned with the actual reconfiguration of the TISSs and the hosts based on the inputs received from TNA and RMA, respectively. This reconfiguration is triggered at all micro components of the SoC at the same global point in time, denoted as *reconfiguration instant*. Therefore, it has to be ensured that the static schedule of the SoC-Channels required for executing *Phase 6* guarantees a timely reception of the configuration information at each micro component, i.e., before the reconfiguration instant.

Static Message Schedule for Resource Management

The chronology of the individual phases of resource management as defined above in combination with the necessary message exchanges is depicted in Figure 5.3—except for Phase 0 which is executed only once at the startup. The here depicted sequence of the reconfiguration phases is recurred periodically with a period denoted as *reconfiguration period*.

The message exchanges over the NoC, which occur at Phase 1 and Phase 6, are realized by static SoC-Channels. Since they are mandatory for the correct function of the resource management procedure, they are not subject to reconfiguration and are protected by the TNA. These static SoC-Channels can be classified into three different categories: The first category (related to Phase 1) comprises the static channels between a selected set of micro components and the RMA. Only those micro components which are permitted to invoke resource requests are provided with such a channel to the RMA. The second category (related to Phase 6) comprises static channels from the TNA to each micro component of the SoC. These channels are used for updating the TISS configuration by the TNA. The third category (also related to Phase 6) comprises static channels to all micro components of the SoC in

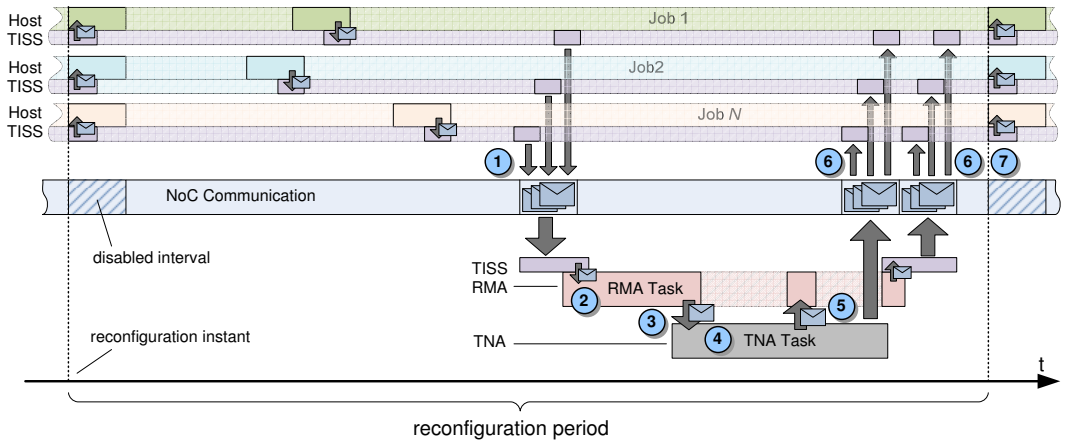


Figure 5.3: Resource management message sequence diagram

order to inform the jobs hosted at the respective micro components about potential changes of the resource allocation, which have been either invoked by themselves or by another job.

As depicted in Figure 5.3 it is not mandatory that the initiation of a resource request by a job is aligned with its actual transmission over the NoC to the RMA. The reason for this is that the information disseminated in this first phase of the reconfiguration procedure exhibits state semantics and is sent by the use of periodic time-triggered messages. Thus, repeated resource requests of a job within one reconfiguration period are overwritten. So, it is possible to activate the computations at the RMA only once in each reconfiguration period, namely after the reception instant of the last statically scheduled SoC-Channel relating to Phase 1, regardless whether a resource request has been disseminated over this channel or not. This is important for simplifying the computations of the RMA, since it is ensured that the RMA has a stable view of all requested resources from the individual DASs before starting its computations.

After the newly generated configuration information has been disseminated to the micro components (cf. Phase 6), the actual reconfiguration of the TISSs and the hosts can be performed. This reconfiguration is triggered at the reconfiguration instant, which is an SoC-wide consistent global point in time, at which the new configuration becomes active at all micro components. The *reconfiguration interval* is the time required for performing the update of the affected data structures at the TISSs and jobs. During this time interval all communication activities on the NoC are disabled. The length of this interval is determined by the micro component requiring most time for performing its reconfiguration. It represents an important value for the parameterization of the timing of the resource management solution, because when constructing a communication schedule, the RMA has to ensure that TDMA slots during this interval remain unused.

Since the generation of the resource schedule (cf. Phase 2) and the execution of the correctness checks (cf. Phase 4) have to be performed sequentially, the sum of the

times required for those two phases determines the minimal reconfiguration period of the SoC component.

5.3 Manageable Resources in the TTSoC Architecture

Before we explain how resource management is actually performed in the TTSoC architecture, this section identifies those resources of the SoC that are subject to reconfiguration. These resources are the time-triggered NoC, the micro components by altering their configuration, and the consumed power of the SoC chip.

5.3.1 Time-Triggered Network-on-Chip

The central shared resource of the TTSoC architecture is the time-triggered NoC. Since it is shared among all micro components, a host that is able to monopolize the access to the NoC, directly impairs the services provided by all other hosts. Therefore, the coordination of this resource is performed via the generation of a conflict free time-triggered schedule that is protected by the TISSs of each micro component. In a static system the calculation of a conflict-free message schedule and the configuration of the individual micro components could be performed off-line. For dynamic systems, however, management of this resource means to dynamically adapt the message schedule during runtime in order to fulfill the (possible changing) temporal requirements, e.g., bandwidth or latency, of the messages utilized by a particular application.

So far, we have not made any assumption on the actual implementation of the NoC, e.g., the physical structure of the interconnect (bus, mesh, etc.). It is the intention of the TTSoC architecture that the choice of implementation of the NoC is not restricted as long as it permits the predictable, time-triggered transport of periodic and sporadic messages between micro components. For the allocation of periodic sending slots of the time-triggered NoC, we have introduced in Section 3.2.2 a communication primitive, denoted as *pulsed data stream* [Kopetz, 2006]. Please note that even pulse data streams represent a highly generic communication primitive abstracting from the concrete implementation of the NoC.

However, for actually managing the communication resources, i.e., calculating a new time-triggered communication schedule, details regarding the physical structure of the NoC are vital parameters. In the course of the European project *DECOS*, a first implementation of the TTSoC architecture has been devised, which employs a bus-based implementation of the NoC [DECOS, 2006]. Additionally, within the course of the national project *TT-SoC* two further implementations are planned including an improvement of the bus-based implementation founded on the work of the DECOS project and a full-scale NoC implementation exhibiting a mesh structure [TT-SoC, 2007a]. While the full-scale NoC supports the concurrent transmission of fragments originating from multiple pulsed data streams, only a single fragment of one pulsed data stream is permitted to be in transit at any time in the bus-based

implementation. Hence, it is obvious that the generation of a conflict-free message schedule is different for the implementation alternatives. For instance, a bus-based implementation apparently requires that any two fragments must not be scheduled at the same instant in order to create a conflict-free schedule.

This thesis describes in Chapter 6 a case study of the resource management solution that is based on the DECOS implementation of the TTSoC architecture. Hence, the algorithms for managing the communication resources presented in that chapter are tailored to the requirements of a bus-based NoC realization.

5.3.2 Micro Component Configuration

The micro components (more precisely the hosts of the micro components) provide the computational resources for the execution of jobs. The initial allocation of micro components to jobs is performed offline at design time of the system, i.e., during the generation of the *Physical Allocation Model* (PAM) in the model-based development process. In particular for safety-critical application subsystems, these assignments are typically static and not subject to change over time.

However, to react to changing requirements of the applications or on changing availability of computational resources, the TTSoC architecture provides the capability to reallocate particular micro components to different jobs. On an SoC containing multiple general purpose micro components (e.g., realized by a host implemented as a general purpose CPU), dynamic allocation of micro components to jobs can be achieved by loading system software (e.g., a simple operating system) and application software onto the micro components during runtime.

Besides changing the entire application that is executed on a particular micro component, the requirements of an application regarding resource demand will also change over time. As described beforehand, we assume that those expected changes are specified using modes, which are defined during the design time of the system. Such a mode change of an application subsystem could have impact on various parameters of a micro component.

Consider the simple example depicted in Figure 5.4. It depicts a simplified path planning system for an autonomous mobile robot consisting of a planning job and two sensor data processing jobs for providing information about the position of the robot. While one of the sensor data processing jobs provides odometry information to the job responsible for path planning, the other job provides GPS information. In order to improve the energy efficiency of the system, solely the odometry information is used at the planning job for calculating the position of the robot and the micro component hosting the GPS job is put into standby, e.g., a primary mode denoted as *mode 0*. Only for determining the initial position and for correcting the accumulated errors of the odometric sensor data, the GPS sensor data is used (e.g., expressed by the specification of a primary mode *mode 1*).

Besides updating the message schedule of the NoC, which concerns resource management w.r.t. communication resources, the management of the micro component

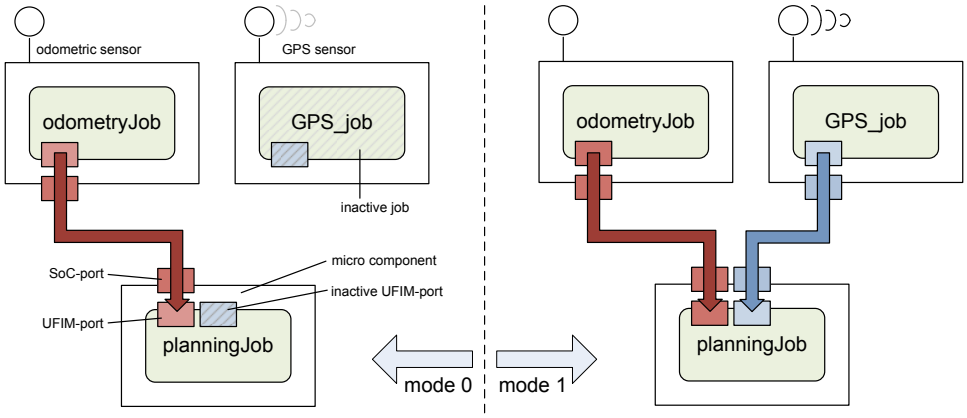


Figure 5.4: Changing the mapping of UFIM-ports to SoC-ports

configuration after a change from mode 0 to mode 1 entails the following activities: First, the micro component (more precisely the host of the micro component) has to be activated from standby (e.g., using the power control service of TISS if supported by the actual implementation of the micro component). In addition, the RMA has to allocate new SoC-ports at the micro component hosting the planning job and the micro component hosting the GPS sensor job, in order to enable the establishment of an SoC-channel for data exchange. Finally, the newly generated ports have to be made visible to the jobs. This is realized by configuring the mapping of UFIM-ports to SoC-ports which is stored within each job (or within its underlying execution environment). This has to be realized for both affected jobs, the GPS job, as well as, the planning job.

In addition, to change the allocation of jobs to micro components, respectively, to activate or deactivate particular jobs and all corresponding channels and ports, we assume that each host permits via its CP interface the modification of its computational resources. In its most basic realization this means that the host supports only two modes: the entire host can be switched off or on again. In refined host realizations various gradations of the computational resources of the host could be distinguished. Then, it is one challenge of the RMA to select the appropriate operating mode of the host, which is, on the one hand, sufficient to provide the computational resources required for the timely completion of the hosted jobs and, on the other hand, is efficient regarding the power consumption of the host.

5.3.3 Power

Similarly to the time-triggered NoC, power can also be seen as a shared resource on the SoC: A faulty micro component that consumes more power than permitted is able to directly influence the correct operation of the entire chip. Typically, from two different aspects power plays an essential role in embedded systems, namely, maximum power dissipation and energy consumption.

Maximum Power Dissipation: For every silicon chip, the maximum dissipated power is a limit arising from the physics of the chip that constrains the chip design. In most cases, exceeding this limit entails irreparable damage of the chip caused by heat development exceeding the critical values of the deployed materials. Thus, being able to take influence on the dissipated power of an SoC enables the system to react to changing environmental conditions and preserve the SoC from damage. Roughly speaking, the total power dissipation of an SoC chip results from the power dissipation of a single transistor on the chip times the number of transistors a chip comprises. A detailed formulation of the sources of power dissipation in CMOS technology is given in Section 2.2.3.

Energy Consumption: The energy consumed for the completion of a task is the product of the dissipated power and the time that is required for the task completion. A system using a finite source of energy, e.g., a battery-operated device, is an energy-constrained system. For such a system, the improvement of the energy consumption is comparably more important than the reduction of the maximum power dissipation. In general, reducing the power dissipation and improving energy consumption are two different system design goals [Unsal and Koren, 2003]. For example, for the reduction of power dissipation *Dynamic Voltage and Frequency Scaling* (DVFS) approaches (cf. Section 2.2.3) can be used. However, by reducing the clock frequency it is possible that the low performance of the SoC will increase the actual energy consumption, because of the extended time required for the completion of the task.

Influencing Power/Energy Consumption:

The TTSoC architecture supports different possibilities for parameterizing the configuration of the SoC to have a beneficial influence on the power or energy consumption of the chip. First, *Dynamic Power Management* (DPM) techniques (cf. Section 2.2.3) can be deployed for micro components. On the one hand, unused micro components like spare micro components that would only be exploited in case a permanent failure of another micro component on the SoC occurs, can be put into an energy efficient standby mode, e.g., by exploiting the power management service of the TISS. Similarly, the idle time of hosts could be exploited. For instance, the RMA can use knowledge about the points in time at which a particular host is idle, e.g., a host that executes a job that is triggered only after the reception of a message. By putting such a host into a standby mode, the overall power consumption of the micro component could be reduced. On the other hand, more fine-grained operation modes of individual micro components could be investigated, which support, for instance, the deactivation of particular parts of the TISS or the host. Consider, for instance, the functional elements within the TISS implementing the network interface towards the NoC. When the processing of pulsed data streams of different periods is realized by dedicated hardware, parts of the hardware can be deactivated (e.g., by means of clock gating), if no pulsed data stream of the associated period is part of the current

message schedule. This would lead to a reduction of the dynamic power consumption of the respective TISS.

Likewise, unused resources of the NoC can be exploited for achieving power savings. As described in [TT-SoC, 2007b, p. 29f], the full-scale NoC that is implemented in the course of the *TT-SoC* project consists of so-called fragment switches, which form the topology of the NoC. Depending on the actual schedule of pulsed data streams and their routing, at any time a particular amount of fragment switches is idle. By setting this idle fragment switches into a power-efficient standby mode, the power consumption of the entire NoC can be reduced. Since the extent to which a reduction can be achieved depends on the number of idle fragment switches and their idle times, appropriate scheduling and routing of pulsed data streams can improve the power consumption of the NoC.

In addition to DPM techniques, with the use of DVFS (see Section 2.2.3) further power and energy savings could be achieved. Nowadays, several processor families, as for example the Intel® XScale™ processor family [Intel, 2000], support the optimization of the power dissipation of a processor produced for the completion of a given workload by adapting the performance per mWatt ratio via DVFS techniques. This would again require in-depth application knowledge at the RMA, in order to decide to which extent it is permitted to lower the power dissipation at the expense of performance.

5.4 Trusted Network Authority

The TNA together with the time-triggered on-chip network and the TISSs of each micro component form the trusted subsystem of the SoC. Since failures within these core architectural elements are likely to cause failures of the entire SoC, a thorough design of the TNA is of great importance. In case the SoC is deployed for building distributed systems for applications with high dependability requirements, the TNA has to be certified at least to the same level of criticality as required for the most critical application subsystem hosted on the SoC. In order to ease such a certification, the complexity of the TNA is kept as low as possible. Therefore, only a small set of stable, application independent services are provided by the TNA:

1. Protection of the resources of privileged application subsystems
2. Configuration of the micro components via their CP interfaces
3. Establishment and maintenance of the global time

In the following subsections we address each of those services in more detail.

5.4.1 Resource Protection for Privileged Application Subsystems

The computation of the allocation of resources to individual DASs in case of a reconfiguration of the SoC, e.g., allocation of communication resources and power among

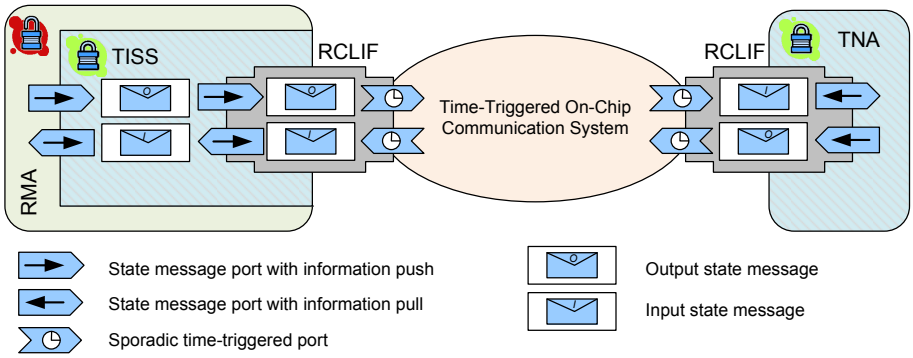


Figure 5.5: Data flow between RMA and TNA

DASs, as well as, the configuration of computational resources within single micro components, is performed by the RMA. The RMA periodically disseminates the (updated) resource allocation of the SoC to the TNA, which checks the validity of the resource allocation. Based on this configuration information, the TNA updates the SoC via the CP interfaces of the individual micro components.

Figure 5.5 depicts this data flow from RMA to TNA. Since the RMA is not part of the trusted subsystem, it is realized as a dedicated micro component consisting of a host (implementing the functionality of the RMA, denoted as RMA for short in the following) and a TISS, which protects the on-chip interconnect.

The inner ports of the TISS towards the RMA are realized as state message ports with information push semantics for the output ports and information pull semantics for the input ports. This means, the point in time at which the state message that is contained within the TISS is written or read is always in the sphere of control of the RMA. The inner port of the *Resource Coordination Linking Interface* (RCLIF) towards the TISS is also realized as a state message port (controlled by the TISS). The outer port of the RCLIF of the RMA micro component is a time-triggered state message port supporting sporadic time-triggered messages, i.e., the points in time at which messages are permitted to be disseminated or received are a priori defined, but the actual dissemination/reception depends on whether new information is available.

The outer ports of the RCLIF of the TNA are also realized as time-triggered state message port supporting sporadic time-triggered messages. Each incoming message replaces the state message stored in the RCLIF. The inner ports of the RCLIF towards the TNA are realized as state message ports, i.e., it is in the sphere of control of the TNA to decide when to read, respectively write the state messages. Using these port characteristics, the message exchange between RMA and TNA (and vice versa) is performed autonomously as soon as the state message ports within the respective RCLIF are updated. In addition, the computations of TNA and RMA are not interrupted by the reception of messages.

The configuration data is disseminated in a state message via a dedicated channel from the RMA to the TNA. The layout of this message is illustrated in Figure A.2 and is explained in detail in the appendix in Section A.2. The message is grouped into

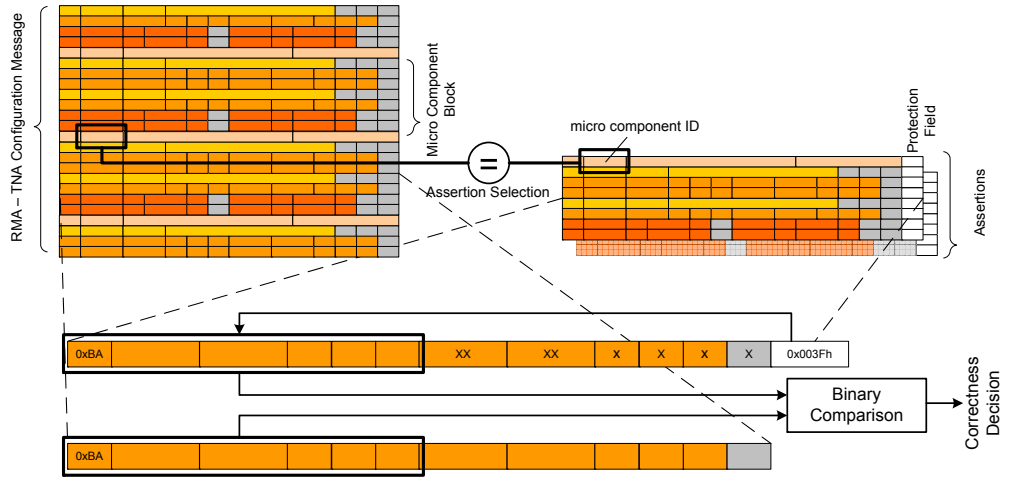


Figure 5.6: Protection of guaranteed resources by the TNA

several blocks—one for each micro component. Each block comprises the following sequence of frames. (The detailed content of the frames is explained in Section A.2.):

1. Each micro component specific block is started by exactly one TISS/host configuration frame. This frame is used to parameterize the services provided by the TISS to the host, except the communication service.

2. Afterwards, a (possibly empty) sequence of frames follows, which holds the description of the input ports, i.e., those parts of the micro component's MEDL specifying the messages to be received.

3. Each micro component configuration block is concluded by a (possibly empty) sequence of output port description frames, which specify the send instants of the particular micro component. The union of input port descriptions and output port descriptions form the MEDL of the micro component.

To protect resource allocations, the TNA enables the specification of assertions, which operate on the information contained in the state message received from the RMA. These assertions have to be generated off-line and are either part of the functionality of the TNA software itself or downloaded to the TNA at the startup of the system. The concept of those assertions is depicted in Figure 5.6.

As depicted in Figure 5.6, the assertions comprise the identical structure as the individual blocks of the configuration message, which is disseminated from the RMA to the TNA (cf. Figure A.2 in the appendix). Thus, each assertion covers exactly the configuration of one micro component. In addition to the data fields of the RMA–to–TNA configuration message, each frame is extended by a *resource protection field*. The resource protection field is simply a sequence of bits, where each bit is related to exactly one field in the respective frame and indicates whether the field should be protected by the assertion or not. In the example depicted in Figure 5.6 the

value of the protection field is *0x003F*, which equals *00000000 00111111b* in binary representation, denoting that the first 6 fields of the actual frame are protected by the assertion.

We opt for this simple realization for the specification of assertions to minimize the computational overhead and complexity at the TNA, because for this part of resource protection only bit streams have to be compared. The actual resource protection is performed at the TNA as follows: The configuration message received from the RMA is analyzed until a new TISS/host configuration frame (indicated by a particular type field as the first byte of the frame) is detected. The micro component ID contained in this frame is utilized as an index to identify the matching assertion for the particular micro component configuration, if available. The compliance of the actual configuration message to the selected assertion is verified by a bit-wise comparison of the assertion and the data contained in the message. However, only those fields which are marked by the protection field, have to be included into the comparison. An implementation of the resource protection using XML for the specification of assertions is described in Section 6.3.4.

For ensuring the protection of guaranteed resources and the non-interference of message transmissions of jobs belonging to different DASs, the time-triggered message schedule has to be checked for correctness within the TNA. A correct, i.e., conflict free, message schedule on the time-triggered on-chip network has to ensure that fragments of individual pulses must not interfere. A pulse fragment is disseminated on the on-chip network in a single TDMA slot. Thus, the TNA ensures that for any two pulsed data streams none of their fragments are disseminated at the same TDMA slot.

In case the TNA detects a violation in the message schedule or in any of the assertions for resource protection, the entire configuration message is discarded and the configuration data of all micro component remains unchanged. Furthermore, the RMA is informed that the actually disseminated configuration message has been rejected.

5.4.2 Micro Component Configuration

As depicted in Figure 5.1, only the TNA is able to alter the configuration of a micro component regarding component-wide global parameters like the message schedule of the time-triggered on-chip network. Via the CP interface of the micro component following parameters of a micro component can be configured (see Section A.1 in the appendix for a detailed description of the layout of the CP interface):

1. **Ports towards the NoC:** Each port of a micro component is associated with a single pulsed data stream. The CP interface permits the configuration of the temporal parameter of the pulsed data stream, i.e., its period and phase offset to the start of the period, as well as, its length (measured in the number of fragments the pulse consists of), and the direction of the port (input or output).

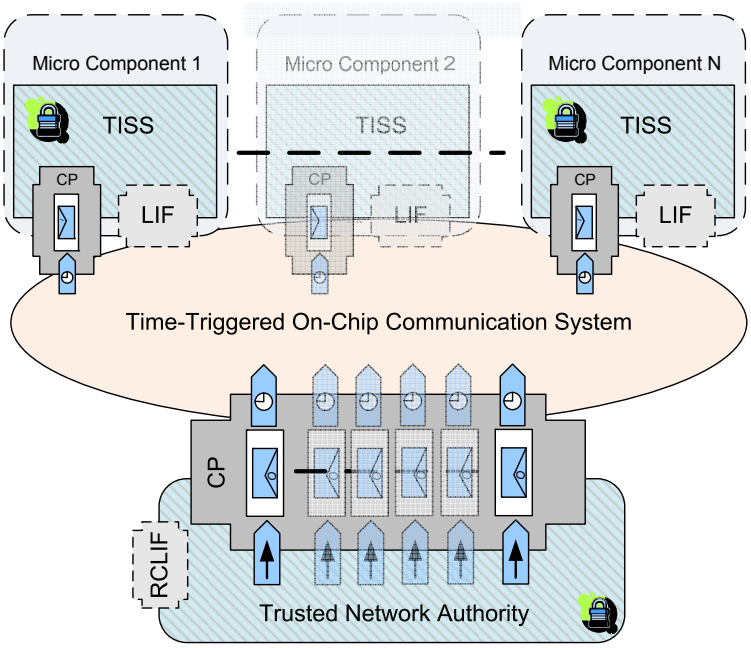


Figure 5.7: Data flow between TNA and micro components

2. **Service level of a micro component:** We assume that each micro component can be put into different service levels, each characterized, e.g., by the provided performance and the demanded power dissipation. These service levels are managed by the power control service of the TISS. The parameterization of this service is performed by the TNA via the CP interface. Depending on the implementation of the host, setting the service level of a micro component affects, for instance, the supply voltage and operation frequency of a host or enables/disables special hardware blocks located within the host. However, the TISS requires at least the ability to physically power off/on the host.

3. **Watchdog functionality of the TISS:** In order to detect crash failures of the host and to possibly restart a failed host, the TISS acts as a watchdog for the host. The period by which the life sign has to be set by the host is parameterized by the TNA (cf. the field *WDP* in Figure A.1).

The design of the micro component must guarantee that these configuration parameters cannot be altered by software from inside a micro component. So, it is not possible for faulty software inside a host to interfere with a correct operation of the on-chip network and to take unintended influence on the power dissipation of the micro component.

Analogously to the description of the TNA–to–RMA interaction, Figure 5.7 depicts the internal structure of the interfaces of micro components and the TNA. The configuration information for a single micro component is contained in a state message, which is sent by the TNA over a dedicated channel on the on-chip network.

The CP interface of each TISS comprises one outer input state message port for holding the configuration message. This port is a time-triggered state message port served by the TNA, which periodically disseminates the corresponding configuration messages to each micro component.

The CP interface of the TNA comprises output state message ports for each micro component on the SoC. The outer ports of the CP interface of the TNA are realized as periodic time-triggered state message ports. The inner output ports are realized as state message port with push semantics, i.e., the point in time for updating the state messages within CP interface is in the sphere of control of the TNA.

5.4.3 Establishment and Maintenance of the Global Time

The on-chip network is realized as a time-triggered communication system, where TDMA is used for arbitrating the communication medium. To each TISS a number of conflict free sending slots described by period and phase are assigned. Therefore, a global time base for all TISSs is mandatory, which is established and maintained by the TNA.

We assume that the TISSs are free of design faults—they are part of the trusted subsystem of the SoC—and each micro component forms a *Fault Containment Region* (FCR) with respect to software faults within the host of the micro component. However, with respect to transient hardware faults, we regard the entire SoC component as a single FCR and thus do not assume that individual micro components fail independently in case of transients. Therefore, fault-tolerance mechanisms, e.g., a TMR configuration [Lyons and Vanderkulk, 1962] of multiple SoCs, have to be utilized for building ultra-dependable systems. Within a single SoC, it is therefore sufficient to implement a non fault-tolerant clock synchronization strategy for the establishment of the global time. We propose to use the TNA as a central time master that establishes the global time by periodically updating the global time values stored in all TISSs accordingly, e.g., by the use of rate correction.

When an external time-reference is available, e.g., the SoC component is connected via the gateway to an external cluster, the TNA in combination with the gateway can perform external clock synchronization and synchronizes the SoC internal global time with the time base of the external cluster. However, due to the technology differences of the on-chip network compared to the network of the external cluster, we assume that the precision of the SoC component-wide time base is orders of magnitude better than the precision of the time base of the external cluster.

5.5 Resource Management Authority

The resource management authority (RMA) performs the scheduling of the available resources of the SoC. We consider communication resources (e.g., bandwidth and

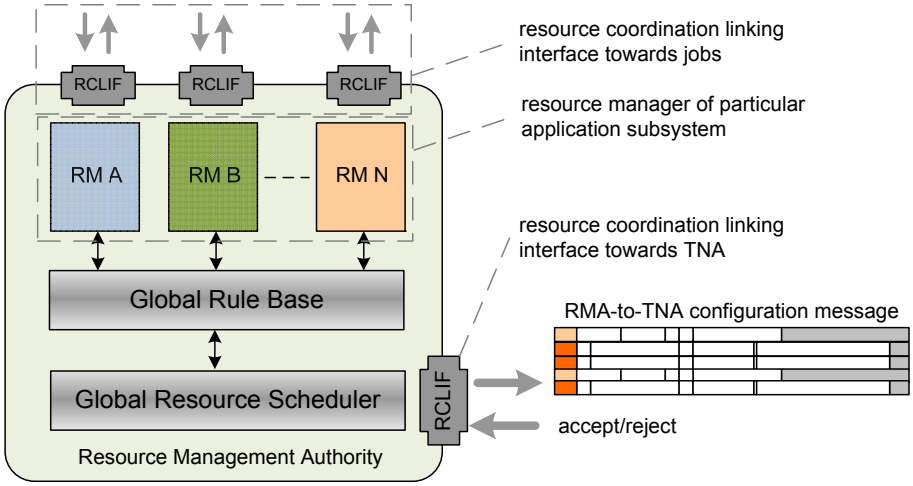


Figure 5.8: Constituting parts of the RMA

latency of the on-chip network), computational resources (e.g., allocation of jobs to dedicated micro components), and power (e.g., maximum power dissipation as a worst-case boundary) as integral resources that are managed by the RMA.

In contrast to the constituents of the trusted subsystem, certification issues are not the focus of the design of the RMA. Thus, an increased functionality at the expense of higher complexity can be realized. Therefore, we regard the RMA as a dedicated micro component that is realized by a TISS and a host, whereas the TISS ensures that error propagation from the RMA to the trusted part of the SoC is prohibited.

Figure 5.8 depicts on a high abstraction level the internal structure of the RMA. The RMA consists of several *resource manager*, each responsible for a single application subsystem, a so-called *global rule base*, which is instantiated during the integration of entire system and resolves resource conflicts between application subsystems, as well as, a *global resource scheduler* module.

For each DAS of which a job is hosted on the SoC and which should utilize the resource management facilities of the SoC, an application-specific resource manager could be realized within the RMA. The purpose of those resource managers is to receive and process resource requests of the jobs of a single DAS. We opt for an own resource manager for each application subsystem hosted on the SoC for the following reasons:

1. The way of efficiently raising resource requests may differ from application subsystem to application subsystem and should not be restricted by the design of the RMA. For instance, in the resource management strategy described in Section 5.2, resource requests are based on an explicitly defined limited set of operation modes. In such an approach, the resource requirements of each job for each respective mode have to be known and defined a priori. The resource

coordination between the jobs and their resource manager can be efficiently performed by requesting the transition from one operation mode to another. However, for other DASs it either may not be possible to define all operation modes in advance or may be to inefficient. For those it should be possible to explicitly request resources from the resource manager (e.g., a bandwidth enhancement from 100 kbit/s to 250 kbit/s due to a changing environment of the job). This flexibility could be achieved, if the RMA provides the possibility to deploy its own resource manager to several DASs.

2. Processing of resource requests requires in-depth knowledge of the application. Therefore, the resource manager can be seen as an outstanding job of the DAS, which has the purpose to coordinate the required resources within the DAS and which is qualified to communicate changes in the resource allocation to the other jobs of its DAS. However, it is important to note that the resource manager within the RMA is not able to physically assign resources to an application subsystem; this can only be done by the TNA.

Moreover, the separation of the RMA functionality into the application-specific resource manager, the global rule base and the global scheduler module as depicted in Figure 5.8 facilitates the integration of application specific knowledge from different application subsystem vendors for the realization of the RMA. The constituting parts of the RMA are outlined in the following:

Application-Specific Resource Manager

The RMA interacts with jobs via a dedicated resource manager for each application subsystem. The protocol with which the jobs are able to request resources or are able to free allocated resources is application specific. An example using dedicated modes is described in the case study in the following chapter. On the one hand, the purpose of the *application-specific resource manager* is to resolve resource conflicts within the DAS it belongs to and, on the other hand, to transform the resource requests of the jobs to a consistent format, which is supported by all application subsystems.

Consider as the common representation the specification of a finite number of primary modes for each DAS, each of them defined by the number of active jobs and its resource requirements, e.g., execution time on a particular micro component, bandwidth requirements, and expected power dissipation. It is the task of an application-specific resource manager to map received resource requests like the enhancement of the bandwidth that is granted to a particular job, onto those predefined primary modes, for instance, by selecting the next mode that fulfills the requirements of all jobs.

Global Rule Base

The actual resource allocation is performed via the interaction of the individual application-specific resource managers and the *global rule base*. The global rule base

has the purpose to check the acceptance of the resource allocations generated by the application-specific resource managers and resolves SoC-wide conflicts if necessary. Therefore, the global rule base contains for each primary mode of an application subsystem a list of degradation levels with less resource requirements. Those degradation levels have to be defined by the application subsystem developer, and represent resource requirements for all jobs that are sufficient for the DAS to provide its service in a degraded form, i.e., at a lower QoS level.

During system integration, for each of the degraded service modes penalties are defined. These penalties represent the importance of a particular application subsystem for the resulting integrated system and indicate how likely the global rule base decides to put a particular application subsystem into a degraded mode. Using this method, assigning a very large penalty, for instance the value infinite, to safety-critical application subsystems ensures that the resources assigned to those applications are treated by the RMA as being unchangeable.

Global Resource Scheduler

Based on the decisions made at the application-specific resource managers and the global rule base, the task of the *global resource scheduler* is to generate a schedule that meets the requirements of all jobs. The schedule is forwarded to the TNA, which is responsible for checking the validity of the resource allocation. Depending on the time bound defined for a single resource management activity, in case a feasible schedule cannot be found by the global resource scheduler, either the global rule base is invoked reiteratively or the reconfiguration request is reported to the jobs as non-feasible.

Recall the example of an infotainment application scenario introduced in Section 4.1: A job outside the boundaries of the SoC requests directly from the navigation job the download of navigation information via the gateway. In order to provide this service to the requester, the navigation job demands an additional communication channel to the gateway. In case the navigation job explicitly requests the change of a primary mode, the application-specific resource manager would pass on the request to the global rule base. Otherwise, the application-specific resource manager has to map the request onto a particular primary mode that would fulfill the resource requirements.

If sufficient resources are already assigned to the job in the current primary mode, the request can be autonomously granted by the application-specific resource manager to the job. Otherwise, it has to change the primary mode of the application subsystem and needs to forward this information, i.e., the new mode and therewith the updated resource requirements of the infotainment application subsystem, to the global rule base. On the basis of all actually scheduled primary modes of all application subsystems hosted on the SoC, the global rule base decides how to handle the request. This can lead to the following alternatives:

1. The new primary mode of the infotainment application subsystem requires

less resources than already granted and all other application subsystems do not reside in a degraded service level, i.e., the required resources are assigned to each application subsystem. In this case, the change of the service mode of the infotainment application subsystem would be granted and the resource information is forwarded to the global scheduler module.

2. The new primary mode of the infotainment application subsystem requires less resources than already granted and there is at least one application subsystem that resides in a degraded service level. Again, the change of the primary mode would be granted to the infotainment application subsystem. Furthermore, it would be examined in the global resource scheduler if the released resources can be utilized by one of the application subsystems which currently reside in a degraded service level.

3. The new primary mode of the infotainment application subsystem requires additional resources and there are still enough resources available. This case would be identically handled as the one described in (1).

4. The new service mode of the infotainment application subsystem requires additional resources and the remaining resources are not sufficient for all application subsystems to provide their service. In this case the penalties assigned to the degradation levels of each application subsystem would be analyzed. The penalty of a degradation level represents the cost it would take to execute a particular application subsystem in a degraded service mode, i.e., the importance of a particular primary mode of an application subsystem for the entire integrated system. Thus, an optimal selection of degradation levels of all application subsystems entails the minimization of the overall penalty.

Based on the resource information forwarded by the global rule base, the scheduler generates a resource schedule and a resource allocation for the entire SoC. Subsequently, the new configuration data is passed on to the TNA, which verifies the correctness of the schedule including the protection of the resources of privileged application subsystems. The new configuration becomes only valid, after the verification by the TNA has been successfully completed and a positive response has been received by the RMA. Afterwards, the new resource allocation is made visible by the application-specific resource managers to all affected jobs.

Chapter 6

Case Study

This chapter describes the implementation of an integrated real-time system that serves as a case study for the presented resource management concepts. This demonstrator has been developed in the course of the European project DECOS through the effort of several people. It provides a first proof of concept of the TTSoC architecture by emulating the functionality of a single SoC by means of a time-triggered distributed real-time system.

The chapter starts with an introduction of an exemplary automotive application in Section 6.1 that defines the requirements for resource management in the case study. Afterwards, the physical setup of the SoC emulation is explained in Section 6.2. This is followed by a detailed description of the implementation of the TNA functionality in Section 6.3 and the RMA functionality in Section 6.4.

6.1 Exemplary Automotive Application

Within the course of this case study we have implemented a simplified automotive application. The structure of this application is depicted in Figure 6.1. For sake of simplicity, the TNA is not depicted in this representation. The scope of the application is to emulate the acquisition of environmental values, e.g., road temperature and vehicle dynamics, via dedicated *sensor jobs* forming a *data acquisition DAS* and their further processing in an *information DAS*. To this end, these sensor values are imported via a gateway in the *information DAS*, which should enable the visualization of sensor data and of navigation information. For the generation of the navigation information, we assume that the *navigation job* exploits the data comprising the vehicle dynamics.

Since the focus of this exemplary application is the demonstration of resource management w.r.t. communication resources and to demonstrate the reconfiguration capabilities of the TTSoC architecture, the application functionality of the jobs is only emulated. This means, instead of disseminating real sensor data, the jobs (sensor jobs, navigation job, and rear camera job) produce their output values by

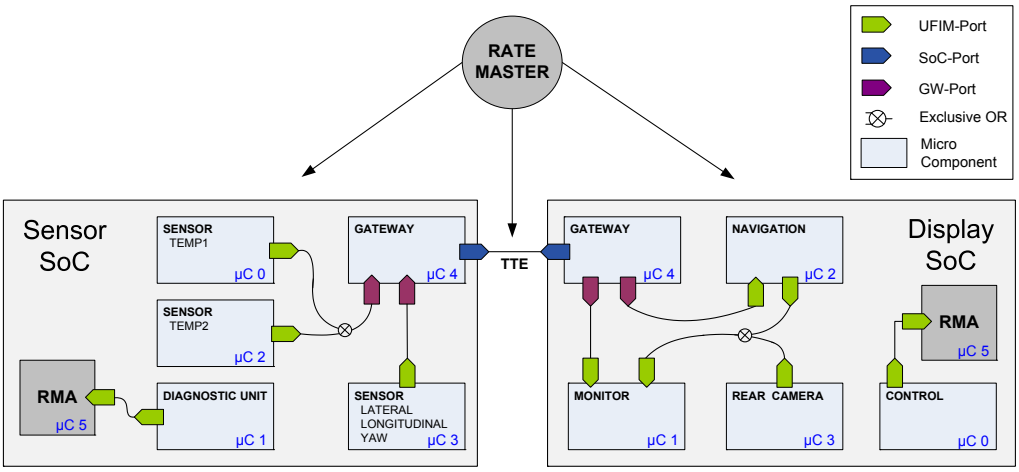


Figure 6.1: Structure of the exemplary automotive application

computing a simple mathematical function (e.g., the temperature sensor computes a ramp function with a value range from -30 to $+50$ degree centigrade).

As depicted in Figure 6.1, the exemplary automotive application is distributed over two SoCs, denoted *sensor SoC* and *display SoC*. Both SoCs host four jobs which are executed on their respective micro components. For the interconnection of the SoC components a gateway, which is implemented on a dedicated micro component, is deployed on each SoC. The chip external network is implemented by a TTE network [Kopetz et al., 2005].

The sensor SoC hosts a job that emulates the measurement of the vehicle dynamics. This job disseminates values for lateral and longitudinal acceleration, as well as, the yaw rate. Further on, this SoC hosts two replicated jobs that emulate a temperature sensor. Related thereto, a diagnostic unit is hosted on the SoC, which has the task to switch between the two replicated temperature sensors in order to demonstrate the usage of stand-by micro components. This is realized by a dedicated channel to the RMA.

The central job on the display SoC is the monitor job, which visualizes the temperature value received via the gateway and, depending on the currently active mode, either the data received from the navigation job or from the rear camera job. The selection of the active primary mode is performed via a user interface provided by the control job. For the (emulated) computation of the navigation information, the navigation job utilizes the exported sensor data from the sensor SoC.

Both SoC components, as well as, the chip-external network exhibit the same global time-base. This synchronization is performed by a dedicated *rate master*. Since the SoC-external network and the SoC internal networks are implemented as TTE networks (see the next section for details on the implementation of the SoC component), the rate master is responsible for generating appropriate *TTsync messages* [Kopetz et al., 2006, p. 35], which are disseminated in all three networks.

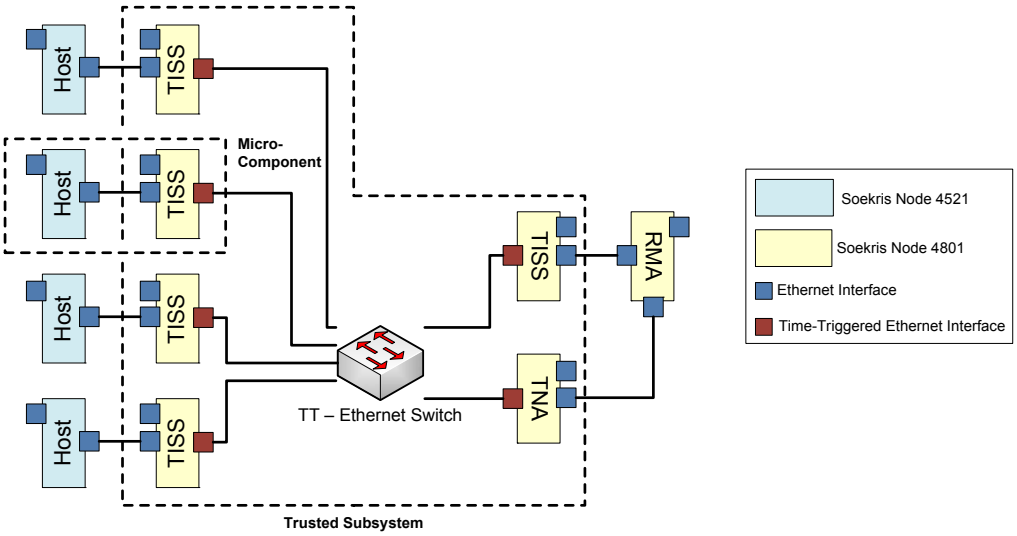


Figure 6.2: Setup of a DECOS SoC component

6.2 SoC Component Setup

The physical setup of the SoC components that have been realized in the course of the DECOS project is depicted in Figure 6.2. Each SoC component comprises four general purpose micro components, which are deployed for the execution of the application functionality. In addition, two dedicated micro components are realized on each SoC, which are utilized for the implementation of the gateway and the RMA.

Each micro component is implemented by two physically separated single board computers. One is dedicated to the implementation of the host, i.e., for the execution of the application software, while the other realizes the TISS and thereby establishes the link to the component-internal network. The interconnection of those single board computers is established via standard Ethernet interfaces using a point-to-point connection. Similarly, the data exchange between RMA and TNA is performed using a dedicated communication channel based on standard Ethernet.

The component-internal network is realized by a TTE network, which emulates a bus-based on-chip interconnect. The access to this network is realized by an *Field Programmable Gate Array* (FPGA)-based implementation of a TTE controller [Steinhammer, 2006]. The TTE network exhibits a star topology with the individual network segments interconnected by a dedicated TTE switch [Steinhammer et al., 2006]. All messages disseminated on the internal TTE network are transported to all TTE controllers connected to the switch. However, the actual reception of a message depends on the configuration of the MEDL at the respective TTE controller.

Host Computer.
The implementation of the host is performed on a single board compact com-

puter of type *Soekris Engineering net4521*, which incorporates a 133 MHz 468 class ElanSC 520 processor from AMD. This compact computer provides 64 MByte SDRAM main memory, uses a CompactFlash module for program and data storage, and is equipped with two 10/100 Mbit Ethernet ports. As the operating system we have deployed the real-time Linux variant *Real-Time Application Interface* (RTAI) [Mantegazza et al., 2000]. This real-time operating system is used to minimize the latency and the jitter of computational and communication activities. For the communication with the TISS an optimized version of the standard real-time Natsemi driver distributed by the RTnet open-source project [1] is used.

TISS.
The TISSs are implemented on similar single board compact computers, namely on Soekris Engineering net4801 boards. In contrast to the board described above, it incorporates a 266 MHz 586 class NSC SC1100 processor and provides 256 MByte SDRAM main memory. In addition to its three on-board 10/100 Mbit/s Ethernet ports, the execution platform of the TISSs is extended with PCMCIA-based implementation of the TTE controller. Like the host, the TISS uses the real-time Linux variant RTAI as its operating system.

Time-Triggered Ethernet Switch.
The TTE switch supports the predictable transmission of time-triggered messages, even when event-triggered and time-triggered messages are transmitted over the same Ethernet network [Steinhammer et al., 2006]. For this purpose, time-triggered TTE messages are identified by a dedicated type field in the message header, which triggers a specific treatment of those messages within the TTE switch. Unlike standard Ethernet frames which are buffered within the switch, TT-Ethernet frames are directly switched to their destination ports. This means, whenever a run-time conflict between event-triggered traffic and time-triggered traffic occurs, the TTE Switch preempts the processing of the event-triggered traffic and transmits the time-triggered message with a constant transmission delay. To achieve determinism of the transmission delay, it is important that each time-triggered frame takes the same route through the switch. Therefore, the FPGA design of TTE switch was optimized w.r.t. a symmetric layout of the data paths [Steinhammer et al., 2006]. Immediately after the transmission of the time-triggered message, any preempted event-triggered message is autonomously retransmitted.

TNA.
The TNA is realized on the same hardware platform as the TISSs, i.e., on Soekris Engineering net4801 compact computers extended by a PCMCIA card implementing a TT-Ethernet controller. Likewise, the TNA utilizes RTAI as its real-time operating system. However, it exploits the services provided by the *Linux Real-Time* (LXRT) extension of RTAI, which enables the development of real-time programs running in

[1] `http://www.rts.uni-hannover.de/rtnet/`

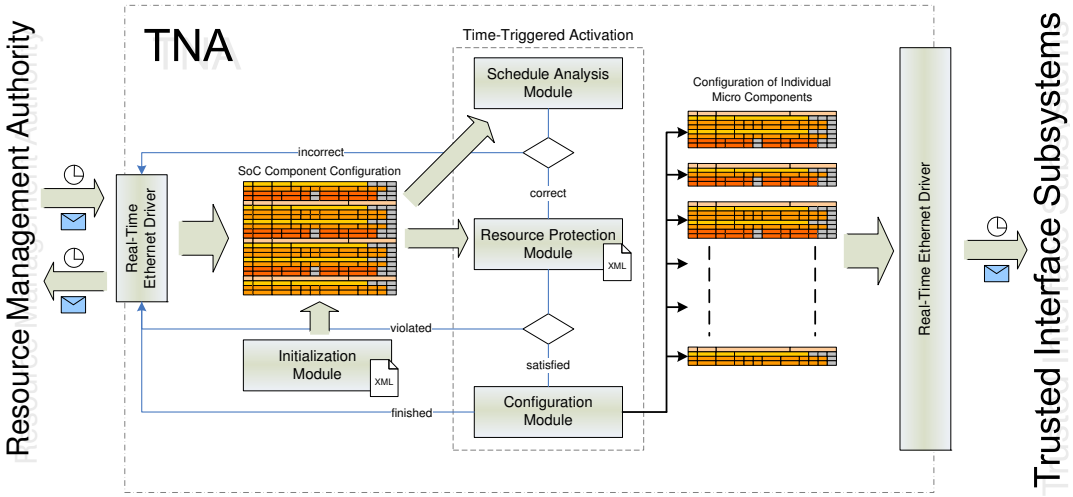


Figure 6.3: Flow diagram of the TNA software

user space. Typically, application software for real-time operating systems have to be realized as Linux kernel modules executed in the kernel space of the operating system; thus, circumventing the memory protection mechanisms of the operating system. This is prevented by using LXRT. The communication towards the TISSs is realized via TTE, while the interface towards the RMA is realized via a standard Ethernet connection using an a priori defined time-triggered communication schedule.

6.3 TNA Implementation

The services of the TNA are implemented as software modules on the above described Soekris embedded computing platform. The constituting software modules are depicted in Figure 6.3 (gray-shaded boxes represent individual software modules) and are explained in detail in the following subsections.

6.3.1 Initial TISS Configuration

After the start-up of the SoC component an initial system configuration is distributed to the TISSs of all micro components. This initial configuration is located in the local memory of the TNA and has to be generated during the integration of the individual DASs on the particular SoC, i.e., during the transformation of the UFIM to the PAM in the model-driven development process. This initial system configuration is static and requires no interaction between TNA and RMA. It comprises the following information:

- an initial configuration for each micro component regarding the parameterization of the core services provided by the TISS (e.g., watchdog period) and the host operation mode (e.g., standby or active mode),

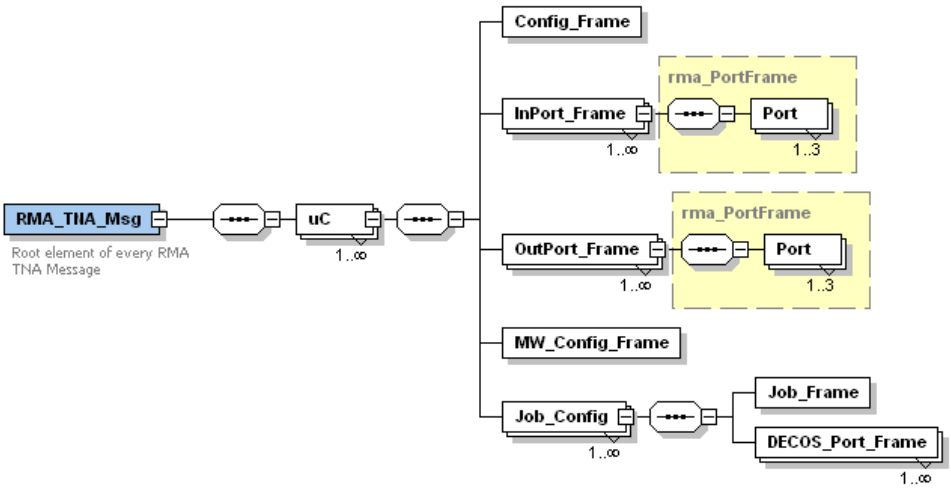


Figure 6.4: XSD schema defining the XML document structure for the specification of the initial SoC configuration

- an initial message schedule for those communication channels on SoC-internal TTE network which are required at the start-up of the system (e.g., the communication channels to the RMA for requesting additional resources), and

- an initial allocation of jobs to micro components, as well as, the UFIM-Port–to–SoC-Port mapping for the communication channels required for the start-up.

XML Representation of the Configuration Data

For describing the initial configuration of an SoC component, we have created an *XML Schema Definition* (XSD) document that facilitates the capturing of the information comprised in the RMA–to–TNA message. The content of this message is described in the appendix in Section A.2. The XSD schema is depicted in Figure 6.4. It enables the comfortable creation of XML files holding the initial system configuration and supports the modification of the initial system configuration without the need to change the implementation of the TNA. This means, the object code of the TNA can remain unchanged, regardless of the current configuration of the SoC component, i.e., regardless which DASs are hosted on the SoC component, as well as, the actual allocation of jobs to micro components.

At the start-up of the TNA, respectively the start-up of the entire SoC, the *Initialization Module* (cf. Figure 6.3) parses the XML file containing the initial system configuration. The name and location of the XML file is specified at the invocation of the TNA software, otherwise default values are taken. A partial example of such an initial system configuration file is presented in the appendix (see Section B.1 on page 141).

Realization of the Initialization Module

For the implementation of the Initialization Module of the TNA software, the *Expat XML Parser* in version 2.0 has been deployed. Expat[2] is a stream-oriented XML parser with a simple to use application interface that sets high standards for reliability, robustness, and correctness [Coope, 1999].

The operation of the Expat XML Parser is the following: The XML file is fed to the parser. Every time it catches an XML start or end tag, a callback or handler function that is registered to the respective event is called. The arguments passed to the callback function contain the value of the XML element, as well as, all discovered attribute/value pairs. For the implementation of the Initialization Module a handler function for processing the start tag of an XML element is defined, which generates a C data structure containing the entire configuration information of the SoC component. The format of this data structure is identical to the format of the data structure of the RMA-to-TNA message, which is described in Section A.2 in the appendix.

To check the initial configuration for its correctness, the same mechanisms as used for verifying the RMA–to–TNA configuration message are applied on the initial configuration. This involves the *Schedule Analysis Module* and the *Resource Protection Module* of the TNA software, which are explained in detail in Section 6.3.3 and Section 6.3.4.

6.3.2 RMA–TNA Communication

The SoC configuration proposal computed by the RMA is passed to the TNA via a standard Ethernet connection. However, the dissemination of the configuration message is periodic and strictly time-triggered. The reason for this design decision is to ensure that the TNA is not interrupted by the RMA and that the timing behavior of the TNA is independent of the RMA.

The communication between RMA and TNA is established using the *Real-Time Ethernet Driver Module* as depicted in Figure 6.3. This part of the TNA software is implemented as a kernel module of the Linux 2.6 kernel and directly accesses the so-called *skb* data structures, which are widely used in the Linux community for the implementation of Ethernet network drivers [Rubini and Corbet, 1998, p. 452]. In our particular case, it is the task of the Real-Time Ethernet Driver Module to extract the data of the received Ethernet message and build the data structure as defined for the RMA–to–TNA message within a shared memory region that is accessible by the *Schedule Analysis Module*, the *Resource Protection Module*, and the *Configuration Module* for further processing.

[2] available at `http://expat.sourceforge.net/`

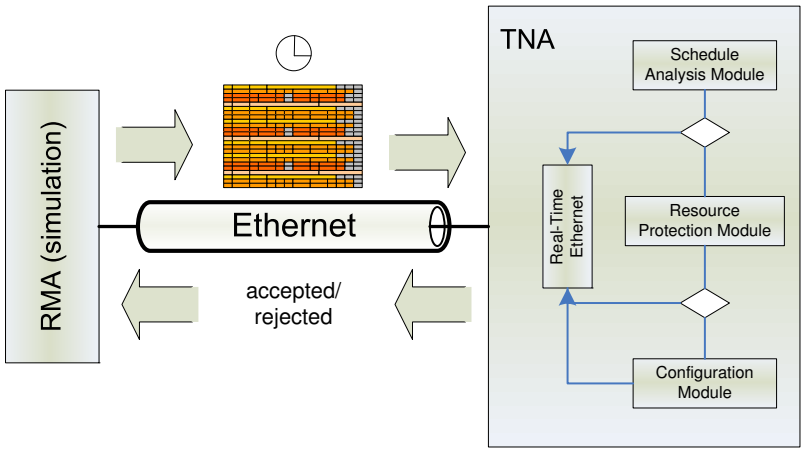


Figure 6.5: Flow diagram of RMA–to–TNA data exchange

RMA–TNA Data Exchange

The RMA disseminates new configuration information in a sporadic time-triggered message. This means the points in time at which the RMA is permitted to transfer a message to the TNA are defined a priori, but an actual transmission occurs only, if the RMA has been triggered to compute a new configuration due to a requested change in the resource allocation. For this prototype implementation we have defined a period of 62,5 ms for the dissemination of the RMA–to–TNA messages.

Figure 6.5 depicts the flow diagram of a data exchange between RMA and TNA. As soon as the TNA has finished its analysis of the configuration data, a response is transmitted back to RMA. With this response, the TNA informs the RMA whether it has accepted or rejected the new configuration proposal. The RMA is responsible for forwarding this information to the jobs, respectively, the DASs that have requested the resource reallocation or are affected by an updated configuration.

6.3.3 Schedule Analysis

As depicted in Figure 6.3, the first software module that operates on the received configuration information is the *Schedule Analysis Module*. The purpose of this module is to analyze the message schedule contained in the configuration information for its correctness. This means it verifies that all messages disseminated over the same communication channel (regardless of their periods) are free of collisions. For the bus-based implementation this implies that the time spans between start of transmission and end of transmission of any two messages must not overlap.

For the analysis algorithm we have chosen the following model for representing the temporal properties of messages and the respective communication channel (cf. Figure 6.6). A communication channel is characterized by its communication topology, bandwidth, and required *Inter Frame Gap* (IFG).

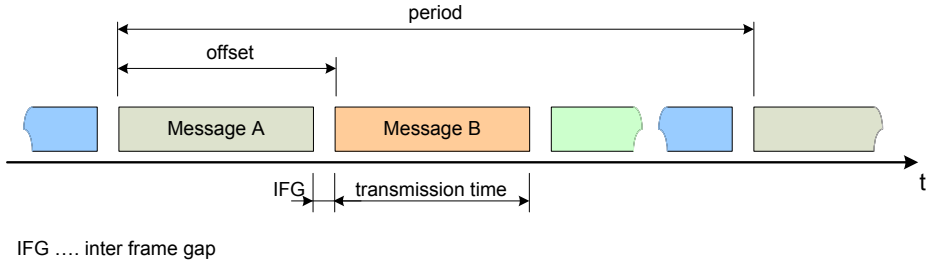


Figure 6.6: Used model of communication channel and messages

Topology: The algorithm is based on the assumption that the communication channel is realized as a single bus with broadcast topology, using *Time Division Multiple Access* (TDMA) for bus arbitration. Hence, no message transmissions are allowed to occur simultaneously.

Bandwidth: The bandwidth determines the amount of data per time unit that can be transmitted over the channel. For the scheduling analysis algorithm we assume a bandwidth of 100 Mbit/s, which results in 80 ns for the transmission of a single byte.

Inter Frame Gap: The IFG determines the minimal time between the transmissions of two consecutive messages. The adherence of the message schedule to this time constraint has to be evaluated by the analysis algorithm.

A message in our model is characterized by its periodic send instant described by period and phase offset, as well as, by its required transmission time, which depends on the message length. Since the realization of our prototype implementation deploys TTE for the interconnection of the individual micro components, the minimal message length—according to the TTE specification [Kopetz et al., 2006, p. 34f]—is 72 bytes (Ethernet header and trailer, TTE header, and minimal TTE payload). The configuration data describes only the length of payload data, i.e., the actual message stored in the ports of the TISS towards the on-chip network. Thus, the analysis algorithm extends the message size with the amount of bytes required for transmitting the Ethernet header and trailer, the TTE headers, and the difference of the actual payload to the specified minimal payload in TTE, which are 34 bytes.

Detection of Message Collisions

The primary task of the Schedule Analysis Module is to detect whether two messages that are specified in the configuration information collide on the shared communication medium. In a first step, the analysis algorithm checks messages of the same period. In other words, given an ascending sorted list of messages M of the same period P where the sorting criterion is the offset of the message, the analysis algorithm checks whether the following two conditions hold.

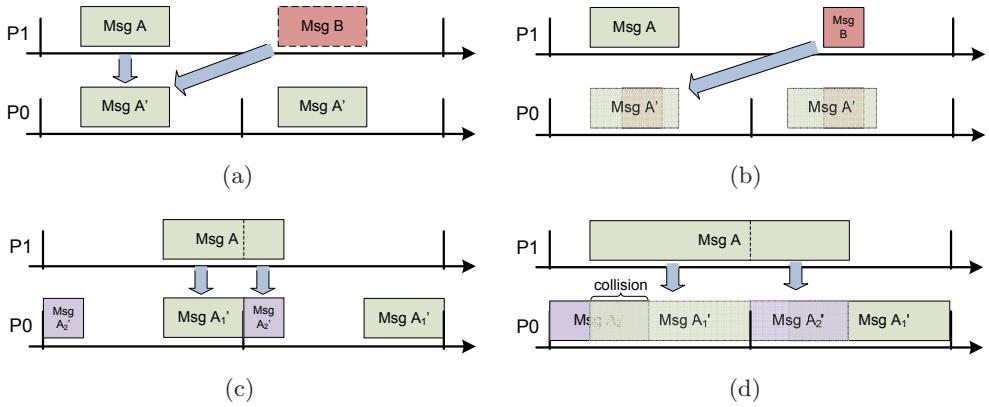


Figure 6.7: Transformation of messages to the next smaller period

$$\forall m \in M: \quad m_i.offset + m_i.transmission\_time + IFG < m_{i+1}.offset \qquad (6.1)$$
$$\text{and} \quad 0 \leq i \leq |M| - 2$$

This condition states that for all messages of the same period the instant at which the transmission of a new message is started has to be later than the time instant resulting from transmission of the previous message plus the demanded idle time of the to bus (the IFG).

$$\forall m \in M: \quad m_i.\text{offset} + m_i.\text{transmission\_time} + IFG < m_0.\text{offset} + P \qquad (6.2)$$
$$and \quad i = |M| - 1$$

The second condition states that the last message in the ordered list of Messages M does not interfere with the first one in the list, which is the first message transmitted in the following period.

However, the detection of collisions of messages of the same period is only a first step. It has also to be ensured that messages of different periods do not collide with each other. For this, a specific property of the time format of the on-chip network is exploited, namely that all periods are negative powers of two of one second. Thereby, all periods are whole-number multiples of each other. This strongly facilitates the transformation of the messages of one period to the next lower one, which is mandatory for detecting message collisions across different periods. Four possible outcomes of such a transformation are depicted in Figure 6.7.

The simplest case for a transformation of a message of given period P_1 to the next lower period P_0 (which is exactly the half of P_1) is shown in Figure 6.7(a). A message comprising an offset lower than the length of period P_0 and a transmission duration that finishes before the end of period P_0 (as it is the case for message Msg_A

in Figure 6.7(a)) is transformed directly from period P_1 to period P_0. This means offset and length of the message remain unchanged, the period is changed to P_0. Please note that these message transformations are performed on temporary copies of the message specifications, because it is obvious that the temporal characteristics of a message are changed (e.g., the required bandwidth for a message transformed to the next lower period is doubled).

If the offset of a message of P_1 is greater than the length of period P_0 (as depicted by Msg_B in Figure 6.7(a)), the offset has to be transformed. This is realized by reducing the offset by the length of P_0. After performing the transformation, there is no difference between message Msg_A and message Msg_B. Thus, this is an example which shows that by performing the transformation of messages from P_1 to P_0, the number of messages that has to be analyzed could be reduced, which would speed up the analysis process.

A further example demonstrating this reduction is presented in Figure 6.7(b). The transformation of Msg_A and Msg_B from P_1 to P_0 would lead to two messages (Msg'_A and Msg'_B) with different offsets and lengths. However, since Msg'_B is fully enclosed by Msg'_A, only Msg'_A is important for the collision detection algorithm and Msg'_B can be removed from the list. In the general case, for two messages which are non-colliding in P_1, but overlap in P_0 (as it is shown in Figure 6.7(b)), only the union of the transmission intervals of both messages is of interest for further analysis.

As shown so far, it is possible that after the message transformation several messages of P_1 are collapsed to a single message of P_0. However, it is also a reasonable case that one particular message of P_1 has to be split up into two messages of P_0 as it is shown in Figure 6.7(c). This case occurs when the message in P_1 has an offset that is lower than the length of P_0, but the transmission of the message exceeds the end of period P_0.

In scenarios such as the one depicted in Figure 6.7(d), the transformation directly delivers information on message collisions. In the scenario depicted here, the length of message Msg_A is greater than the length of the entire period P_0. Thus, it is not possible to find any time slot with the periodicity of P_0 (or smaller) that would not collide with message Msg_A. In other words, due to the existence of the long Msg_A it is not possible to find a conflict free schedule of messages with periods lower or equal than P_0.

A practical example for such a case is the dissemination of a full-length (i.e., 1500 bytes) Ethernet message on a 100 Mbit/s network. The transmission time of such a message over the network equals approximately 123 μs. Thus, all periods exhibiting a length lower than 123 μs cannot be used for message transmission at the presence of at least one single full-length Ethernet message. This means for TTE that the two smallest periods, which have a duration of 122 μs and 61 μs, respectively, cannot be used at the presence of at least one full-length Ethernet message.

Listing 6.1: Pseudo-code of the algorithm AnalyzeMEDL

```
Algorithm: analyzeMEDL
Input: Configuration data structure (incl. MEDL) from RMA
Return: Number of message conflicts detected

1: generate sorted lists M_i of messages (with P_min <= i <= P_max);
2: for (period = P_max; period >= P_min; period--) do
3:  conflicts += checkMessageSchedule(M_period);
4:  if (period > P_min) then
5:    transformAndStoreMessages(period, M_tmp);
6:    insertMessagesToList(M_tmp, period-1);
7: return conflicts;
```

Analysis Algorithm

In this section we elaborate on the message schedule analysis algorithm. It exploits both methods described in the previous section for detecting message collisions contained in the configuration proposal computed by the RMA.

Listing 6.1 depicts the general outline of the algorithm. The algorithm takes as its input the configuration data structure provided by the RMA. In this data structure the description of the message schedule (i.e., the MEDL) is included. The return value of this algorithm is the number of detected conflicts, which is exploited by the TNA software to decide whether the schedule is correct or not.

The first step that is performed by the algorithm is to extract the MEDL of all micro components out of the entire configuration data structure and to store the messages in ascending sorted lists respective to their period (cf. line 1). For each period a dedicated list is generated. In the prototype implementation we have 16 periods from 2^0 to 2^{-15} seconds; thus, 16 lists are generated.

In section 6.3.3 two conditions are presented that are exploited to detect collisions between any two messages of the same period. In order to verify that also the messages of different periods are free of collisions, the lines 2–6 are executed by the algorithm. Starting with the list of the largest period, all lists are consecutively checked for message collisions. Therefore, the function *checkMessageSchedule(M_period)* is called. This function inspects the message specification contained in M_{period} for any violations of the two conditions specified in Section 6.3.3. The return value of this function denotes the number of message conflicts, i.e., message collisions within the schedule, found in the message list of the respective period. If the currently analyzed period is equal to P_{min}, the algorithm exits the loop and returns the cumulative number of detected message conflicts.

Otherwise, all messages are transformed to the next lower period as explained in the previous section by the function *transformAndStoreMessages(period, M_tmp)* (cf. line 5). During this transformation redundant messages (e.g., like in scenarios as depicted in Figure 6.7(a,b)) are detected and removed. The transformed messages of the current period are stored in a temporal message list M_{tmp}. Afterwards, all

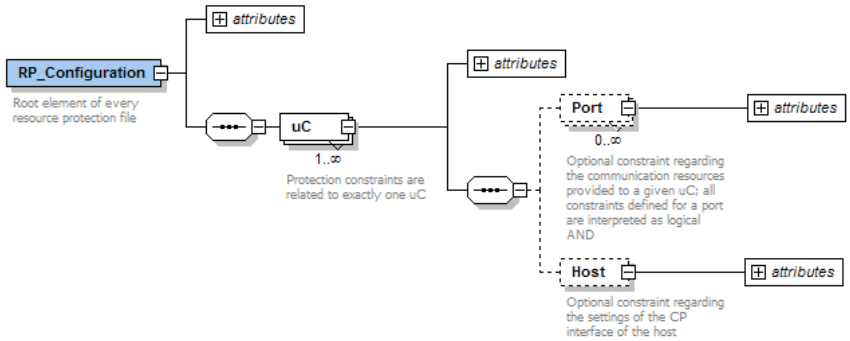


Figure 6.8: XSD schema defining the XML document structure for the specification of protected resources

messages of M_{tmp} are inserted into the sorted message list of the next lower period and the collision detection is continued with this period.

As it is depicted in Figure 6.3, the outcome of the Schedule Analysis Module is utilized to decide, whether the schedule is correct and further analysis (i.e., the resource protection) has to be performed, or the analysis is aborted and the RMA is informed. In case message collisions are found, the configuration data of the individual micro components remains unchanged.

6.3.4 Resource Protection

For the SoC component implementation in the course of this case study it is possible to specify off-line dedicated sets of resources, which are statically assigned to a given micro component and must not be reassigned during the operation of the SoC component. This can be used, for instance for safety-critical DASs in order to statically reserve the required communication bandwidth on the SoC-internal network. These statically assigned resources are protected by the TNA in order to avoid that a software fault within the RMA can invalidate these allocations. The protection is realized by the specification of assertions on the values of the configuration parameters (cf. Section 5.4.1).

For the case study implementation, the protection of the resources includes the configuration of the input and output ports, as well as, the host operation mode and the watchdog period. The following subsections elaborate on the implementation of this resource protection functionality.

XML Representation of Protected Resources

In order to ease the specification of those assertions, an XML representation has been devised. An overview of the XML schema that defines the document structure for XML files comprising valid specifications for protected resources is depicted in Figure 6.8. A partial example of such a specification is shown in the appendix in

Listing 6.2: Pseudo-code of the algorithm checkConstraints

```
Algorithm: checkConstraints
Input:  SoC Component Configuration from RMA (rma_tna_msg_t),
        List of Resource Protection Constraints for Ports (rp_port_t),
        List of Resource Protection Constraints for Hosts (rp_host_t)
Return: true if all constraints are fulfilled, false otherwise

 1:  for (uC = 0; uC < NUMBER_OF_uC; uC++) do
 2:    if (host resource_protection == enabled) then
 3:      if (host_sysmode_protection == enabled) then
 4:        return checkAndReturnOnViolationHostSysMode(uC);
 5:      if (wd_period_protection == enabled) then
 6:        return checkAndReturnOnViolationWDPeriod(uC);
 7:    if (input_port_resource_protection == enabled) then
 8:      while (not end_of_input_port_list) do
 9:        return checkAndReturnOnViolationPeriod(uC, port);
10:        return checkAndReturnOnViolationLength(uC, port);
11:        if (phase_offset_protection == enabled) then
12:          return checkAndReturnOnViolationPhaseOffset(uC, port);
13:        if (type_protection == enabled) then
14:          return checkAndReturnOnViolationType(uC, port);
15:        if (channel_protection == enabled) then
16:          return checkAndReturnOnViolationChannel(uC, port);
17:    if (output_port_resource_protection == enabled) then
18:      repeat lines 8 to 16 with output ports
19: return true;
```

Listing B.2. The XML file containing the specification of the resources that are to be protected by the TNA is parsed only once, at the start-up of the TNA software. If no file name and file location for that XML file is specified at the invocation of the TNA, default values are taken.

Realization of the Resource Protection Module

Similar to the XML parser described in Section 6.3.1, the *Resource Protection Module* deploys the Expat XML Parser in version 2.0 for extracting the information out of the XML file. The monitoring of the constraints is performed by the Resource Protection Module with the algorithm described in Listing 6.2. However, as depicted in Figure 6.3 this module is only activated if no violations of the message schedule have been detected by the Schedule Analysis Module before.

The algorithm *checkConstraints* takes as input the entire configuration information of the SoC as it has been received from the RMA, as well as, the resource protection constraints for host and port configuration, i.e., the MEDL, generated by the Expat XML parser. The output of the algorithm is a binary value that indicates whether all checks have been passed (return value true) or not (return value false).

For the analysis process the algorithm traverses all assertions using the micro component identifier as index to the data structures. For each micro component it is first evaluated whether the protection for the host configuration and respectively the port configuration is activated. This is done by analyzing the *resource protection field* (cf. Section 5.4.1) of the individual constraints within an assertion.

The first detection of a violation of a protected resource causes the algorithm to abort with the return value false. If the Resource Protection Module detects a violation of any of the specified assertions, the RMA is informed that an invalid resource configuration has been received. In this case, the actual configuration of all micro components remains unchanged. Otherwise, the configuration data is considered to be correct and the *Configuration Module* is activated, which is responsible for disseminating the new configuration to the individual micro components via the CP interface of the TISSs.

6.3.5 Micro Component Configuration

The TNA periodically updates the configuration of the micro components by the dissemination of periodic time-triggered messages over the internal TTE network. The update of the configuration information is in the responsibility of the *Configuration Module*. The Configuration Module serves two purposes: On the one hand, it processes the data structure and separates the configuration information of the individual micro components. On the other hand, the Configuration Module updates the respective memory regions of the TTE controller with the extracted information, i.e., it accesses the CNI memory representing the ports of the TNA towards the TISSs.

At the start-up of the TNA software, the CNI memory of the TTE controller, which is physically located on the PCMCIA card realizing the TTE controller is mapped into the main memory of the Soekris embedded node computer. The purpose of this is to enable the Configuration Module to directly access the CNI memory of the TTE controller. After the configuration data is updated, the TNA informs the RMA via the Real-Time Ethernet Module that the configuration message has been accepted and the configuration data for each micro component has been successfully updated. The actual dissemination of the configuration data from the TNA to the CP interface of the TISSs is done autonomously by the TTE controller based on the (temporal) specification of the output ports of the TNA.

6.4 RMA Implementation

We demonstrate in this case study the dynamic resource management w.r.t. communication resources. For this purpose the implementation of the RMA realizes the following functions (each of them implemented as distinct software modules as depicted in Figure 6.9): (i) reception and processing of resource requests from jobs, (ii) scheduling and allocation of the available resources, (iii) initiation of schedule verification by establishing the RMA–to–TNA communication, and (iv) preparation of configuration data structures for the affected jobs. The software modules identified in Figure 6.9 are described in more detail in the following sections (except the module *TNA communication* which is realized identically to the module described in Section 6.3.2).

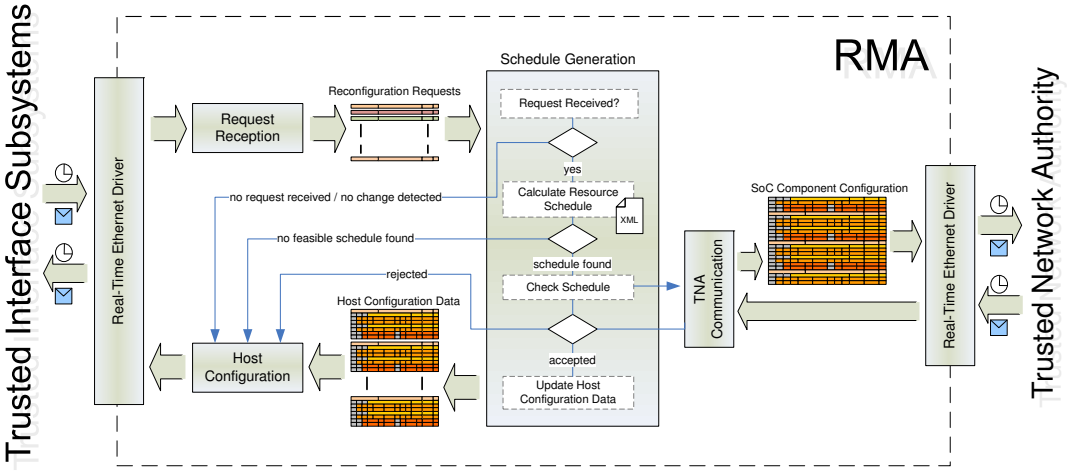


Figure 6.9: Flow diagram of the RMA software

6.4.1 Job–to–RMA Communication - Request Reception

Each micro component hosting a job that is permitted to disseminate resource requests to the RMA is provided with a static SoC-Channel to the RMA. The temporal characteristics of this channel are exactly defined by the periodicity of the messages transmitted over this channel (which is equal to the reconfiguration period introduced in Section 5.2.2), their phase offset, and the maximum amount of data disseminated in a single message (i.e., the message length). The static communication channels are implemented by sending sporadic time-triggered messages, in order to ensure that messages are only sent if an updated resource request is inserted in the respective SoC-Port of the micro component.

The communication service provided by the TISS of the RMA is configured to notify the RMA after the reception of a request message over one of these static communication channels. This notification mechanism is implemented in the prototype implementation via an Ethernet message that is sent from the TISS to the RMA over the point-to-point Ethernet connection between TISS and RMA. After the reception of the last request message, the RMA schedule generation module is executed (cf. Figure 6.9).

In the case study, a resource request is indicated by *mode request* message that is disseminated from a job to the RMA. Note, that only a dedicated set of jobs is permitted to submit these messages, namely only those jobs that are assigned with a static channel to the RMA. The structure of a mode request message is depicted in the following.

```
struct mode_req_msg_t {
  unsigned char mode;
}__attribute__((__packed__));
```

The mode in the mode request message refers to a dedicated primary mode of the entire DAS. A primary mode reflects a different behavior of a DAS's service,

e.g., a display changing its behavior with respect to the information it has to show. The available set of primary modes of a particular DAS have to be defined at design time of the DAS. The specification of the modes comprises all information about the resource requirements of the entire DAS in the given mode that is required for generating the resource schedule in the RMA. Therefore, these specifications include for every DAS and every primary mode:

- a list of all jobs which are part of the respective DAS,
- the allocation of the jobs to micro components,
- the parameterization data of the core services that are provided by the TISSs (e.g., watchdog period) and the host operation mode (e.g., standby or active mode), and
- a list of communication channels that are active in the respective mode including the specification of the communication topology for each channel.

According to these a priori defined primary modes for each DAS, the RMA checks the received mode requests for their validity (e.g., whether the requested mode is defined at all) and extracts thereof the newly requested resource demands for the individual DASs as input for the resource schedule generation.

6.4.2 Resource Schedule Generation

The central task of the RMA is the generation of a feasible resource schedule for the requested modes of the individual DASs. The pseudo code depicted in Listing 6.3 shows a high-level view of the procedure, which is executed every reconfiguration period for resource allocation. The procedure assumes that for every DAS that is realized on the SoC at most one job is permitted to request mode changes from the RMA.

After the activation of the RMA task by an interrupt message received from the TISS, a first check is performed whether any mode changes have been requested (cf. line 1 in Listing 6.3). If not, the current configuration is kept and the procedure is exited. In this case, the current configuration is disseminated to all jobs at the next dissemination instant of the RMA–to–job messages.

If at least one mode change has been requested, a backup of the current configuration is generated (cf. line 2) and for each DAS for which a mode change has been requested, the list of active UFIM-Channels is updated according to the specified primary mode of the respective DASs (cf. lines 3 to 4). With this updated description of the required resources, the scheduling of resources is performed (cf. line 5). A pseudo code representation of the scheduling heuristic is shown in Listing 6.4.

On a successful generation of the resource schedule, the new configuration is transformed to the message format as required for the RMA–to–TNA communication (cf. Figure A.2 in the appendix) and is transmitted to the TNA for verification

Listing 6.3: Pseudo-code of procedure manageResources

```
Procedure: manageResources
Input:  requested modes m^req ∈ M^req, current modes m^cur ∈ M^cur,
        required resources R^cur, current configuration C^cur
Output: updated micro component configuration

1:  if (∄i : m_i^req ≠ m_i^cur for 0 ≤ i ≤ N_DASs) then return;
2:  backup(C^cur);
3:  for (i = 0; i ≤ N_DAS; i++) do
4:      if (m_i^req ≠ m_i^cur) then updateChannelList(i, m_i^req, C^cur);
5:  ret = scheduleResources(C^cur, C^cur.C^comm.first, P_inv, PH_inv, 0, P_max);
6:  if (ret == SUCCESS) then
7:      msg^RMA-TNA = createRMAtoTNAmessage(C^cur);
8:      sendRMAtoTNAmessage(msg^RMA-TNA);
9:      tnaResponse = waitForTNAResponse();
10:     if (tnaResponse == ACCEPTED) then
11:         msg^RMA-job = createRMAtoJobMessage(C^cur);
12:         updateCNI(msg^RMA-job);
13:     else
14:         restore(C^cur);
15: else
16:     restore(C^cur);
17: return;
```

(cf. lines 7 to 9). If no feasible schedule could be found, the currently active configuration is restored.

In case the TNA accepts the proposed resource schedule, the RMA creates an update of the job configuration and stores this configuration in the CNI of the TISS connected to the RMA (cf. lines 10 to 12). Otherwise, the resource schedule is discarded and the currently active configuration is restored.

The pseudo code depicted in Listing 6.4 represents the scheduling heuristic that is deployed for allocating send slots of the time-triggered NoC to UFIM-Channels. The algorithm *scheduleResources* is realized as recursive algorithm. First, it checks whether the phase offset of the currently analyzed UFIM-Channel is set to the constant value `NOT_SCHEDULED`. By convention, an invalid phase offset (as for instant represented by the constant `NOT_SCHEDULED`) in the specification of the UFIM-Channel denotes that the UFIM-Channel should be scheduled by the RMA. A valid phase offset on the other hand identifies a static configuration that must not be changed by the RMA, e.g., for realizing phase aligned communication in control loops. If such a static configuration is found, the algorithm is recursively called with the next item in the list of UFIM-Channels C^{comm}. If no further item exists, `SUCCESS` is returned since all available channels have been successfully scheduled so far.

As a next step, the algorithm performs a message schedulability test (cf. lines 4 to 6) in order to discover UFIM-Channel characteristics that preclude the generation of a correct schedule. This check is concerned with the maximum message length that is specified for the actually analyzed communication channel. If the time to transmit the message, plus the *Inter Frame Gap* (IFG), exceeds the duration of the smallest period P^{min} that has been found in the configuration so far, this message

Listing 6.4: Pseudo-code of algorithm scheduleResources

```
Algorithm: scheduleResources
Input:  current configuration C^cur (including a sorted list of communication
        channels C^comm; input is modified by the algorithm),
        UFIM–Channel list iterator \textit{channel}, last scheduled
        period P^last, last used phase PH^last, length of largest
        message l^max, and shortest period P^min
Return: SUCCESS if a schedule has been found, FAILED otherwise

1:  if (channel.phase == NOT_SCHEDULED) then
2:    if (channel == C^comm.end) then return SUCCESS;
3:    else return scheduleResources(C^cur, channel.next, P_inv, PH_inv,
                                    l^min, P^min);
4:  if (channel.period < P^min) then P^min = channel.period;
5:  if (channel.msgLen > l^max) then l^max = channel.msgLen;
6:  if (l^max + IFG > length(P^min)) then return FAILED;
7:  channel.phase = getFirstValidPhase(channel.msgLen, channel.Period,
                                       P^last, PH^last);
8:  collision = detectCollision(C^cur);
9:  while (collision) do
10:   channel.phase = getNextPossiblePhase(channel.phase);
11:   if (channel.phase < 0) return FAILED;
12:   else collision = detectCollision(C^cur);
13: if (channel == C^comm.end) then return SUCCESS;
14: return scheduleResources(C^cur, channel.next, channel.period,
                             channel.phase, l^max, P^min);
```

will collide in any case with every message that is disseminated with period P^{min}. Thus, the algorithm returns `FAILED` to its caller.

Only if both checks are passed, the actual message placement heuristic starts. It is the strategy of the heuristic to increase the probability to find a feasible placement for subsequent messages by generating a dense placement of messages within the same period, which leaves free sending slots for messages with smaller periods. Therefore, message lengths and their periods are taken into account. The available sending slots are used in a reverse order, i.e., starting with larger phase offsets at the beginning.

The function *getFirstValidPhase(...)* returns a first guess for a feasible placement of the actual message by exploiting the fact that the list of communication channels is decreasingly sorted according to their period duration. If *channel.period* equals to the period of the previously scheduled message P^{last}, then *getFirstValidPhase(...)* returns the phase offset of the previously scheduled message PH^{last} reduced by the amount of time required for the transmission of *channel.msgLen* bytes. Otherwise, the largest phase offset that permits the transmission of the message before the end of the period *channel.period* is chosen as a first guess.

The currently created message placement is evaluated by the function *detectCollision(...)* (cf. line 8). This function is based on the same algorithm for detecting schedule violations as the one realized within the TNA for the verification of the schedule. This algorithm is described in Section 6.3.3. As long as a collision is detected, a new placement is calculated by the function *getNextPossiblePhase(...)*. In essence, this function reduces the current phase offset by the time required for

transmitting the message.

If the calculated phase becomes negative (which indicates an invalid value), the *scheduleResources* algorithm returns `FAILED`, since no valid sending slot could have been found for the actual message. If the call of *detectCollision(...)* in line 12 of the depicted pseudo-code in Listing 6.4 detects no message collision within the current schedule, the loop is exited. If the currently scheduled message was the last one to be scheduled, `SUCCESS` is returned. Otherwise, *scheduleResources(...)* is recursively called with the next communication channel in the list.

6.4.3 RMA–to–Host Communication

After a positive response of the TNA, the last activity that has to be performed by the RMA is to update the host configuration data structure in the CNI of the RMA's TISS. The configuration data is then autonomously transmitted to the TISSs of all other micro components according to the static RMA–to–host communication channels (cf. phase 6 in Figure 5.3) in order to inform all jobs about the resource reallocation that has been carried out.

As depicted in Figure 5.3, the configuration data is transferred from the RMA to the hosts once every reconfiguration period. These periodic configuration messages carry state information describing the host configuration. This facilitates the reintegration of hosts, which, for instance, have been restarted by the TISS due to the omission of an update of the local watchdog. The C structure that realizes the interface of the RMA for the host configuration is depicted in the following listing.

```
struct rma_host_msg_t {
  unsigned short das_id;
  unsigned short job_id;
  uint8_t mode_id;
  uint8_t input_UFIMport_SoCport_mapping[MAX_UFIMPORTS];
  uint8_t output_UFIMport_SoCport_mapping[MAX_UFIMPORTS];
}__attribute__((__packed__));
```

The data structure rma_host_msg_t is subdivided into five member elements. The element *das_id* (2 bytes) stores the system-wide unique identifier of the DAS, to which the job identified by *job_id* (2 bytes) belongs to. In our prototype implementation, a host executes a single job. However, due to mode change requests, the RMA can reallocate jobs to different hosts over time. The job, which has to be executed by the host after the next reconfiguration instant (cf. phase 7 in Figure 5.3), is determined by those two identifiers. The third identifier *mode_id* (1 byte) determines the mode of operation of the job, i.e., it corresponds to a concrete degradation level for the requested primary mode selected by the RMA.

As described in Section 3.3 data exchange between jobs is realized via so-called UFIM-Ports. For the physical realization of those logical communication channels, the UFIM-Ports have to be mapped onto SoC-Ports, which form the connection point of a micro component to the NoC. Since the mapping between

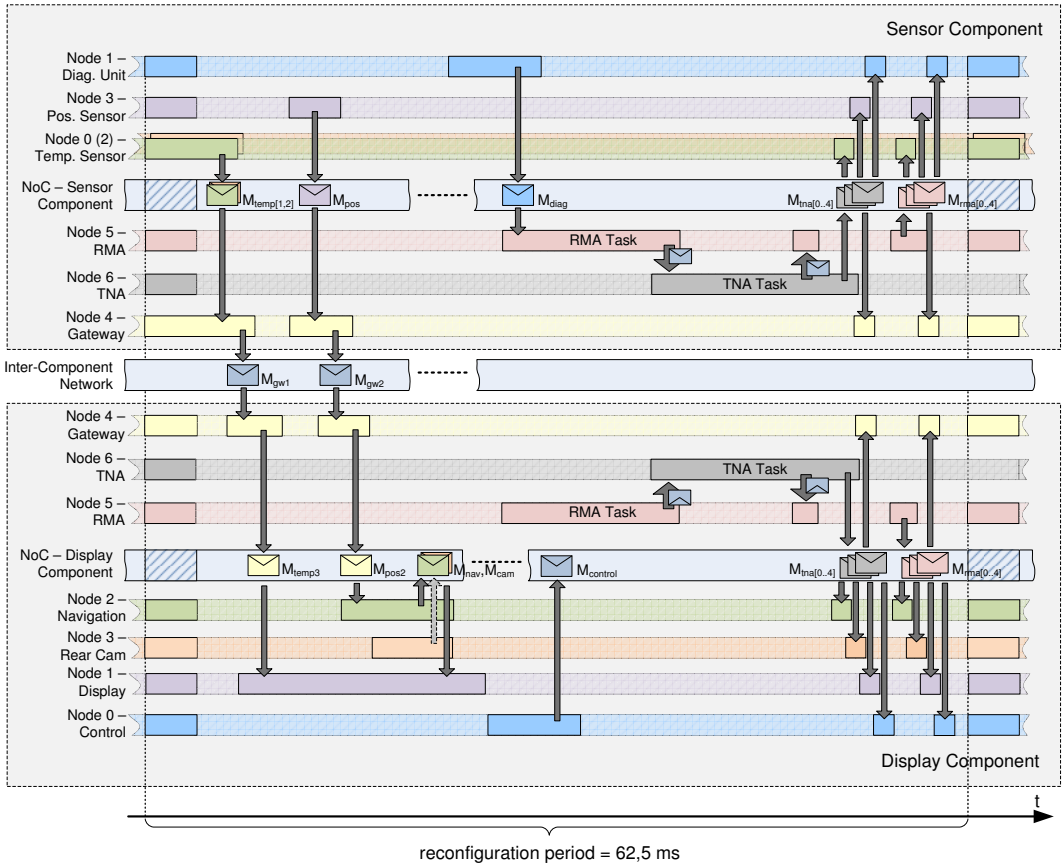


Figure 6.10: Message schedule for the exemplary automotive application

UFIM-Ports and SoC-Ports is dynamically calculated by the RMA according to the active communication channels in the currently selected primary mode, the mapping of input UFIM-Ports to input SoC-Ports *input_UFIMport_SoCport_mapping* (`MAX_UFIMPORTS` bytes) and the mapping of output UFIM-Ports to output SoC-Ports *output_UFIMport_SoCport_mapping* (`MAX_UFIMPORTS` bytes) is also part of the host configuration message.

The actual reconfiguration of the hosts and of the TISSs is performed by all micro components at the same instant of the global time, the SoC-wide consistent reconfiguration instant.

6.4.4 Exemplary Message Schedule

This section describes a message schedule for the exemplary automotive application that has been introduced in Section 6.1. The timing diagram illustrated in Figure 6.10 depicts the schedule of all messages as it has been generated by the RMA. In the top half of this figure, the messages transmitted over the SoC-internal network of the Sensor Component are depicted. The bottom half represents the schedule of the inter-component network of the Display Component. In between, Figure 6.10

Name	Sender	Receiver	Len	Phase	M	Description
M_{temp1}	Sensor-Node0	Sensor-Node4	64	198	0	Temperature value from primary temperature sensor
M_{temp2}	Sensor-Node2	Sensor-Node4	64	198	1	Temperature value from secondary (standby) temperature sensor
M_{pos}	Sensor-Node3	Sensor-Node4	64	462	0,1	Sensor values for longitudinal, lateral, and yaw position value
M_{diag}	Sensor-Node1	Sensor-Node5	64	2500	0,1	Diagnostic unit can initiate with this message a mode change ($0 \rightleftharpoons 1$)
$M_{tna[0..4]}$	TNA	Sensor-Node[0..4], Display-Node[0..4]	512	3870, 3873, 3876, 3879, 3882	0,1	Configuration information for all TISSs; for simplicity, both SoC components use the same schedule for those messages
$M_{rma[0..4]}$	RMA	Sensor-Node[1..4], Display-Node[1..4]	256	3890, 3892, 3894, 3896, 3898	0,1	Configuration information for all hosts; for simplicity, both SoC components use the same schedule for those messages
M_{gw1}	Sensor-Node4	Display-Node4	64	384	0,1	Gateway-to-gateway message containing the temperature value
M_{gw2}	Sensor-Node4	Display-Node4	64	640	0,1	Gateway-to-gateway message containing the lateral and longitudinal value
M_{temp3}	Display-Node4	Display-Node1	64	330	0,1	Temperature value forwarded to monitor node
M_{pos4}	Display-Node4	Display-Node2	64	594	0,1	Lateral and longitudinal value forwarded to navigation node
M_{nav}	Display-Node2	Display-Node1	256	726	0	Navigation information to be displayed at the monitor node
M_{cam}	Display-Node3	Display-Node1	256	726	1	Rear cam data to be displayed at the monitor node
$M_{control}$	Display-Node0	Display-Node5	64	2500	0,1	Control node initiates with this mes-sage a mode change ($0 \rightleftharpoons 1$)

Table 6.1: Message specification for the automotive example

depicts the message exchange over the TTE network interconnecting the two SoC-components. Please note that the generation of a schedule for those two messages is not in the scope of the RMA. These messages have been statically defined for both SoC components (as output messages of the gateway in the Sensor Component and as input messages to the gateway in the Display Component) and are part of the MEDL of the TTE controllers towards the component-external network.

A description of the individual messages required for the realization of the exemplary automotive application is given in Table 6.1. For each message the following

properties are listed:

- the communication topology by specifying the sender micro component and the list of receiver micro components,
- the temporal characteristics by specifying the message phase (the message period of all messages is 62.5 ms) and its length in bytes,
- the primary mode for which the respective message is active, i.e., for which it is part of the schedule, and
- a description of the purpose of the message

As it can be seen from Table 6.1, M_{temp1} and M_{temp2} use the same TDMA slot on the component internal TTE network of SoC component realizing the Sensor Component. However, at any primary mode only one of those messages is active. Thus, it is not allowed that in any configuration computed by the RMA both messages are active at the same time. A switch between those two mutual exclusive configurations is initiated by the diagnostic unit.

A similar situation applies to the messages M_{nav} and M_{cam} in the Display Component. While the mode change in the Sensor Component does not affect the semantics of the message, the mode change in the Display SoC component completely changes the semantics of the message received by the micro component hosting the monitor job. Thus, the monitor job in the Display Component has to change its behavior at a mode change initiated by the control job, while the gateway in the Sensor Component, which forwards the temperature values, has not to be aware of the mode change initiated by the diagnostic unit.

Chapter 7

Evaluation and Results

In the previous chapters we have identified and specified the services for resource management provided by the cooperation of *Resource Management Authority* (RMA) and *Trusted Network Authority* (TNA). Furthermore, an implementation of those services on a prototypical hardware setup for the realization of a case study has been presented. It is now the scope of this chapter, to provide the results of an experimental evaluation of those services.

For this purpose, we present the results of three validation experiments in this chapter. The first experiment demonstrates the non-interference of the reconfiguration of the communication schedule with already ongoing communication activities. The second one shows the ability of the RMA to handle excessive resource requests from jobs by rejecting resource requests in case the available resources are exceeded, or by selecting degraded service modes if available, respectively. With the last experiment we show that the resource protection mechanisms of the TNA guard the resources of dedicated application subsystems like safety-critical application subsystem, even in case of the presence of a failure of the RMA.

7.1 Preservation of Encapsulation

As elaborated in Section 3.1, one primary design driver of the TTSoC architecture is the support for integrating multiple application systems, possibly possessing different levels of criticality and/or originating from different vendors, into a single distributed embedded system. Therefore, the TTSoC architecture strives for facilitating the independent development of those application subsystems by offering encapsulation mechanisms that ensure that any unintended interference between application subsystems is prevented.

One key attribute of the realized resource management mechanisms is to preserve those encapsulation characteristics of the architecture. In particular, it is the goal to preserve the encapsulation of the communication channels, which are the subject of resource management in this thesis, in order to prevent any unintended interference

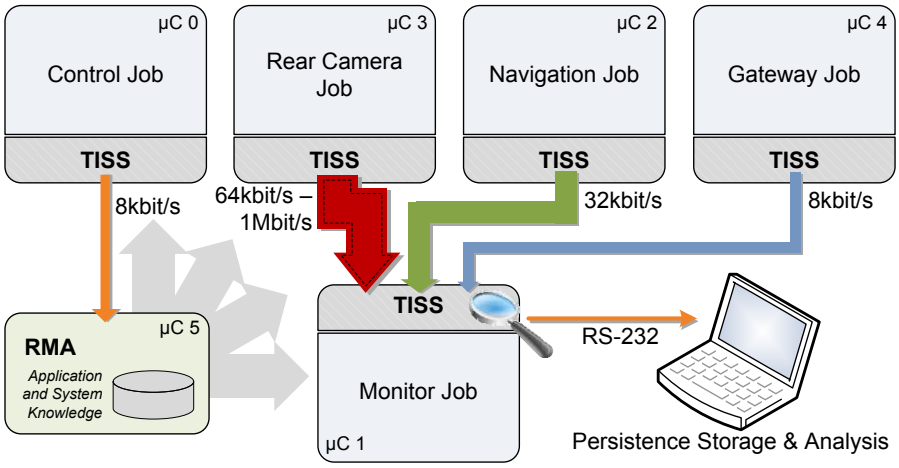


Figure 7.1: Setup of the first evaluation experiment. *The dashed arrows indicate channels with variable bandwidth.*

between application subsystems. Therefore, it is the intention of the first experiment to demonstrate that communication channels which are not subject of reconfiguration are not interfered by an ongoing reconfiguration activity.

7.1.1 Experiment Setup

The setup of the first experiment is based on the exemplary automotive application, which is described in Section 6.1: For the evaluation we have exploited all five micro components of the *Display Component*, as well as, RMA and TNA using the following system configuration (cf. Figure 7.1). The *monitor job* on *micro component 1* is configured to receive messages via two statically configured encapsulated communication channels. Via the first channel, the *gateway job*, located on *micro component 4*, periodically disseminates a message containing 64 bytes of data to the monitor job. The second encapsulated communication channel originates from the *navigation job*, which is located on *micro component 2*, and permits the periodic dissemination of a message comprising 256 byte of data to the monitor job. The exact parameters of these communication channels can be found in Table 7.1.

In addition, a dynamically reconfigurable communication channel is established that is used by the *rear camera job* located on *micro component 3*. In contrast to the earlier two channels for which the allocated bandwidth remains constant throughout the entire experiment, the bandwidth allocated to the third communication channel is varied over time (from 64 kbit/s to 1 Mbit/s). This variation of the bandwidth is achieved by altering the periodicity of the encapsulated communication channel (from 62.5 ms down to 3.9 ms), while holding the length of the transmitted data constant at 512 byte. The bandwidth variation is initiated via respective reconfiguration requests, which are disseminated to the RMA by the *control job* located on *micro component 0*.

Besides the message exchanges depicted in Figure 7.1, dedicated communication channels from the RMA to all five jobs, as well as, from the TNA to all TISSs of the

Sender	Receiver	Mode	Period	MsgLen	Bandwidth	Comment
Gateway	Monitor	0-4	62.5 ms	64 Byte	8 kbit/s	constant period
	Navigation	0-4	62.5 ms	64 Byte	8 kbit/s	and msg length
Navigation	Monitor	0-4	62.5 ms	256 Byte	32 kbit/s	constant period and msg length
Rear cam	Monitor	0	62.5 ms	512 Byte	64 kbit/s	period is varied
		1	31.2 ms	512 Byte	128 kbit/s	during the exper-
		2	15.6 ms	512 Byte	256 kbit/s	iment; msg length
		3	7.8 ms	512 Byte	512 kbit/s	is kept constant
		4	3.9 ms	512 Byte	1 Mbit/s	
Control	RMA	0-4	62.5 ms	64 Byte	8 kbit/s	constant period and msg length
RMA	Monitor	0-4	62.5 ms	256 Byte	32 kbit/s	constant period
	Navigation	0-4	62.5 ms	256 Byte	32 kbit/s	and msg length
	Rear cam	0-4	62.5 ms	256 Byte	32 kbit/s	
	Monitor	0-4	62.5 ms	256 Byte	32 kbit/s	
TNA	TISS0*	0-4	62.5 ms	512 Byte	64 kbit/s	constant period
	TISS1*	0-4	62.5 ms	512 Byte	64 kbit/s	and msg length
	TISS2*	0-4	62.5 ms	512 Byte	64 kbit/s	
	TISS3*	0-4	62.5 ms	512 Byte	64 kbit/s	
	TISS4*	0-4	62.5 ms	512 Byte	64 kbit/s	
	TISS5*	0-4	62.5 ms	512 Byte	64 kbit/s	

* receiver *TISSx* represents the TISS of micro component x

Table 7.1: Configuration parameters of the time-triggered NoC for the first evaluation experiment

micro components (including the TISS of the RMA) are established. An overview of all required communication channels for this experiment is given in Table 7.1.

7.1.2 Evaluation Procedure and Results

It is the objective of this first experiment to demonstrate that communication channels, which are not subject of reconfiguration, are not interfered by an ongoing reconfiguration activity. For this purpose, two temporal attributes of the static encapsulated communication channels received by the monitoring job are monitored and analyzed while reconfiguration activities take place, namely the phase offset and the transmission latency.

The acquisition of this information is performed directly at the TISS of micro component 1. As depicted in Figure 7.1, the TISS of this micro component is connected via a RS-232 interface to an SoC-external PC. The RS-232 connection is used to log the measurement records generated by the TISS in a persistence storage at the PC for later analysis. Listing 7.1 depicts three typical measurement records, which are generated on message reception by the TISS.

The first entry in the record identifies the communication channel—it represents the local SoC-Port identifier of the channel at the TISS of micro component 1. This is followed by a time-stamp of the measurement, which is expressed in macro ticks

Listing 7.1: Log file excerpt of first evaluation experiment

```
...
4    46-e3012208    56    7    2265    -1    64
2    46-e3014da0    0     10    330    -1    8
3    46-e302daa8    57    10    726    -3    32
...
```

of the global time in the SoC component. The third entry represents one dedicated byte of the message that is transmitted via the respective channel in order to check that the data is correctly transmitted and no messages are lost.

The following two fields identify period and phase offset of the message as it is currently configured by the TNA. This information is extracted from the configuration memory of the TISS, i.e., in this implementation from the MEDL memory of the TTE controller. During the analysis of the log file, this information is used to determine, whether the temporal attributes of the static communication channels are affected by an ongoing reconfiguration.

The sixth entry of each measurement record holds the *TimeDiff Capture* value (see [Steinhammer, 2007, p. 124]), which denotes the difference between the expected and the actually occurred reception instant of a particular message. This value is automatically adjusted by the TTE controller with the transmission latency induced by the TTE switch. This latency is a configuration parameter of the TTE controller and is set for a system configuration comprising a single TTE switch to 9 macro ticks, i.e., approximately 4.29 μs.

The last field of each measurement record holds the length of the message disseminated over the respective communication channel. According to the TTE specification [Kopetz et al., 2006, p. 34], the message length is stored in the TTE controller in blocks of 8 bytes. Hence, in the example presented in Listing 7.1 the value 64 denotes an actual message size of 512 bytes.

Impact on the Message Schedule

The chart depicted in Figure 7.2 presents the change of the phase offsets of the individual messages of the experiment setup over time. The x-axis represents the progression of real-time in terms of the number of elapsed periods with a length of 62.5 ms since the epoch of the TTE system, i.e., the power-on instant of the SoC component. This period length is used in the case study as the periodicity of reconfiguration instants. Thus, the chart plots the observed phase offset of the individual messages after every reconfiguration instant.

At particular instants, marked in the chart with dashed lines, mode changes are requested by the control job. As shown in Figure 7.2, despite the presence of mode changes, the phase offset of the static communication channels remain constant at 10560 macro ticks for message 0, respectively, 23232 macro ticks for message 1, while

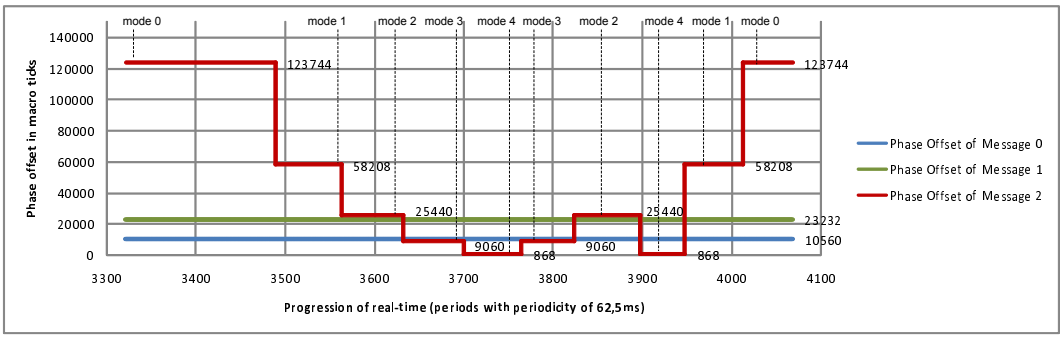


Figure 7.2: Modification of the phase offset due to reconfiguration

the communication channel originating from the rear camera job is rescheduled at every reconfiguration instant.

The phase offset of the dynamic communication channels varies from 868 macro ticks in mode 4 (with a message period of 3.9 ms) to 123744 macro ticks in mode 0 (with a message period of 62.5 ms). It is obvious that the maximum phase offset which can be assigned to a message by the scheduling algorithm increases with the length of the period (see Table 7.2).

Mode	Period	Phase	max. Phase*
0	62.5 ms	123744	131072
1	31.2 ms	58208	65536
2	15.6 ms	25440	32768
3	7.8 ms	9060	16384
0	3.9 ms	868	8192

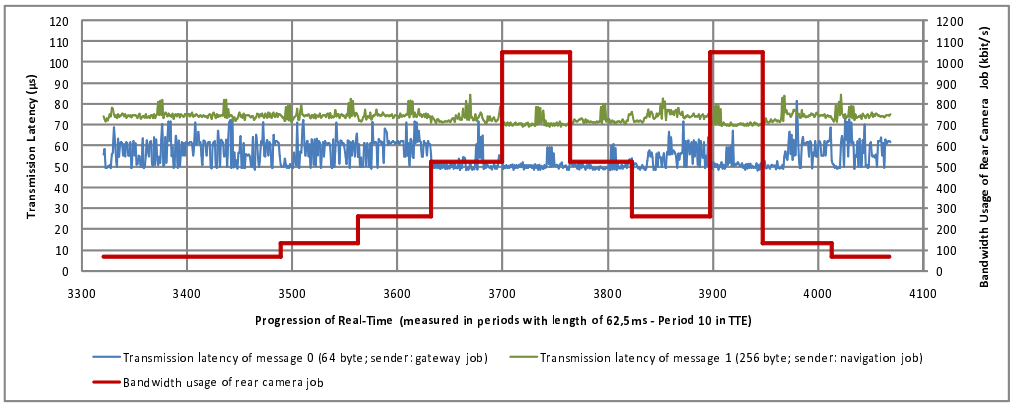
* max. phase offset for the respective period according to the TTE specification

Table 7.2: Phase offset (in macro ticks) of message 3 in different modes

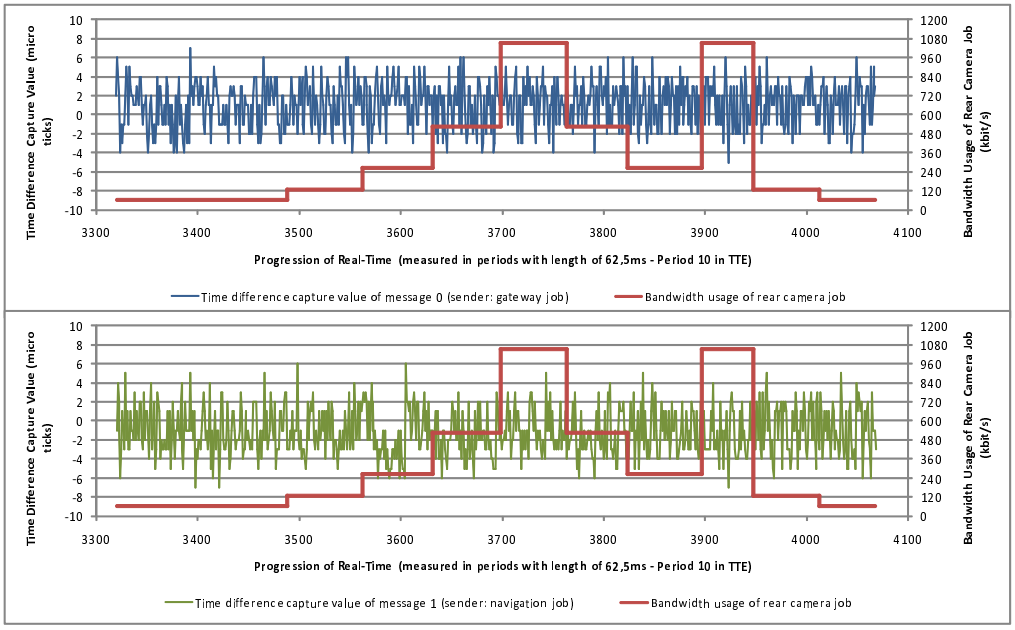
Impact on Message Transmission Latency

The chart depicted in Figure 7.3(a) represents the variation of the message reception latency of both static communication channels at micro component 1. The latency values are acquired by taking a time-stamp of the global time at the TISS of micro component 1, immediately before the receive interrupt for the particular message is signaled to the host and calculating the difference to the expected reception instant as defined in the MEDL. Thus, the latency subsumes the following time durations:

- the transmission time of the entire message over the 100 Mbit/s Ethernet connection (including Ethernet header, TTE header, and the delay of one TTE switch),
- the operating system overhead for the activation of the TISS software (realized as RTAI Linux kernel module),



(a) Message reception latency of message 0 and message 1



(b) Deviation of expected and actual receive instants at micro component 1

Figure 7.3: Jitter of message reception latency during reconfiguration

- the access of the TISS software to the MEDL memory of the TTE controller,
- the preparation of the required data structures for signaling the reception to the host, as well as,
- the deviation of the actual from the expected instant of reception of a particular message.

The deviation between expected and actual instant of reception at a particular micro component depends on the quality of the clock synchronization of the TTE system, which interconnects the micro components. Evaluation experiments of

Steinhammer [Steinhammer, 2006, p. 134] have shown that the maximum deviation is below half a macro tick, which equals $2^{-21}\,s = 476.8\,ns$ (one macro tick typically equals 25 micro ticks). The chart shown in Figure 7.3(b) depicts this deviation, by plotting the contents of the *TimeDiff Capture* value (see [Steinhammer, 2007, p. 124]) of the TTE controller measured in micro ticks.

In all three charts depicted in Figure 7.3(a) and Figure 7.3(b), the secondary y-axis displays the bandwidth usage of the message originating from the rear camera job. Thus, this graph denotes the instants of reconfigurations initiated by requests from the control job (cf. Figure 7.1). As it can be seen from those charts, no significant correlation between reconfiguration instants and the jitter of the message reception latency, respectively, the deviation of expected and actual message receive instant can be detected. Thus, the communication channels that are not subject of reconfiguration are not impaired by the reconfiguration activities of the dynamic communication channel.

7.2 Handling of Excessive Resource Requests

The purpose of the second validation experiment is to show the ability of the RMA to react to increasing resource requests from individual jobs, resulting in bandwidth requirements which would exceed the available resources, respectively, would fail to be scheduled due to message collisions.

In such a case, there exist two alternatives for the RMA to react:

- Reject of the resource request and notification of the job regarding the reject
- Selection of a suitable degraded mode of operation, if defined

Nevertheless, it is the purpose of the TNA as the final authority in the entire resource management process to detect possible collisions on the time-triggered NoC and ensure the correct operation of protected communication channels.

7.2.1 Experiment Setup

The setup of this experiment is similar to the one described in the previous section. As depicted in Figure 7.4, the micro components involved in the experiments, as well as, the established communication channels are the same as in the previous experiment. The acquisition of the experiment data, however, is performed at the RMA. At the RMA, the output regarding the requested mode of operation, the actually chosen mode after the scheduling, as well as, temporal properties of the resource management process (e.g., the required time to generate the schedule, the response time from the TNA, etc.) is transmitted via a serial connection to a persistent storage on a PC.

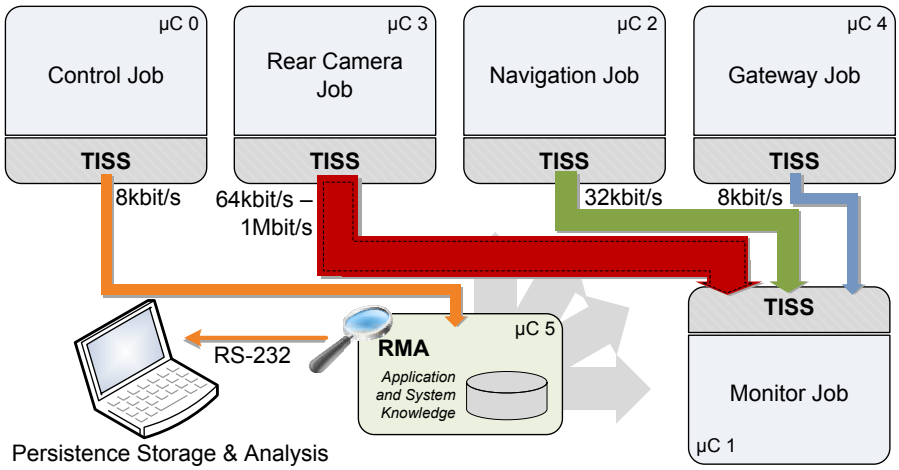


Figure 7.4: Setup of the second evaluation experiment

The configuration of the static communication channels (i.e., the communication channels between gateway job and monitor job, navigation job and monitor job, control job and RMA, as well as, all communication channels originating from RMA and TNA) equals the previous experiment (see Table 7.1). For testing the behavior of the RMA in presence of resource requests that exhaust the available resources, six primary modes have been defined (cf. Table 7.3), which increase the bandwidth requirement of the rear camera job.

In *mode 0* the rear camera job uses a single communication channel with a period of 1.95 ms and a message length of 512 byte, which results in a bandwidth usage of 1 Mbit/s. In each further mode, two additional communication channels are requested for the rear camera job, resulting in a bandwidth usage of 11 Mbit/s in *mode 5* and *mode 6*. For mode 6 we have defined a *degraded mode 6a*, which has an overall bandwidth demand of 10 Mbit/s. The reason to add additional communication channels instead of decreasing the periodicity of individual communication channels is that 1.95 ms is the shortest TTE message period that is usable in our case study implementation. This limitation is caused by the time required for the reconfiguration of the TISS software during which no messages are allowed to be on transit on the NoC. In our case study implementation, this time equals approximately 1 ms.

7.2.2 Evaluation Procedure and Results

The objective of this experiment is to demonstrate that the RMA detects resource requests that exceed the available resources or fail to be scheduled due to conflicts with other communication channels. Therefore, the input to the RMA, the time required for generating a communication schedule, as well as, the output of the RMA are monitored and analyzed.

As depicted in Figure 7.4, data acquisition for this experiment is performed at the RMA via a RS-232 connection to an SoC-external PC, which logs the measurement

Mode	UFIM-port Nr.	Period	MsgLen	Bandwidth
0	rear-cam:0	1.95 ms	512	1 Mbit/s
1	based on configuration of mode 0			1 Mbit/s
	rear-cam:1	1.95 ms	512	1 Mbit/s
	rear-cam:2	1.95 ms	512	1 Mbit/s
2	based on configuration of mode 1			3 Mbit/s
	rear-cam:3	1.95 ms	512	1 Mbit/s
	rear-cam:4	1.95 ms	512	1 Mbit/s
3	based on configuration of mode 2			5 Mbit/s
	rear-cam:5	1.95 ms	512	1 Mbit/s
	rear-cam:6	1.95 ms	512	1 Mbit/s
4	based on configuration of mode 3			7 Mbit/s
	rear-cam:7	1.95 ms	512	1 Mbit/s
	rear-cam:8	1.95 ms	512	1 Mbit/s
5	based on configuration of mode 4			9 Mbit/s
	rear-cam:9	1.95 ms	512	1 Mbit/s
	rear-cam:10	1.95 ms	512	1 Mbit/s
6	based on configuration of mode 4			9 Mbit/s
	rear-cam:9	1.95 ms	512	1 Mbit/s
	rear-cam:10	1.95 ms	512	1 Mbit/s
6a	degraded mode of mode 6 - based on mode 4			9 Mbit/s
	rear-cam:9	1.95 ms	512	1 Mbit/s

Table 7.3: Primary modes for the second evaluation experiment

Listing 7.2: Log file excerpt of second evaluation experiment

```
...
Mode requested:  2 - time: 22468310 us
Scheduling done - time: 22473254 us
TNA accepted mode change - time: 22513308 us
Active mode: 2 - time: 22513621 us
...
```

records. Listing 7.2 depicts a typical measurement record produced by the RMA.

The first line represents the input to the RMA (i.e., the requested primary mode) as it is received from the control job, followed by a time stamp of the local clock at the RMA that denotes the reception instant of the resource request.

The result of the message scheduling at the RMA is stored in the second line of the measurement record. This line holds the information, whether the scheduling was successful ("Scheduling done") or has failed. The time stamp represents the value of the local clock at the RMA immediately after finishing the scheduling. Thus, the time needed for generating the message schedule yields from the difference of the first two time stamps.

The third line indicates whether the TNA has accepted the new SoC configuration or not. The response and processing time of the TNA can be calculated out of the third time stamp. The last line of the measurement record depicts the output of the RMA again. It contains the primary mode (or degraded mode of operation) that is actually chosen after the entire resource management procedure and that is

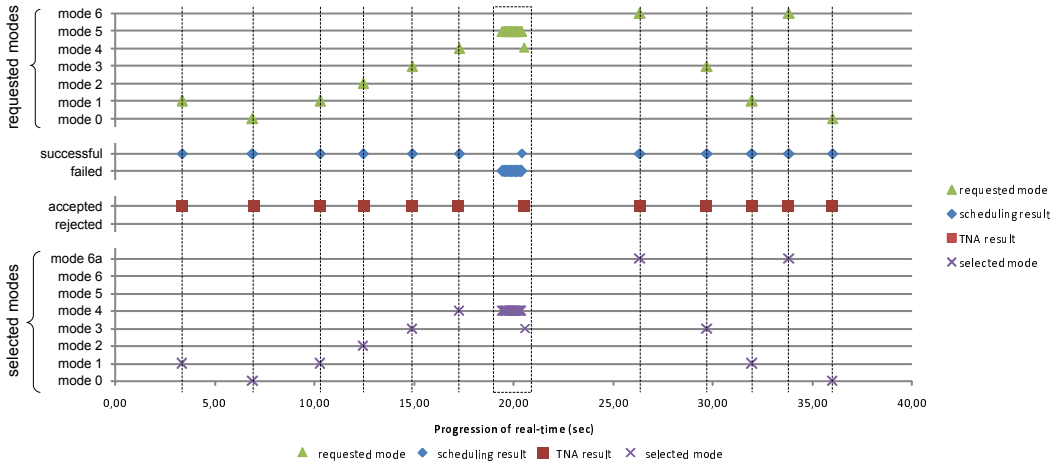


Figure 7.5: Response of RMA and TNA to resource requests

communicated to all hosts of the respective DAS. The last time stamp is taken after the preparation of the RMA-to-Host messages, which inform the hosts about the actual configuration, i.e., active mode, active job, as well as, UFIM-Port to SoC-Port mapping.

For the validation of the RMA reaction on resource requests, we have used the following test procedure: The control job located on micro component 0 requests periodically a primary mode from the RMA. The requested mode is modifiable via the user interface of the control job. Starting with *mode 0*, the bandwidth requirements of the rear camera job are stepwise increased by successively requesting *mode 1*, *mode 2*, and so on. The output of RMA and TNA as response to those mode requests is depicted in Figure 7.5 (each mode request is marked by a dashed vertical line).

As it can be seen from Figure 7.5, the RMA is able to derive a schedule for the demanded messages for the modes *0* to *4*. Figure 7.6 visualizes this computed schedule. The selection of *mode 5*, respectively *mode 6* would lead to an overall bandwidth usage of the rear camera job of 11 Mbit/s (cf. Table 7.3), for which, as it is indicated in Figure 7.6, no valid schedule can be found by the RMA. In such a case, the RMA rejects the particular resource request and the current mode remains active. This is depicted in Figure 7.5 at approximately 20 sec after the start of the experiment: The control job requests *mode 5*, but the last successfully requested mode, *mode 4*, remains active. In the current implementation of the RMA, the scheduling algorithm is only started, if at least for one DAS a mode is requested that differs from the actually active one. Since in this experiment the control job periodically disseminates the currently selected mode to the RMA (the selection is realized via the job's user interface), the RMA is invoked every reconfiguration period, which equals 62.5 ms in the current setup.

In contrast to mode 5, a degraded mode of operation representing a lower QoS level is defined for mode 6, denoted as *mode 6a*. Since the RMA is able to construct

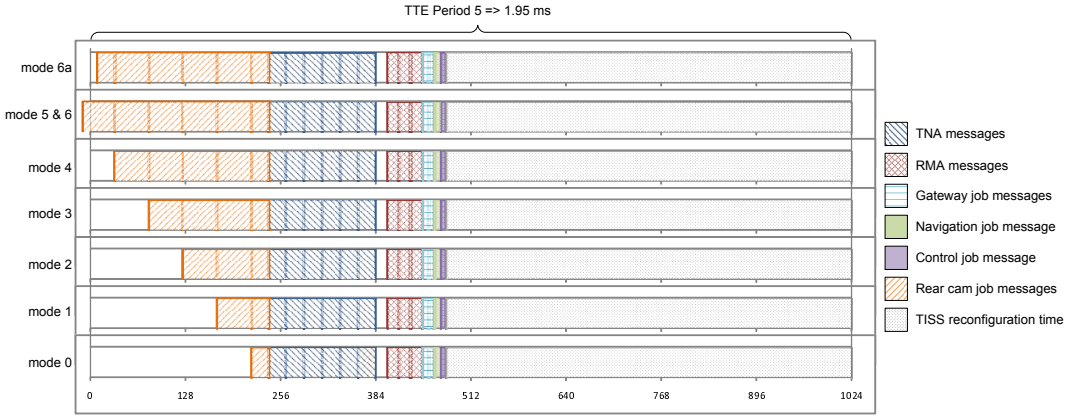


Figure 7.6: Message schedule calculated by the RMA

a valid schedule for this degraded mode, the mode is selected by the RMA and is communicated to all jobs of the DAS.

For the visualization of the schedule in Figure 7.6, only a single period with a periodicity of 1.95 ms is presented. In order to visualize potential collisions with messages exhibiting larger periods (e.g., originating from the communication between gateway job and monitor job, navigation job and monitor job, etc.), the phase offsets of those messages are transformed to their correspondents in the chosen period of 1.95 ms. The process of message transformation is explained in Section 6.3.3. The interval denoted as *TISS reconfiguration time* represents the time required for the reconfiguration of the TISS software during which no messages are allowed to be on transit on the NoC. As explained before, this limits us to use TTE period 5, which equals 1.95 ms, as the shortest period for our experiments.

7.3 Validation of Resource Protection Mechanisms

As stated in Section 3.2.5, the main purpose of the TNA is to act as a guard for reconfiguration activities performed by the RMA. Therefore, the TNA is enabled to detect potential collisions on the time-triggered NoC or violations on resource protections, e.g., a differing phase of a protected communication channel. In such a case, the TNA rejects the new configuration and the current one remains unchanged.

The objective of this validation experiment is to supply a first evidence that the resource protection mechanisms of the TNA preserve the temporal properties of the protected communication channels (phase offset, bandwidth reservation) even in presence of a failure of the RMA.

7.3.1 Experiment Setup

The architectural elements of the TTSoC architecture involved in this experiment are solely RMA and TNA. As depicted in Figure 7.7, a static configuration of

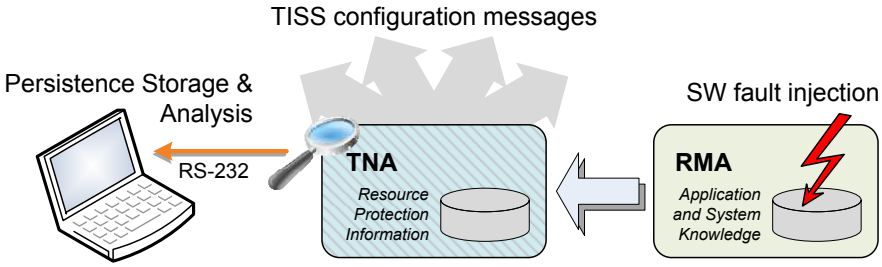


Figure 7.7: Setup of the third evaluation experiment

the communication schedule is stored at the RMA, which is disseminated to the TNA to be checked for correctness, i.e., for the lack of collisions, as well as, for the correct configuration of protected resources. The static configuration used for this experiment equals to the configuration of *mode 0* used in the first experiment (see Table 7.1), except for the rear camera job. Unlike as described in Table 7.1, a periodicity of 1.95 ms is specified for the outgoing message of the rear camera job.

The protected resources are specified by an XML file as defined in Section 6.3.4. The protection file used for this experiment includes the following single entry:

```
<uC id="0">
    <Port period="PERIOD_P10" phase="2500" length="64"
          direction="DIR_OUTPUT" type="TYPE_PERIODIC"/>
</uC>
```

This entry specifies that the parameters of an outgoing message from micro component 0 (used by the control job) have to be protected by the TNA. The concrete parameters that are fixed for this experiment are *period*, *phase offset*, *message length*, *message direction*, and *message type*. Please note that it is also possible to specify only a subset of those parameters, which would decrease the level of protection by the TNA.

The acquisition of the experiment data is done at the TNA, where the output stating whether a configuration from the RMA is accepted or not, is transmitted via a serial connection to a persistent storage on a PC.

7.3.2 Evaluation Procedure and Results

In order to analyze the protection mechanisms of the TNA, two different tests using software fault-injection are performed. While the first one inspects the ability of the TNA to detect collisions on the NoC, the second one examines the protection of the temporal configuration parameters period and phase offset of the communication channel originating from the control job.

In the first test, the phase offset of a single message is altered after the schedule generation has been performed by the RMA. This simulates a failure of the RMA, which may result in a conflicting schedule. This is realized by successively assigning

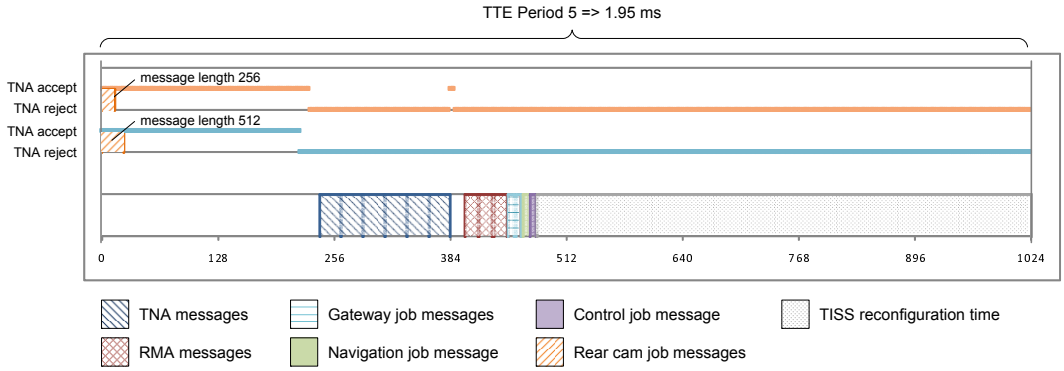


Figure 7.8: TNA response to varying phase offset of rear camera job message

all possible values of the phase offset, i.e., $0x000$ to $0x3FF$ since the phase offset of a message with period 5 in TTE is made up of 10 bits, to the respective data field in the configuration data structure. Afterwards, the modified configuration data is disseminated to the TNA and the response of the TNA is recorded.

Figure 7.8 depicts the output of the TNA, i.e., either accept or reject, depending on the phase offset of the outgoing message of the rear camera job. The first test run is performed with a message length of 512 bytes, for a second test run the length is reduced to 256 bytes. As it can be seen from this figure, any time a collision with the other messages would occur, the TNA rejects the configuration.

The main difference between the two configurations with differing message lengths arises at a phase offset of 384: While the message with a length of 512 bytes is too long to be sent during the time between the transmission of TNA and RMA messages, a collision-free transmission of the 256 byte long message during this time is possible. At a phase offset of 390, the TNA rejects the configuration again, due to a detected collision.

For the second test, exhaustive software fault-injection has been performed for all protected parameters of the message originating from the control job. For this reason, all possible values for the parameters message period, phase, length, direction, and type have been successively assigned to the respective fields in the configuration data structure. Table 7.4 represents the theoretical range of the individual parameters, as well as, their valid range according to design restrictions. For instance, for the TTE periods 15 to 7, 12 bits are available for the specification of the phase of the message. For each TTE period shorter than period 7, the number of bits is reduced by one resulting in five bits for the shortest TTE period that is used for these experiments.

The results of this second test have shown that any deviation of a configuration parameter of a protected communication channel is detected by the TNA, which results in a reject of the proposed configuration. Thus, a fault within the RMA that affects the contents of the configuration data cannot affect the correct operation of protected communication channels.

Name	Width	Min.	Max.	Restriction
period	4 bit	0x0	0xF	no restriction: 16 periods are defined in TTE
phase	12 bit	0x000	0xFFF	12 bits for period 10 to period 7; number of bits is reduced by one for each following period
length	8 bit	0x00	0xFF	length is specified in 8 byte blocks; the theoretical max. length of 2048 bytes, is restricted by Ethernet specification (max. 1500 bytes allowed)
direction	1 bit	0	1	no restriction: direction either input or output
type	2 bit	0	4	we support only periodic (type = 0) and sporadic (type = 1) messages

Table 7.4: Data range of configuration parameters

Chapter 8

Conclusion

The main contribution of this thesis is the design and development of a resource management solution for the TTSoC architecture. One major design goal is to preserve the encapsulation and fault-isolation properties of the TTSoC architecture despite the introduction of dynamic resource management. Additionally, we aim at an easier development of integrated systems that host applications with different dependability requirements. Due to the advancements of computational power and hardware costs in the area of embedded systems, these mixed-criticality systems become attractive for many application domains (e.g., aerospace domain or automotive domain).

8.1 Encapsulation of Application Subsystems

In the recent past, in various application domains (e.g., in the aerospace domain or the automotive domain) a number of efforts have been made to realize a paradigm shift from federated system architectures towards integrated architectures. In a federated system, each application system has its own dedicated computer system which is often tailored to a particular application. In contrast, an integrated architecture is characterized by the integration of multiple application systems within a single distributed computer system. The paradigm shift taking place is mainly caused by the massive increase of functionality of embedded applications, which results in a continuous escalation of the number of interacting physical components. The consequences of this development are twofold: first, the costs for system development and for hardware increase, and secondly, the inherent system complexity also rises. Choosing integrated architectures reduces these issues.

For this purpose it is important for an integrated architecture to facilitate composability, robustness, and modular certification. Therefore, it is a pivotal property of an integrated architecture to achieve encapsulation of the individual hosted applications. The TTSoC architecture provides this encapsulation by offering a predictable on-chip interconnect that is free of interference. Each computational element, denoted a micro component, is assigned dedicated slots in a time-triggered

communication schedule, which are protected from other micro components through the communication system. In addition, this encapsulation results in a complexity reduction, because the behavior of interfering subsystems is more difficult to understand and to reason about than the behavior of cleanly encapsulated subsystems.

In this thesis we presented a resource management solution for the TTSoC architecture that is able to preserve this encapsulation despite the presence of dynamic reconfiguration. This is achieved by combining the advantages of static resource allocations with the flexibility that is provided by a QoS-based resource allocation. Therefore, all possible system operation modes, together with their corresponding resource requirements need to be specified a priori for each application subsystem. This is realized by defining the *primary modes* for each application subsystem. In addition, we propose to specify different *degradation levels* for each primary mode, which represent the operational mode of the application subsystem with a lower QoS level, and related therewith, with changed resource requirements. On the one hand, this strategy enables the dynamic reallocation of resources to application subsystems even across application subsystem borders in order to react to changing resource demands. On the other hand, it also permits a static analysis of the allotment of resources to application subsystems, hence it becomes possible to verify that in all possible operation modes a feasible allocation can be found four all applications.

In addition to the possibility of this static analysis, we preserve the encapsulation of the application subsystems by enabling the off-line specification of static resource allocations. To this end, we provide architectural protection mechanisms to ensure that these static allocations are not affected by any system reconfiguration. We experimentally evaluated and approved these protection mechanisms while performing software fault-injection experiments in our prototypical implementation of the TTSoC architecture.

8.2 Support for Mixed Criticality Systems

Encapsulation is also of particular importance for the implementation of mixed-criticality systems for the above mentioned application domains. For instance, a future automotive system will incorporate applications ranging from a safety-critical drive-by-wire application subsystem to a non safety-critical comfort application subsystem. In such a mixed criticality system, a failure of micro components of a non safety-critical application subsystem must not cause the failure of application subsystems of higher criticality.

Additionally, mixed-criticality systems have significant effect on the system design paradigm. In general, safety-critical and non safety-critical applications systems will involve fundamentally different design paradigms. The focus of safety-critical applications lies on simplicity and determinism in order to facilitate thorough verification and validation. In contrast, non safety-critical applications can provide more complex application services, e.g., they need to deal with insufficient a priori knowledge about the environment, and require dynamic behavior to handle the challenges of

evolving application scenarios and changing environments. Likewise, the support for mixed criticality systems had a significant impact on the design and development of the resource management solution: On the one hand, the resource allocation should be flexible and nearly optimal w.r.t. resource utilization to be competitive in the implementation of non safety-critical applications. On the other hand, predictability, determinism, and fault isolation are vital characteristics for resource management regarding safety-critical systems.

The resource management solution presented in this thesis follows this distinction of application systems and provides two distinct architectural elements for enabling integrated resource management, the *Trusted Network Authority* (TNA) and the *Resource Management Authority* (RMA). The RMA computes new resource allocations for the non safety-critical application systems, while the TNA verifies and actually executes the resource reallocation and ensures that the new resource allocations have no negative effect on the behavior of all hosted application systems (in particular the safety-critical application systems).

For this purpose, the TNA is responsible (i) for verifying that the resource schedule computed by the RMA is free of collisions, (ii) for verifying that protected resources, i.e., resources that are statically assigned to a dedicated application system, comply to their off-line specification, and (iii) for updating the configuration of the micro components without interfering with the service of the hosted applications. This is achieved, e.g., by periodically disseminating configuration messages at an a priori known instant—the reconfiguration instant.

For the development of safety-critical applications, the certification of the system is of utmost importance. We address this issue by splitting the responsibilities of resource management into two parts, one solved by the RMA the other by the TNA. However, only the TNA is part of the trusted subsystem of the TTSoC architecture, but not the RMA. This way, we simplify the certification of the SoC significantly, since the TNA provides only a small stable set of generic services. These services need not to be changed for the realization of different application services. Thus, once an implementation of the TNA is certified, it can be reused for different applications. For ultra-dependable systems for which a static resource allocation is often feasible, only the TNA has to be certified up to the highest criticality level. In contrast, the RMA, which has to perform the quite complex task of computing the resource allocation, has only to be certified to that level of criticality that is required by the most critical application system that demands dynamic resource management.

The validation experiment of the resource protection mechanisms provided by the TNA confirms that they preserve the temporal properties of protected communication channels (e.g., phase offset, bandwidth reservation) also in the presence of a failure of the RMA.

8.3 Further Work

This thesis presents the design and first implementation of a resource management solution for the TTSoC architecture. The implementation has been realized in the course of the European research project DECOS. At present, in the scope of the national research project TT-SoC an implementation of the TTSoC architecture on an FPGA-based hardware platform is ongoing, which opens up a lot of interesting fields of research. Among these are:

Handling of concurrent message transmissions. As part of this FPGA-based implementation of the TTSoC architecture, the resource management solution is realized on an updated hardware platform. The realization of the RMA is of particular interest, since the (re)implementation of the time-triggered NoC will permit the concurrent transmission of messages via parallel communication channels.

Integration in model-driven design. By integrating resource management into a model-driven design methodology, the data structures for parameterizing the architecture w.r.t. resource management can be automatically derived from the models. This dramatically reduces the manual workload for system designers.

Power awareness. The resource management solution presented in this thesis lays the foundation for realizing power-aware behavior of the TTSoC architecture. In particular, the utilization of DVFS and DPM techniques for managing the power dissipation of individual micro components represents a promising approach for further research.

The theoretical and practical findings that have been elaborated in this thesis provide a basis for further research on resource management in integrated time-triggered real-time systems.

Appendix A

TTSoC Resource Management Interfaces

This chapter describes in detail the realization of the interfaces for resource management for the particular implementation of the TTSoC architecture presented in this thesis.

A.1 TISS CP Interface

The Configuration and Planning (CP) interface of the TISS is accessed by the TNA via the memory interface depicted in Figure A.1. The layout of this memory interface is identical at each micro component. However, its size differs from micro component to micro component depending on the number of input or output ports provided by the TISS. The content and layout of this memory interface towards the TNA is described in the following.

Global Time Register (32 bit): Each TISS possesses a 32 bit register holding the actual instant of the global time. The global time is established at each TISS by synchronizing its local global time register to dedicated synchronization messages (*TTsync* messages of the TTE protocol), which originate from a dedicated rate master. The rate master is realized by the TNA.

Service Level (16 bit): This data field represents the designated service level of the host at which it should operate. The service level of a host can be used for instance to perform DPM by deactivation the host whenever the particular service of the hosted job is not required. The supported gradations of the service level may differ from host to host. However, at least two levels have to be supported by every host implementation: full-operational mode and shutdown mode (the shutdown can be physical enforced by the TISS using the power control service).

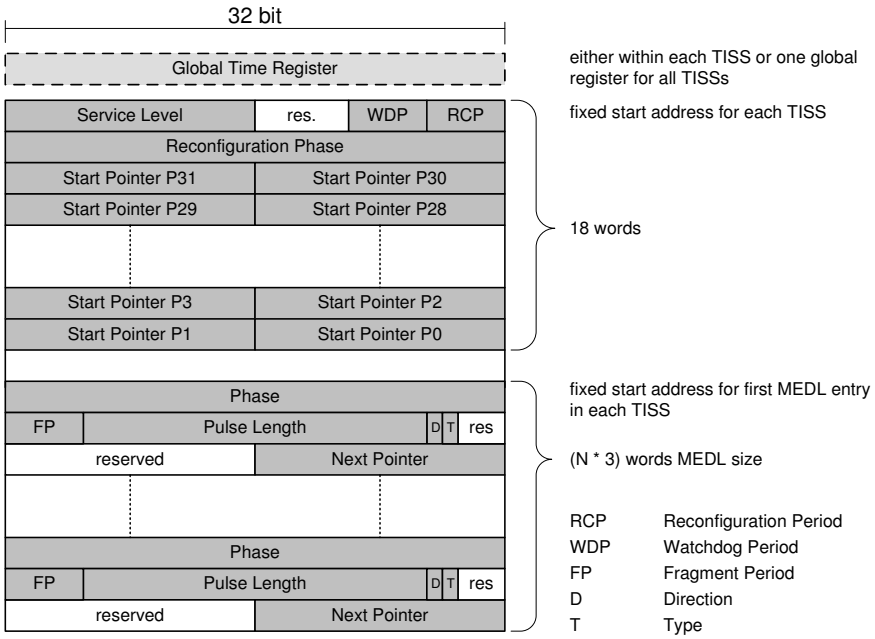


Figure A.1: Memory interface of the TISS towards the TNA. The figure represents a design alternative in which the global time is updated within each TISS by the TNA

Watchdog Period (5 bit): As described in Section 3.2.1 each TISS provides a watchdog service for its host in order detect and recover from crash failures of hosts. The watchdog period (WDP) data field specifies the frequency with which the host has to send its life sign to the TISS. With the five bit-wide field, up to 32 different watchdog periods can be distinguished.

Reconfiguration Period (5 bit): The reconfiguration period (RCP) field contains the periodicity of the reconfiguration actions performed by the TNA. The five bits of the field reconfiguration period permit the specification of up to 32 different reconfiguration periods.

Reconfiguration Phase (32 bit): The reconfiguration phase denotes the 32 bit phase offset from the start of the reconfiguration period. Together with the reconfiguration period, it denotes the global point in time at which all TISSs and all hosts switch from the old configuration to the new one. This permits the synchronization of the reconfiguration of all micro components on the SoC.

Start Pointer (P0-P31, each 16 bit): The configuration data of the communication ports, i.e., the description of the pulsed data stream realizing the message transfer over a particular port, is organized as linked list in the memory of each TISS. The memory address of the configuration data for the first pulsed data stream of a particular period (i.e., pulsed data stream with the lowest phase offset) is stored in the respective start pointer field.
If no configuration data for a particular period exists, the value of the start

pointer is set to the invalid address *0xFFFF*.

Besides this information regarding the general configuration of the TISS and the host of the micro component, each micro component possesses a memory region of fixed size (but with different size for the individual TISSs) for holding the port configuration data. The configuration of a single port requires the following information (cf. the lower part of Figure A.1).

Phase (32 bit): A pulsed data stream can be uniquely identified by the period and the phase offset of the pulse (since there exist no parallel communication channels). Since the period of the pulse is implicitly known for each port configuration entry (either the *start pointer* of a particular period or the *next pointer* of a port configuration entry of the same period points to this memory location), only the phase offset of the pulse is stored.

Fragment Period (5 bit): The fragment period denotes the dissemination period of the individual fragments a pulse consists of. Due to its five bit representation, 32 different fragment periods can be distinguished. However, for pulsed data streams comprising more than one fragment, the fragment period has to be lower than the period of the pulsed data stream itself. For example, for a pulsed data stream comprising 8 fragments, the fragment period has to be at least 8 times lower than the period of the pulsed data stream (e.g., period 2^{-4} for the pulsed data stream requires at least a period lower or equal than 2^{-7}).

Pulse Length (22 bit): The number of fragments a pulsed data stream consists of is represented by the *pulse length* field. Due to the field length of 22 bit, the design of the TTSoC architecture permits a maximum number of $2^{22} = 4194304$ fragments for a single pulse. Considering, e.g., , a fragment size of 128 bit, this leads to a maximum pulse length of more than 67 $MByte$.

Direction (1 bit): The *direction* bit defines whether the port is an output port (*0b1*) or an input port (*0b0*).

Type (1 bit): The *type* bit defines whether the message is realized as periodic time-triggered (*0b1*) or sporadic time-triggered (*0b0*) message.

Next Pointer (16 bit): The next pointer points to memory region containing the next pulsed data stream configuration of the same period. If the actual port configuration data is the last for a particular period, the *next pointer* is set to the constant value *0xFFFF*.

A.2 TNA RCLIF Interface

The RCLIF interface of the TNA is accessed by the RMA via a memory interface. The layout of this interface is illustrated in Figure A.2 (the layout of the interface

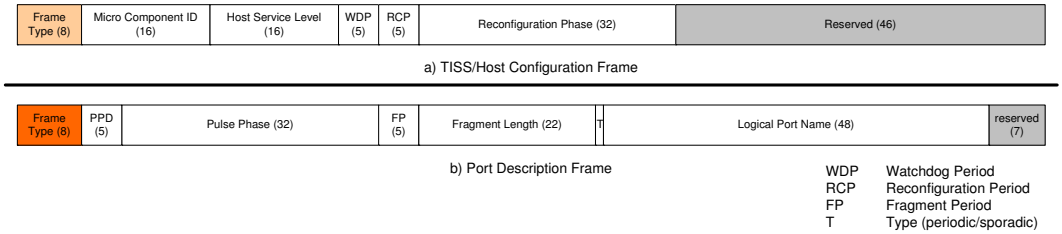


Figure A.2: Memory interface of the RMA towards the TNA

is identically structured as the RMA-to-TNA configuration message). The interface is grouped into several blocks, one for each micro component. Each block has the following layout of frames (the content of the frames is also depicted in Figure A.2):

1. Each block starts with exactly one TISS/host configuration frame. (see frame a) in Figure A.2)
2. The first frame is followed by one or more input port description frames (see frame b) in Figure A.2), which define the input ports of the TISS towards the NoC.
3. The micro component configuration block is concluded by one or more output port description frames (see frame b) in Figure A.2). Input and output port description frames are only distinguished by their *frame type* field.

The structure of the entire memory layout is composed out of three different types of frames, each made up of 16 bytes. The first byte of each frame always identifies the type of the frame. We distinguish TISS/host configuration frames, as well as, input and output port description frames.

TISS/host configuration frame

The TISS/host configuration frame denotes the beginning of a micro component configuration block within the memory layout of the RCLIF at the TNA, as well as, of the configuration message disseminated from RMA to TNA.

Frame Type (8 bit): The first byte identifies the type of the frame by the use of a unique identifier. In case of the TISS/host configuration frame the type field is identified by the value *0x78*.

Micro Component ID (16 bit): The micro component ID establishes the mapping of the micro component configuration block to the corresponding micro component.

Host Service Level (16 bit): This data field represents the service level of the host at which it should operate. At least two service levels have to be supported by

every host implementation: full-operational mode (*0xFFFFh*) and shutdown mode (*0x0000h*). The implementation has to ensure that shutdown can be physically enforced by the TISS.

Watchdog Period (5 bit): The *watchdog period* data field specifies the frequency with which the host has to send its life sign (e.g. updating particular bits in a watchdog register) to the TISS. With the *watchdog period* data field, up to 32 periods can be distinguished.

Reconfiguration Period (5 bit): The micro components are periodically reconfigured by the TNA. The *reconfiguration period* in combination with the *reconfiguration phase* determines the the periodic global point in time at which the micro component has to to switch to the new configuration.

Reconfiguration Phase (32 bit): The *reconfiguration phase* denotes the phase offset of the start of the *reconfiguration period* and thus represents the periodic instant, at which the new configuration becomes valid.

Port description frame

The contents of the port description frame denote the configuration of the outer ports of the TISS towards the on-chip network. We distinguish two types of port descriptions frames: one for the description of input ports and one for output ports. Both types are syntactically identical and only discriminated by their *frame type*.

Frame Type (8 bit): The port description frame for input ports is identified by the constant value *0x8B* in its type field. The port description frame for output ports by the constant value *0x49*.

Pulse Period (5 bit): The *pulse period* field permits the differentiation 32 periods (from the longest period with duration 2^0 seconds to the shortest one with a duration of 2^{-31} seconds.

Pulse Phase (32 bit): The *pulse phase* field denotes the phase offset of the pulse from the start of the period defined in *pulse period*.

Fragment Period (5 bit): The fragment period denotes the dissemination period of the individual fragments a pulse consists of. The fragment period has always to be lower than the period of the pulsed data stream itself.

Pulse Length (22 bit): The field represents the length of the pulse by specifying the number of fragments the entire pulse comprises. The maximum number of fragments that can be specified for a single pulse is $2^{22} = 4194304$ fragments.

Type (1 bit) : The *type* bit defines whether the message transmission is realized as a periodic time-triggered (*0b1*) or sporadic time-triggered (*0b0*) message.

UFIM Port Name (48 bit): The *UFIM port name* is used to specify a port identifier that is unique for the entire SoC. The *UFIM port name* is composed out of a *DAS identifier* (unique identifier of the DAS), a *job identifier* (unique name of a job within a DAS), and a *port identifier* (unique identifier in the name space of a single job).

Appendix B

XML Configuration Files

This chapter demonstrates the use of XML documents for the specification of initial parameters for the system configuration. For this purpose, an exemplary configuration file for the TNA, as well as, an exemplary specification for protected resources is shown in the following.

B.1 Initial TNA Configuration Example

The XML document excerpt depicted in Listing B.1) presents a partial example of an initial TNA configuration, which is compliant to the XSD schema described in section 6.3.1 on page 98. As defined by the XSD schema, the root element of the XML file is called `RMA_TNA_Msg`. The root element is followed by a sequence of (at least one) `uC` elements. For the support of version management of the XML files, the `RMA_TNA_Msg` element comprises a mandatory element *version*, as well as, an optional attribute *author*.

Each `uC` element encloses the entire configuration of a micro component. The configuration block of a particular micro component is identified by the attribute *id*. The *id* of a micro component is represented by a 16 bit value; thus, up to 2^{16} different micro components can be distinguished on one single SoC component.

A valid XML document – valid according to the described XSD schema - continues with the following sequence of XML elements: exactly one `Config_Frame` element, one or more `Inport_Frame` elements, one or more `Outport_Frame` elements, exactly one `MW_Config_Frame` element, and one or more `Job_Config` elements.

The `Config_Frame` element comprises five mandatory attributes. The first attribute (*type*) is a constant value always containing "FT_RMA_TNA_CONFIG". This is replaced by the Initialization Module of the TNA software (cf. Section 6.3.1 on page 6.3.1) with the numerical constant representing this frame type (the type of the *Config_Frame* is identified by $0x78h$). The *host_sysmode* attribute is a 16 bit value that is used to describe the initial operating mode of the host of the micro

Listing B.1: XML document showing initial configuration of an SoC

```
<?xml version="1.0" encoding="UTF-8"?>
<RMA_TNA_Msg xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<!-- ************************************************************ -->
<uC id="0">
  <Config_Frame status_slot="40960" wd_period="PERIOD_P8" update_po="38128"
                host_sysmode="1" type="FT_RMA_TNA_CONFIG"/>
  <OutPort_Frame type="FT_RMA_TNA_OUTPORT">
    <Port slotID="41020" channelID="0" length="64" type="TYPE_PERIODIC"  />
    <Port slotID="41010" channelID="0" length="256" type="TYPE_PERIODIC" />
    <Port slotID="41000" channelID="0" length="512" type="TYPE_PERIODIC" />
  </OutPort_Frame>
  <MW_Config_Frame update_po="38128" type="FT_RMA_TNA_MW"/>
    <Job_Config>
      <Job_Frame scheduling="0" DAS_ID="0" type="FT_RMA_TNA_JOB" jobID="0"/>
      <DECOS_Port_Frame portType="0" slotID="0" portID="0" portDir="0"
                        type="FT_RMA_TNA_DECOS_PORT" qLen="0" update_time="0"/>
    </Job_Config>
</uC>
<!-- ************************************************************ -->
<uC id="1">
  <InPort_Frame type="FT_RMA_TNA_INPORT">
    <Port slotID="41020" channelID="0" length="64" type="TYPE_PERIODIC" />
  </InPort_Frame>
  [...]
</uC>
<!-- ************************************************************ -->
<uC id="2">
  <InPort_Frame type="FT_RMA_TNA_INPORT">
    <Port slotID="41010" channelID="0" length="256" type="TYPE_PERIODIC" />
  </InPort_Frame>
  [...]
</uC>
<!-- ************************************************************ -->
<uC id="3">
  <InPort_Frame type="FT_RMA_TNA_INPORT">
    <Port slotID="41000" channelID="0" length="512" type="TYPE_PERIODIC" />
  </InPort_Frame>
  [...]
</uC>
[...]
</RMA_TNA_Msg>
```

component. With the attribute *wd_period* the frequency is defined, with which the host has to update its life-sign in order to prevent to be classified as failed. The valid values for the watchdog period are defined by an enumeration representing the periods defined for TTE (i.e., "PERIOD_P0" up to "PERIOD_P16"). The attributes *update_po* and *status_slot* define the periodic instants at which a new micro component configuration becomes valid and at which the TISS is permitted to disseminate diagnostic information, respectively.

The XML elements `Inport_Frame` and `Outport_Frame` are identical except to their *type* attribute, both are derived from the complex XML, type `Port_Frame`. The value of the type attribute of `Inport_Frame` is constant and comprises the string "FT_RMA_TNA_INPORT" whereas the type of `Outport_Frame` equals to the

constant value "FT_RMA_TNA_OUTPORT". Both XML elements comprise four attributes. The 16 bit value *slotID* defines the instant (i.e., period and phase offset) at which a message is received or sent on the TTE network. The valid values for the *type* attribute of the complex XML type `Port_Frame` are defined by an enumeration and restricted to "TYPE_PERIODIC" and "TYPE_SPORADIC". The *length* attribute defines the length of the message received/sent via the actual port. The length is defined in number of bytes of the message and restricted to the maximum length of a TTE message, which is 1488 Bytes.

The XML element *MW_Config_Frame* contains two attributes. The *type* attribute has the constant value "FT_RMA_TNA_MW", the *update_po* attribute holds a 16 bit value denoting the instant at which the configuration information contained in the XML element `Job_Config` becomes valid (*update_po* usually equals the reconfiguration period). The XML element `Job_Config` is composed of two XML elements. `Job_Frame` stores information about a particular job assigned to a micro component. It therefore comprises the attributes *type* (which equals the constant value "FT_RMA_TNA_JOB"), two 16 bit values that uniquely identify a job, as well as, a 64 bit long field, which can be exploited to derive scheduling constraints (e.g., activation time and deadline) of the job. The second XML element contained in `Job_Config` is `DECOS_Port_Frame`, which is used to establish the mapping between UFIM-Ports and concrete send/receive slot on the TTE network (i.e., SoC-Ports). Therefore, a logical 16 bit identifier *portID* can be specified which is assigned to a *slotID*. Further attributes of the element `DECOS_Port_Frame` are *portType* (event-triggered or time-triggered), *portDir* (input or output), *qLen* (length of an eventual message queue), and *update_time* (can be used for implicit synchronization of the host access to the port).

B.2 XML Representation of Protected Resources

A partial example of a valid resource protection document is depicted in Listing B.2. As defined in the XSD schema, the root element of a resource protection file is formed by the XML element `RP_Configuration`. For this XML element two attributes are defined. The mandatory attribute *version* of type xs:string holds information of the current version of the resource protection specification and can be exploited for tracking changes on the XML file. The second one is the optional string attribute *author*. For each micro component for which particular resources should be protected, an element `uC` has to be created. The individual XML elements `uC` are related to the physical micro components of the SoC component by the mandatory numerical attribute *id*. The resource protection specification of a single micro component continues with an optional sequence of `Port` specifications and/or an optional XML element `Host` for specifying protection information on the host configuration.

The XML element `Port` comprises six attributes. The attribute *direction* specifies whether the port is an input (DIR_INPUT) or an output (DIR_OUTPUT) port. The attributes *period* and *offset* are used to specify the time instant at which a

Listing B.2: Example of an XML resource protection file

```
<?xml version="1.0" encoding="UTF-8"?>
<RP_Configuration xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
                  version="0.1">
  <!-- ************************************************ -->
  <!-- Resource protection constraints for micro component X -->
  <!-- ************************************************ -->
  <uC id="0"> <!-- mandatory attribute denoting the uC id;
    <!-- ********************************************************** -->
    <!-- ********** Optional constraints regarding the ports ********** -->
    <!-- ********************************************************** -->
    <!-- period:  mandatory; one of [PERIOD_P0 | ... | PERIOD_P10]   -->
    <!-- length:  mandatory; int value with maximum length 1488      -->
    <!-- direction: mandatory; one of [DIR_OUTPUT | DIR_INPUT]       -->
    <!-- phase: optional; positive int value with                    -->
    <!--        maximum value 0xFFFFFF (in dec format)               -->
    <!-- type:  optional; one of [TYPE_PERIODIC | TYPE_SPORADIC]     -->
    <!-- channel  optional; positive int value with max. value 127   -->
    <Port period="PERIOD_P10" length="64" direction="DIR_OUTPUT"
          phase="60" type="TYPE_PERIODIC" channel="0"/>
    <Port length="256" direction="DIR_OUTPUT" period="PERIOD_P10"
          channel="0"/>
    <Port length="512" direction="DIR_OUTPUT" period="PERIOD_P10"
          phase="40"/>
    <!-- ********************************************************** -->
    <!-- ********** Optional constraints regarding the host ********** -->
    <!-- ********************************************************** -->
    <!-- sysMode: optional; positive int value with                  -->
    <!--          max. value 0xFFFF (in dec format)                  -->
    <!-- watchdogPeriod: optional; [PERIOD_P0 | ... | PERIOD_P10]    -->
    <Host sysMode="1" watchdogPeriod="PERIOD_P8"/>
  </uC>
  <!-- ************************************************ -->
  <!-- Resource protection constraints for micro component Y -->
  <!-- ************************************************ -->
  <uC id="1">
    [ ]
  </uC>
</RP_Configuration>
```

message is either sent or received via the shared TTE network. The *length* attribute allows the specification of the message length in bytes (up to 1488 bytes are permitted, which is in conformance to the maximum size of a standard Ethernet messages reduced by the size of the additional TTE header information). By the use of the attribute *type* one can specify whether the message should be handled as a periodic time-triggered message (TYPE_PERIODIC) or as a sporadic time-triggered message (TYPE_SPORADIC). While *period*, *length*, and *direction* are mandatory attributes of a *Port* element, the remaining ones are optional. Thereby, it is possible to specify the required bandwidth for a particular input or output port by specifying the *period* and the *length* of the message related to that port without restricting the concrete dissemination instant. The phase offset is not specified and is dynamically determined by the RMA.

The XML element *Host* comprises two attributes. The optional attribute *sysMode*

specifies the operational system mode at which the host has to reside over the entire time of operation of the SoC component. By specifying a value for the optional attribute *watchdogPeriod* the frequency with which the host has to send its life sign to the TISS is protected and cannot be changed during runtime.

Appendix C

List of Acronyms

AAM	Abstract Application Model
ACPI	Advanced Configuration and Power Interface
ADC	Analog-to-Digital Converter
AFDX	Avionics Full-Duplex Switched Ethernet
AIMS	Airplane Information Management System
APEX	APplication EXecutive
API	Application Programming Interface
AUTOSAR	Automotive Open System Architecture
BCU	Basic Connector Unit
BIU	Bus Interface Unit
CAN	Controller Area Network
CIM	Computation Independent Model
CMOS	Complementary Metal Oxide Semiconductor
CNI	Communication Network Interface
COEX	COre EXecutive
CP	Configuration and Planning
CSMA/CA	Carrier Sense Multiple Access / Collision Avoidance
DAS	Distributed Application Subsystem
DECOS	Dependable Embedded Components and Systems

DPM	Dynamic Power Management
DSE	Domain-Specific Editor
DU	Diagnostic Unit
DVFS	Dynamic Voltage and Frequency Scaling
ECU	Electronic Control Unit
E/E	Electric/Electronic
EEE	Encapsulated Execution Environment
ESP	Electronic Stability Program
FIM	Fully-specified Interface Model
FCR	Fault Containment Region
FIFO	First-In/First-Out
FPGA	Field Programmable Gate Array
GME	Generic Modeling Environment
GPS	Global Positioning System
GRAM	Grid Resource Access and Management
IFG	Inter Frame Gap
IMA	Integrated Modular Avionics
I/O	Input/Output
IP	Intellectual Property
LDAP	Lightweight Directory Access Protocol
LIF	Linking Interface
LIN	Local Interconnect Network
LRM	Line Replaceable Module
LRU	Line Replaceable Unit
LXRT	Linux Real-Time
MEDL	Message Descriptor List
MDA	Model Driven Architecture

MFIM	Macro FIM
MMU	Memory Management Unit
MOST	Media Orientated Systems Transport
MPSoC	Multi-Processor System-on-a-Chip
NBW	Non-Blocking Write Protocol
NoC	Network-on-a-Chip
OEM	Original Equipment Manufacturer
OMG	Object Management Group
ONA	Out-of-Norm Assertion
PAM	Physical Allocation Model
PCMCIA	Personal Computer Memory Card International Association
PIL	Platform Interface Layer
PIM	Platform Independent Model
PiP	Picture-in-Picture
PM	Platform Model
PMC	Power Manageable Component
PMS	Power-Managed Systems
PSM	Platform Specific Model
QoS	Quality of Service
RCLIF	Resource Coordination Linking Interface
RM	Resource Manager
RMA	Resource Management Authority
RMS	Resource Management System
RTAI	Real-Time Application Interface
RTE	Run Time Environment
SCU	Safety-Critical Connector Unit
SoC	System-on-a-Chip

SW-C	Software Component
TDMA	Time Division Multiple Access
TISS	Trusted Interface Subsystem
TMR	Triple Modular Redundancy
TNA	Trusted Network Authority
TTA	Time-Triggered Architecture
TTE	Time-Triggered Ethernet
TTSoC	Time-Triggered System-on-a-Chip
UFIM	Uniform FIM
UML	Unified Modeling Language
UNI	Uniform Network Interface
VFB	Virtual Function Bus
VIATRA	Visual Automated Transformations
XCU	Complex Connector Unit
XML	Extensible Markup Language
XSD	XML Schema Definition

Appendix D

Glossary

Abstract Application Model (AAM) The *Abstract Application Model* is an abstract representation of the system or a subsystem, where the interfaces of the individual subsystems are not fully specified, and thus, some design decisions are still left open (e.g., the selection of an adequate encryption method to achieve the desired security properties of a communication channel). In a subsequent step in the design process, the AAMs of all subsystems have to be transformed into a *FIM*, which includes the full specification of the *LIF* of each *job* of the system.

Application Computer The *application computer* is part of the *micro component*'s *host*. It provides the computational resources of the micro component and controls the micro component's local I/O interfaces (e.g., for sensors or actuators). It can be realized as a general-purpose microcontroller or FPGA or as a specialized hardware IP block (e.g., an MPEG encoder).

Application Service The *application service* is the intended sequence of messages that is produced by a *job* via output ports at the *LIF* and the *controlled object* interface in response to the progression of time, inputs (via input ports at the LIF and the controlled object interface), and state.

Architecture A technical *system architecture* (or architecture for short) is a framework for the construction of a system for a chosen application domain. It provides *core platform services* and imposes an *architectural style* for constraining an implementation in such a way that the ensuing system is understandable, maintainable, and extensible and can be built cost-effectively. (see also → *federated architecture*, → *integrated architecture*)

Architectural Style The architectural style consists of rules and guidelines for the partitioning of a system into *subsystems* and for the design of the interactions among subsystems. The subsystems must comply with the architectural style to avoid a property mismatch at the interfaces between subsystems.

Behavior The *behavior* of a subsystem is the sequence of messages (i.e., intended and unintended) that is produced by the subsystem at its *LIF*.

Cluster A *cluster* is a physically distributed computer system that consists of a set of *nodes* interconnected by a physical network. If the cluster supports a single DAS only, we speak of a federated cluster. In this case, the DAS is physically separated from the clusters of other DASs. Since the jobs belong to the same DAS, they possess a common level of criticality.
An *integrated* cluster, on the other hand, supports more than one DASs. Each of these DASs receives a share of the communication and component resources of the integrated cluster.

Controlled Object The *controlled object* is the industrial plant, the process, or the device that is to be controlled by the computer system.

Core Platform Services The *core platform services* (e.g., predictable transport of messages, global time service, watchdog service) are provided via the *UNI* and are independent of any particular *DAS*. They facilitate the development of distributed real-time applications and separate the application functionality from the underlying platform technology to reduce design complexity and to enable design reuse.
The core platform services can be adapted, refined, and extended by a *frontend*, which is a hardware element that translates the UNI to the interface of the attached *application computer*.

Declared State The *declared state* is the state of a subsystem, which is considered as relevant by the system designer for future behavior of the subsystem (forward view).

Diagnostic Unit (DU) The *Diagnostic Unit* is a dedicated *micro component* for the purpose of diagnosis. It performs failure detection at the *LIF* of the jobs by executing assertions on the syntactic, temporal, and semantic correctness of messages according to the DSoS message classification [Gaudel et al., 2002]. Furthermore, the DU receives failure indication messages generated by other architectural elements of the SoC (e.g., *hosts*, *TISSs*, the *TNA*, or the *RMA*). A failure indication message includes information concerning the type of the occurred failure (e.g., crash failure of a host), the time of detection w.r.t. to the global time base, and the location within the SoC (i.e., the micro component). Based on the gathered failure information, the DU establishes a holistic system view and executes *Out-of-Norm Assertions* (ONAs) to correlate the different failure indication messages in space and time.

Distributed Application Subsystem (DAS) A *Distributed Application Subsystem* is a nearly independent distributed subsystem of a large distributed real-time system that provides a well-specified application service. Examples of DASs in a present day automotive application are body electronics, the power-train system, and the multimedia system. Examples of DASs in a present day

avionic application are the cabin pressurization system, the fly-by-wire system, and the in-flight entertainment system. DASs are often developed by different organizational entities (e.g., by different vendors) and maintained by different specialists. Since DASs may be of different criticality (e.g., vehicle dynamics control vs. multimedia system), the probability of error propagation across DAS boundaries must be sufficiently low to meet the dependability requirements. A DAS is further decomposed into smaller units called *jobs*.

Event Message An *event message* is a message that contains event observations. An event observation contains the difference between the "old *state*" (the last observed state) and the "new state". The time of the event observation denotes the point in time of the state change. In order to maintain state synchronization, the handling of event messages requires exactly-once semantics. The arrival of an event message usually gives rise to a control signal, which triggers subsequent computational and communication activities.

Error Containment Although a *fault containment region* can demarcate the immediate impact of a fault, fault effects manifested as erroneous data can propagate across the boundaries of fault containment regions. For this reason the system must also provide *error containment* for avoiding error propagation by the flow of erroneous messages (→ *Error Containment Region*).

Error Containment Region The set of *fault containment regions* that performs error containment is denoted as an *error containment region*. An error containment region must consist of at least two independent fault containment regions. The error-detection mechanisms must be part of a different fault containment region than the message sender, otherwise the error detection service can be affected by the same fault that caused the message failure.

Fault Containment Region (FCR) A *Fault Containment Region* is a collection of components that operates correctly regardless of any arbitrary logical or electrical fault outside the region [Lala and Harper, 1994].

Fault Hypothesis The fault hypothesis is the specification of the *faults* that must be tolerated without any impact on the essential system services. The fault hypothesis states the assumptions about units of failure (→ *fault containment region*), failure modes, failure frequencies, failure detection, and *state recovery*.

Fault-Tolerant Unit (FTU) A unit consisting of a number of *replica determinate* micro components that provides the specified service even if some of its micro components fail.

Federated Architecture In a *federated architecture*, each DAS is implemented on a dedicated distributed computer system, consisting of nodes dedicated to jobs (in the automotive industry called Electronic Control Units - ECUs) and a physical network (e.g., a CAN network) among the nodes. In a federated architecture, each *DAS* is physically separated from other DASs, which leads to clear boundaries for responsibility and error propagation.

Frontend The *frontend* is part of the *micro component*'s *host*. It adapts, refines, or extends the *core platform services*, provided by the *UNI*, according to the requirements of the attached *application computer*. In its simplest version, the frontend is realized as a dual-ported memory providing a *temporal firewall interface* [Kopetz, 1997] to the application computer. If required, the frontend can provide *higher-level services*, which are tailored to the needs of specific application domains. Examples are a fault tolerance service which performs majority voting of replicated inputs for failure masking by TMR, or an encryption and decryption service to facilitate secure communication with chip external entities.

Fully-Specified Interface Model (FIM) The *Fully-specified Interface Model* describes the functionality and the interaction patterns of the individual jobs of a system by a behavioral specification, including temporal constraints. It does not include any information about the micro components on which the jobs will be executed and abstracts from micro component specific implementation details of the jobs (e.g., a micro component can be realized as a special purpose microcontroller, as an FPGA or as a special purpose hardware IP block).
The TTSoC architecture defines two different types of FIMs to describe a system at two different levels of abstraction, the UFIM and the MFIM.

Gateway Channel (G-channel) A *Gateway Channel* is an unidirectional communication channel that transports messages from a single source SoC to one or more destination SoCs (i.e., a channel between gateways of different SoCs).

Gateway Port (G-port) A *Gateway Port* is an endpoint of a *G-channel*.

Host The *host* performs the computations that are required to deliver the intended service of a *micro component*. It is structured into two architectural elements, the *application computer* and the *frontend*.

Integrated Architecture An *integrated architecture* is characterized by the integration of multiple *DASs* within a single distributed computer system. An integrated architecture possesses a single physical network that is exploited for the construction of multiple virtual networks. In the TTSoC architecture, architectural services are employed to encapsulate DASs and restore the complexity management advantages and natural error containment between DASs of a federated architecture.

Interface State The *interface state* contains the history of the component that is relevant for the future behavior of the component as seen from this interface. Interface state is defined between the intervals of activity on the *sparse time base*. Interface state is a subset of the state of the component and should be accessible from the interface.

Job A job is a subsystem of a *DAS* and the basic unit of distribution (i.e., a single job cannot be distributed on multiple micro components). It is the object of

temporal and spatial partitioning and interacts with other jobs solely by the exchange of messages through its *LIF*.
An example for a job in a safety-critical brake-by-wire DAS of a car would be the software, which fits into a single micro component, for computing the brake force based on the actual wheel slip. For fault-tolerance reasons, multiple instances of the job will be executed redundantly at different components, e.g., three instances in a triple-modular redundancy configuration.

Linking Interface (LIF) A *job* provides its real-time services, and accesses the real-time services of other jobs by the exchange of messages across its *Linking Interface*. These messages have to be fully specified in a *LIF specification* which consists of an *operational* specification and a *meta-level* specification [Kopetz and Suri, 2003]. While the operational specification deals with the syntactic and temporal aspects of the messages exchanged across the LIF, the meta-level specification describes the meaning of the information contained in these messages.

Macro FIM The *Macro FIM* is a high-level representation of the *FIM*. It facilitates the modeling of *DASs* at a higher level of abstraction than the *UFIM*, by providing macros that translate high-level constructs into constructs supported in the UFIM. Thus, the interface specification of the jobs in the MFIM can rely on higher-level domain-specific services like voted channels for fault tolerance, encrypted channels for security, or bidirectional channels for request/reply transactions. The *MFIM meta model* (→ *meta model*) can exist in multiple variations supporting different sets of domain-specific services. For each MFIM meta model, a set of transformation rules has to be specified that define the transformation of a MFIM to an equivalent UFIM.

Message Descriptor List (MEDL) The *Message Descriptor List* is a data structure within each *TISS* that determines when a message must be sent on, or received from, the NoC.

Micro Component A *micro component* is a self-contained computational unit that provides its functionality over a well defined *message-based* interface. It is composed out of two structural elements, the TISS and the *host*. While the host performs the computations, which are required to deliver the intended service of the micro component, the TISS provides a stable set of *core platform services* to the host. Furthermore, the TISS acts as a guardian for the NoC by ensuring that a fault within the host of a micro component cannot lead to a violation of the micro component's temporal interface specification in a way that the communication between other micro components would be disrupted.

MFIM-channel *MFIM channels* are used to describe the communication between *jobs* in the *MFIM*. Contrary to the *UFIM-channels* in the *UFIM*, MFIM channels are not restricted to be unidirectional. Their characteristics are determined by the chosen MFIM meta model (→ *meta model*).

MFIM-port An *MFIM port* is an endpoint of an *MFIM-channel*. A *job* in the *MFIM* can have multiple MFIM-ports since it can be attached to multiple MFIM-channels.

Non-Blocking Write Protocol (NBW) The *Non-Blocking Write Protocol* is a synchronization protocol between a single writer and many readers. It achieves data consistency without blocking the writer [Kopetz, 1997].

Out-of-Norm Assertion (ONA) An *Out-of-Norm Assertion* detects anomalous component behavior that cannot be judged as correct or faulty at the time of occurrence. Out-of-norm assertions operate on the output messages and the interface state and encode fault patterns on the consistent distributed state induced by a sparse time base and are specified in the dimensions of value, time and space.

Meta Model A *meta model* defines the rules and constructs according to which a model is created.

Physical Allocation Model (PAM) The *Physical Allocation Model* is a more concrete system representation than the FIM. It describes the mapping of the FIM (to be more precise, the UFIM) to the physical system structure. Contrary to the UFIM, the PAM is tailored to the specific characteristics of the micro component on which a job should be executed. Nevertheless, the semantic and syntactic properties of a job's LIF in the PAM are exactly the same as in the UFIM. The temporal properties of a job's LIF in the PAM are fully specified and satisfy the temporal constraints defined in the UFIM.

Platform-Independent Model (PIM) A *Platform Independent Model* is a model of a system that is independent of the specific technological platform used to implement it.

Platform-Specific Model (PSM) A *Platform Specific Model* is a model of system that is linked to a specific technological platform.

Replica Determinism *Replica determinism* is a desired property between replicated subsystems. A set of replicated subsystems is replica determinate if all subsystems in this set produce exactly the same output messages that are at most an interval of d time units apart, as seen by an omniscient outside observer. In a time-triggered system, the subsystems are considered to be replica-deterministic if they produce the same output messages at the same global ticks of their local clock [Kopetz, 1997].

Resource Management Authority (RMA) The *Resource Management Authority* is, besides the *TNA*, one of the two dedicated architectural elements for resource management. It accepts *resource request messages* from the jobs and generates, according to internal rules, a resource allocation mapping for the entire SoC.

SoC-Channel The term *SoC-Channel* denotes an encapsulated unidirectional communication channel in the physical system structure that transports messages at predefined points in time from a single source micro component to one or more destination micro components within the same SoC (SoC-Channels cannot cross the boundaries of a single SoC).

SoC-Port An *SoC-port* is an endpoint of an *SoC-channel*. A *micro component* can have multiple SoC-Ports since it can be attached to multiple SoC-channels.

Sparse Time Base If the time base of the global time in a distributed system is *dense* (i.e., the events are allowed to occur at any instant of the time line), then it is in general not possible to generate a consistent temporal order of events on the basis of the time-stamps. Due to the impossibility of synchronizing clocks perfectly and the denseness property of real time, there is always the possibility that a single event is time-stamped by two clocks with a difference of one tick. By introducing the concept of a *sparse time base* [Kopetz, 1992] this problem can be solved. In the sparse time model the continuum of time is partitioned into an infinite sequence of alternating durations of activity (π) and silence (Δ). Thereby, the occurrence of significant events is restricted to the activity intervals of a globally synchronized action lattice. In this time model, the costly execution of agreement protocols can be avoided, since every action is delayed until the next lattice point of the action lattice.

State The *state* enables the determination of a future output solely on the basis of the future input and the state the system is in. In other word, the state enables a "decoupling" of the past from the present and future. The state embodies all past history of the given system. Apparently, for this role to be meaningful, the notion of the past and future must be relevant for the system considered (taken from [Mesarovic and Takahara, 1989, p. 45]) ($\rightarrow$ *declared state*, $\rightarrow$ *interface state*).

State Message A *state message* is a periodic message that contains state observations. An observation is a state observation, if the value of the observation contains the state of a real-time entity. The time of a state observation denotes the point in time when the real-time entity was sampled. The handling of state messages occurs through an update in place and non-consuming read.

State Recovery *State recovery* is the action of (re)establishing a valid *state* in a *subsystem* after a *failure* of that subsystem.

Subsystem A subsystem is a part of a system that represents a closure with respect to a given property.

Trusted Interface Subsystem (TISS) The *Trusted Interface Subsystem* is part of the *micro component* and provides a stable set of *core platform services* via the *UNI*. Furthermore, it acts as a guardian for the NoC by ensuring that a fault within the *host* of a micro component (e.g., a software fault) cannot

lead to a violation of the micro component's temporal interface specification in a way that the communication between other micro components would be disrupted.

Trusted Network Authority (TNA) The *Trusted Network Authority* is, besides the *RMA*, one of the two dedicated architectural elements for resource management and is responsible for the (re)configuration of the *TISSs* and the NoC. It checks the resource allocation proposal, provided by the RMA, against a set of predefined constraints (e.g., conflict-freeness of the message schedule or availability of statically assigned resources for safety-critical application subsystems). If the mapping is valid, the TNA (re)configures the NoC and the TISSs accordingly.

Trusted Subsystem (TSS) The *Trusted Subsystem* consists of the *TNA*, the time-triggered NoC, and the *TISSs*. The TSS is assumed to be free of design faults and has to be certified according the criticality level of the most critical micro component in the SoC.

Uniform Network Interface (UNI) The *Uniform Network Interface* is the basic architectural interface of the TTSoC architecture. It is located between the *TISS* and the *host*. The UNI provides a set of *core platform services* which facilitate the development of distributed real-time applications and separate the application functionality from the underlying platform technology to reduce design complexity and to enable design reuse.

Uniform FIM (UFIM) The *Uniform FIM* is an uniform representation of the *FIM*. It describes the system at the level of the *UNI*. This means that, with respect to the interface specification of the jobs, the *UFIM meta model* (→ *meta model*) defines exclusively constructs that refer to the communication services that are natively provided by the UNI (e.g., unidirectional communication channels). The specification of a job in the UFIM serves as a contract between the *system integrator* and the *job developer* and can be used for conformance testing.

UFIM-channel *UFIM-channels* are used to describe the communication between *jobs* in the *UFIM*. The term UFIM-channel denotes an encapsulated unidirectional communication channel that transports messages under predefined temporal constraints (e.g., latency, period, absolute phase offset to the start of the period, or relative phase offset to another channel) from a single source job to one or more destination jobs.

UFIM-port A *UFIM-port* is an endpoint of a *UFIM-channel*. A *job* can have multiple UFIM-ports since it can be attached to multiple UFIM-channels.

Bibliography

[Agrawal et al., 2003] Agrawal, M., Cofer, D., and Samad, T. (2003). Real-time adaptive resource management for advanced avionics. *IEEE Control Systems Magazine*, 23(1):76–86.

[Almeida, 2006] Almeida, J. P. A. (2006). *Model-driven design of distributed applications*. PhD thesis, University of Twente, Enschede.

[ARINC, 1991a] ARINC (1991a). *ARINC Specification 629: Multi-Transmitter Data Bus – Part 1: Technical Description*. Aeronautical Radio, Inc., 2551 Riva Road, Annapolis, Maryland 21401.

[ARINC, 1991b] ARINC (1991b). *ARINC Specification 651: Design Guide for Integrated Modular Avionics*. Aeronautical Radio, Inc., 2551 Riva Road, Annapolis, Maryland 21401.

[ARINC, 1993] ARINC (1993). *ARINC Specification 659: Backplane Data Bus*. Aeronautical Radio, Inc., 2551 Riva Road, Annapolis, Maryland 21401.

[ARINC, 2001] ARINC (2001). *ARINC Specification 429: Digital Information Transfer System*. Aeronautical Radio, Inc., 2551 Riva Road, Annapolis, Maryland 21401.

[ARINC, 2002] ARINC (2002). *ARINC Specification 664: Aircraft Data Network Part 1 – Systems Concepts and Overview*. Aeronautical Radio, Inc., 2551 Riva Road, Annapolis, Maryland 21401.

[ARINC, 2003] ARINC (2003). *ARINC Specification 653-1 (Draft 3): Avionics Application Software Standard Interface*. Aeronautical Radio, Inc., 2551 Riva Road, Annapolis, Maryland 21401.

[Audsley and Wellings, 1996] Audsley, N. and Wellings, A. (1996). Analysing APEX applications. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96)*, pages 39–44, Washington, DC, USA. IEEE Computer Society.

[AUTOSAR GbR, 2006a] AUTOSAR GbR (2006a). *Methodology, Vers. 1.0.1*.

[AUTOSAR GbR, 2006b] AUTOSAR GbR (2006b). *Technical Overview, Vers. 2.01*.

[Banachowski and Brandt, 2003] Banachowski, S. A. and Brandt, S. A. (2003). Better real-time response for time-share scheduling. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 124.2, Washington, DC, USA. IEEE Computer Society.

[Benini et al., 2000] Benini, L., Bogliolo, A., and Micheli, G. D. (2000). A survey of design techniques for system-level dynamic powermanagement. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(3):299–316.

[Benini et al., 1999] Benini, L., Bogliolo., A., Paleologo, G. A., and Micheli, G. D. (1999). Policy optimization for dynamic power management. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):813–833.

[Benini et al., 2001] Benini, L., Micheli, G. D., and Macii, E. (2001). Designing low-power circuits: practical recipes. *IEEE Circuits and Systems Magazine*, 1(1):6–25.

[Bernholdt et al., 2005] Bernholdt, D., Bharathi, S., Brown, D., Chanchio, K., Chen, M., Chervenak, A., Cinquini, L., Drach, B., Foster, I., Fox, P., Garcia, J., Kesselman, C., Markel, R., Middleton, D., Nefedova, V., Pouchard, L., Shoshani, A., Sim, A., Strand, G., and Williams, D. (2005). The Earth system grid: supporting the next generation of climate modeling research. *Proceedings of the IEEE*, 93(3):485–495.

[Bihari and Schwan, 1991] Bihari, T. E. and Schwan, K. (1991). Dynamic adaptation of real-time software. *ACM Transactions on Computer Systems*, 9(2):143–174.

[Bosch, 1991] Bosch (1991). *CAN Specification, Version 2.0.* Robert Bosch Gmbh, Stuttgart, Germany.

[Brajou and Ricco, 2004] Brajou, F. and Ricco, P. (2004). The airbus A380 – an AFDX-based flight test computer concept. In *Proceedings of the IEEE Autotestcon*, pages 460–463, San Antonio, TX.

[Brown, 2004] Brown, A. (2004). An introduction to model driven architecture. Part I: MDA and today's systems. Available at: `http://www.ibm.com/developerworks/rational/library/3100.html`.

[Buyya et al., 2000] Buyya, R., Abramson, D., and Giddy, J. (2000). An Evaluation of Economy-based Resource Trading and Scheduling on Computational Power Grids for Parameter Sweep Applications. *Proceedings of the 2nd International Workshop on Active Middleware Services (AMS 2000)*, pages 221–230.

[Chandrakasan et al., 1992] Chandrakasan, A. P., Sheng, S., and Brodersen, R. W. (1992). Low-power CMOS digital design. *IEEE Journal of Solid-State Circuits*, 27(4):473–484.

[Chervenak et al., 2001] Chervenak, A., Foster, I., Kesselman, C., Salisbury, C., and Tuecke, S. (2001). The Data Grid: Towards an Architecture for the Distributed

Management and Analysis of Large Scientific Datasets. *Journal of Network and Computer Applications*.

[Coope, 1999] Coope, C. (1999). Using expat. Online available at: `http://www.xml.com/pub/a/1999/09/expat/index.html`.

[Csertán et al., 2002] Csertán, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., and Varró, D. (2002). VIATRA: Visual automated transformations for formal verification and validation of UML models. In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE'02)*, pages 267–270.

[DECOS, 2004] DECOS (2004). Report about decision on meta model and tools for PIM specification. DECOS project deliverable D1.1.1, Budapest University of Technology and Economics.

[DECOS, 2006] DECOS (2006). Report about DECOS System-on-a-Chip component prototype. DECOS project deliverable D2.2.5, Vienna University of Technology.

[Deicke, 2002] Deicke, A. (2002). The electrical/electronic diagnostic concept of the new 7 series. In *Convergence International Congress & Exposition On Transportation Electronics*, Detroit, MI, USA. SAE.

[Devadas and Malik, 1995] Devadas, S. and Malik, S. (1995). A survey of optimization techniques targeting low power VLSI circuits. In *Proceedings of 32nd ACM/IEEE Design Automation Conference*, pages 242–247.

[Driscoll and Hoyme, 1992] Driscoll, K. R. and Hoyme, K. P. (1992). The airplane information management system: an integrated real-time flight-deck control system. In *Proceedings of the Real-Time Systems Symposium*, volume 1, pages 267–270.

[El Salloum, 2007] El Salloum, C. (2007). *Interface Design in the Time-Triggered SoC Architecture*. PhD thesis, Vienna University of Technology, Institute of Computer Engineering, Treitlstr. 1-3/182-1, 1040 Vienna, Austria.

[El Salloum et al., 2006] El Salloum, C., Obermaisser, R., Huber, B., Kopetz, H., and Suri, N. (2006). Supporting heterogeneous applications in the DECOS integrated architecture. In *Proceedings of International DECOS Workshop at the Mikroelektroniktagung 2006*, pages 183–193, Vienna, Austria.

[Esterel Technologies, 2005] Esterel Technologies (2005). *SCADE Suite Technical and User Manuals, Version 5.0.1*.

[Feltovich et al., 2001] Feltovich, P. J., Coulson, R. L., and Spiro, R. J. (2001). Learners' (mis)understanding of important and difficult concepts: a challenge to smart machines in education. In *Smart machines in education: the coming revolution in educational technology*, pages 349–375. MIT Press, Cambridge, MA, USA.

[Fennel et al., 2006] Fennel, H., Bunzel, S., Harald Heinecke, J. B., Fürst, S., Schnelle, K.-P., Grote, W., Maldener, N., Weber, T., Wohlgemuth, F., Ruh, J., Lundh, L., Sandén, T., Heitkämper, P., Rimkus, R., Leflour, J., Gilberg, A., Virnich, U., Voget, S., anf Kazuhiro Kajio, K. N., Lange, K., Scharnhorst, T., and Kunkel, B. (2006). Achievements and exploitation of the AUTOSAR development partnership. *Convergence 2006*. SAE 2006-21-0019.

[FlexRay Consortium, 2005] FlexRay Consortium (2005). *FlexRay Communications System Protocol Specification Version 2.1*. BMW AG, DaimlerChrysler AG, General Motors Corporation, Freescale GmbH, Philips GmbH, Robert Bosch GmbH, and Volkswagen AG.

[Flynn and Hung, 2005] Flynn, M. J. and Hung, P. (2005). Microprocessor design issues: thoughts on the road ahead. *IEEE Micro*, 25(3):16–31.

[Foster et al., 2001] Foster, I., Kesselman, C., and Tuecke, S. (2001). The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, 15(3):200.

[Fraboul and Martin, 1998] Fraboul, C. and Martin, F. (1998). Modeling and simulation of integrated modular avionics. In *Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing (PDB'98)*, pages 102–110.

[Gaudel et al., 2002] Gaudel, M.-C., Issarny, V., Jones, C., Kopetz, H., Marsden, E., Moffat, N., Paulitsch, M., Powell, D., Randell, B., Romanovsky, A., Stroud, R., and Taiani, F. (2002). Final version of the DSoS conceptual model. *DSoS Project (IST-1999-11585) Deliverable CSDA1*. Available as Research Report 54/2002 at `http://www.vmars.tuwien.ac.at`.

[Gelsinger, 2001] Gelsinger, P. P. (2001). Microprocessors for the new millenium, challenges, opportunities, and new frontiers. In *Proceedings of the IEEE Solid State Circuit Conference (ISSCC'01)*, pages 22–25. IEEE Press.

[Ghosh et al., 1997] Ghosh, S., Melhem, R., and Mossé, D. (1997). Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(3):272–284.

[Giusto et al., 2002] Giusto, P., Ferrari, A., Lavagno, L., Brunel, J.-Y., Fourgeau, E., and Sangiovanni-Vincentelli, A. (2002). Automotive virtual integration platforms: why's, what's, and how's. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 370–378.

[Goossens et al., 2005] Goossens, K., Dielissen, J., and Radulescu, A. (2005). The Æthereal network on chip: Concepts, architectures, and implementations. *IEEE Design and Test of Computers*, 22(5):414–421.

[Gruhn et al., 2006] Gruhn, V., Pieper, D., and Röttgers, C. (2006). *MDA. Effektives Software-Engineering mit UML 2 und Eclipse*. Springer Berlin/Heidelberg.

[Halbwachs et al., 1991] Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. (1991). The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79:1305–1320.

[Hammett, 2003] Hammett, R. (2003). Flight-critical distributed systems – design considerations. *IEEE Aerospace and Electronic Systems Magazine*, 18(6):30–36.

[Hanselman and Littlefield, 2001] Hanselman, D. C. and Littlefield, B. (2001). *Mastering MATLAB 6: a comprehensive tutorial and reference*. "Prentice-Hall", "Upper Saddle River, NJ 07458, USA".

[Heinecke et al., 2006] Heinecke, H., Bielefeld, J., Schnelle, K.-P., Maldener, N., Fennel, H., Weis, O., Weber, T., Ruh, J., Lundh, L., Sandeén, T., Heitkämper, P., Rimkus, R., Leflour, J., Gilberg, A., Virnich, U., Voget, S., Nishikawa, K., Kajio, K., Scharnhorst, T., and Kunkel, B. (2006). AUTOSAR-Current results and preparations for exploitation. *7th EUROFORUM conference" Software in the vehicle"*.

[Heinecke et al., 2004] Heinecke, H., Schnelle, K.-P., Fennel, H., Bortolazzi, J., Lundh, L., Leflour, J., Maté, J.-L., Nishikawa, K., and Scharnhorst, T. (2004). AUTomotive Open System ARchitecture—An Industry-Wide Initiative to Manage the Complexity of Emerging Automotive E/E Architectures. *Convergence 2004, Proceedings of the 2004 International Congress on Transportation Electronics, SAE/P-387*, pages 325–332. SAE-2004-21-0042.

[Herzner et al., 2007] Herzner, W., Schlick, R., Schlager, M., Leiner, B., Huber, B., Balogh, A., Csertán, G., LeGuennec, A., LeSergent, T., Suri, N., and Islam, S. (2007). Model-based development of distributed embedded real-time systems with the DECOS tool-chain. In *Proceedings of the 2007 AeroTech Exhibition and Congress*, Los Angeles, CA, USA.

[Hewlett-Packard Corp. et al., 2006] Hewlett-Packard Corp., Intel Corp., Microsoft Corp., Phoenix Technologies Ltd., and Toshiba Corp. (2006). *Advanced Configuration and Power Interface Specification. Revision 3.0b.*

[Hoyme and Driscoll, 1993] Hoyme, K. P. and Driscoll, K. R. (1993). SAFEbus. *IEEE Aerospace and Electronic Systems Magazine*, 8:34–39.

[Huber and Obermaisser, 2007] Huber, B. and Obermaisser, R. (2007). Model-based development of integrated computer systems: Modeling the execution platform. In *Proceedings of the 5th Workshop on Intelligent Solutions in Embedded Systems (WISES'07)*, Madrid, Spain.

[Huber et al., 2006] Huber, B., Obermaisser, R., and Peti, P. (2006). MDA-Based development in the DECOS integrated architecture - modeling the hardware platform. In *Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-time Distributed Computing (ISORC'06)*, pages 43–52, Gyeongju, Korea.

[Huber et al., 2005] Huber, B., Peti, P., Obermaisser, R., and Salloum, C. E. (2005). Using RTAI/LXRT for partitioning in a prototype implementation of the DECOS architecture. In *Proceedings of the 3rd Workshop on Intelligent Solutions in Embedded Systems (WISES'05)*, Hamburg, Germany.

[IEEE, 2000] IEEE (2000). *IEEE Standard 802.3, 2000 Edition. Carrier Sense Multiple Access with Collision Detect on (CSMA/CD) Access Method and Physical Layer Specifications*. IEEE.

[Infineon, 2005] Infineon (2005). *TC1796, 32-Bit Single-Chip Microcontroller, TriCore. Datasheet Vers. 0.3*. Available at `ftp://ftp.efo.ru/pub/infineon/TC179_5Fds_5Fv03.pdf`.

[Intel, 2000] Intel (2000). *Intel XScale Microarchitecture, Technical Summary*. Available at `http://download.intel.com/design/intelxscale/XScaleDatasheet4.pdf`.

[Islam et al., 2006] Islam, S., Csertá, G., Herzner, W., LeSergent, T., Pataricza, A., and Suri, N. (2006). A SW-HW integration process for the generation of platform specific models. In *Proceedings of International DECOS Workshop at the Mikroelektroniktagung 2006*, pages 194–203, Vienna, Austria.

[Johnson et al., 1999] Johnson, M. C., Somasekhar, D., and Roy, K. (1999). Models and algorithms for bounds on leakage in CMOS circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):714–725.

[Kakumu and Kinugaw, 1990] Kakumu, M. and Kinugaw, M. (1990). Power-supply voltage impact on circuit performance for half andlower submicrometer CMOS LSI. *IEEE Transactions on Electron Devices*, 37(8):1902–1908.

[Kang, 2003] Kang, S.-M. S. (2003). Elements of low power design for integrated systems. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED '03)*, pages 205–210.

[Keshavarzi et al., 1997] Keshavarzi, A., Roy, K., and Hawkins, C. F. (1997). Intrinsic leakage in low power deep submicron CMOS ICs. In *In Proceedings of IEEE International Test Conference*, pages 146–155.

[Kleinrock, 1975] Kleinrock, L. (1975). *Queuing Systems Volume I: Theory*. John Wiley and Sons, New York.

[Kopetz, 1992] Kopetz, H. (1992). Sparse time versus dense time in distributed real-time systems. In *Proceedings of 12th International Conference on Distributed Computing Systems*, Japan.

[Kopetz, 1997] Kopetz, H. (1997). *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, MA, USA.

[Kopetz, 2003] Kopetz, H. (2003). Fault containment and error detection in the Time-Triggered Architecture. In *Proceedings of the The Sixth International Symposium on Autonomous Decentralized Systems (ISADS'03)*, pages 139–146, Washington, DC, USA. IEEE Computer Society.

[Kopetz, 2005] Kopetz, H. (2005). A time-triggered SoC-platform for distributed embedded application. Technical Report 34, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria.

[Kopetz, 2006] Kopetz, H. (2006). Pulsed data streams. In *IFIP TC 10 Working Conference on Distributed and Parallel Embedded Systems (DIPES 2006)*, pages 105–124, Braga, Portugal. Springer.

[Kopetz, 2007] Kopetz, H. (2007). The complexity challenge in embedded system design. Research Report 55/2007, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria.

[Kopetz et al., 2005] Kopetz, H., Ademaj, A., Grillinger, P., and Steinhammer, K. (2005). The Time-Triggered Ethernet (TTE) design. In *Proceedings of the 8th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC'05)*, Seattle, USA.

[Kopetz et al., 2006] Kopetz, H., Ademaj, A., Grillinger, P., and Steinhammer, K. (2006). *Time-Triggered Ethernet Protocol Specification. Version 0.91.* Vienna University of Technology, Institute of Computer Engineering, Treitlstr. 1-3/182-1, 1040 Vienna, Austria.

[Kopetz and Bauer, 2003] Kopetz, H. and Bauer, G. (2003). The Time-Triggered Architecture. *Proceedings of the IEEE*, 91(1):112 – 126.

[Kopetz and Grünsteidl, 1994] Kopetz, H. and Grünsteidl, G. (1994). TTP – a protocol for fault-tolerant real-time systems. *Computer*, 27(1):14–23.

[Kopetz and Obermaisser, 2002] Kopetz, H. and Obermaisser, R. (2002). Temporal composability. *Computing & Control Engineering Journal*, 13:156–162.

[Kopetz et al., 2004] Kopetz, H., Obermaisser, R., Peti, P., and Suri, N. (2004). From a federated to an integrated architecture for dependable embedded real-time systems. Technical Report 22, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria.

[Kopetz and Ochsenreiter, 1987] Kopetz, H. and Ochsenreiter, W. (1987). Clock synchronization in distributed real-time systems. *IEEE Transactions on Computers*, 36(8):933–940.

[Kopetz and Suri, 2003] Kopetz, H. and Suri, N. (2003). Compositional design of RT systems: A conceptual basis for specification of linking interfaces. In *Proceedings of the Sixth IEEE Int. Symposium on Object-Oriented Real-Time Distributed Computing*, pages 51–60. IEEE.

[Krauter et al., 2002] Krauter, K., Buyya, R., and Maheswaran, M. (2002). A taxonomy and survey of grid resource management systems for distributed computing. *Software Practice and Experience*, 32(2):135–164.

[Lala and Harper, 1994] Lala, J. H. and Harper, R. E. (1994). Architectural principles for safety-critical real-time applications. *Proceedings of the IEEE*, 82:25–40.

[Ledeczi et al., 2001] Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., and Volgyesi, P. (2001). The Generic Modeling Environment. In *Workshop on Intelligent Signal Processing, Budapest, Hungary*, volume 17.

[Lee et al., 1998] Lee, Y.-H., Kim, D., Younis, M., and Zhou, J. (1998). Partition scheduling in APEX runtime environment for embedded avionics software. In *Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications (RTCSA'98)*, pages 103–109, Washington, DC, USA. IEEE Computer Society.

[Leiner et al., 2007] Leiner, B., Schlager, M., Obermaisser, R., and Huber, B. (2007). A comparison of partitioning operating systems for integrated systems. In *Proceedings of the 26th International Conference on Computer Safety, Reliability and Security (SAFECOMP'07)*, Nuremberg, Germany.

[LIN, 2003] LIN (2003). *LIN Specification Package Revision 2.0*. LIN Consortium.

[Liu and Layland, 1973] Liu, C. L. and Layland, J. W. (1973). Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61.

[Lorch and Smith, 1998] Lorch, J. R. and Smith, A. J. (1998). Software strategies for portable computer energy management. *IEEE Personal Communications*, 5(3):60–73.

[LynuxWorks, 2007] LynuxWorks (2007). *LynxOS-178 2.0. Certifiable OS for safety-critical computing*. LynuxWorks Inc., 855 Embedded Way, San Jose, California.

[Lyons and Vanderkulk, 1962] Lyons, R. E. and Vanderkulk, W. (1962). The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, 6(2):200–209.

[Mantegazza et al., 2000] Mantegazza, P., Bianchi, E., Dozio, L., Papacharalambous, S., Hughes, S., and Beal, D. (2000). RTAI: Real-Time Application Interface. *Linux Journal*. Available at: `http://www.linuxjournal.com/article/3838`.

[Mellor et al., 2002] Mellor, S. J., Scott, K., Uhl, A., and Weise, D. (2002). Model-driven architecture. In *Proceedings of the Advances in Object-Oriented Information Systems Workshop (OOIS'02)*, pages 233–239, Montpellier, France. Springer Berlin / Heidelberg.

[Mesarovic and Takahara, 1989] Mesarovic, M. D. and Takahara, Y. (1989). *Abstract systems theory*. Springer.

[Morgan, 1991] Morgan, M. J. (1991). Integrated modular avionics for next generation commercial airplanes. *IEEE Aerospace and Electronic Systems Magazine*, 6:9–12.

[MOST Cooperation, 2002] MOST Cooperation (2002). *MOST Specification Version 2.2*. MOST Cooperation, Karlsruhe, Germany.

[Murray, 2003] Murray, C. J. (2003). Auto group seeks universal software. *EE Times*. Available at `http://www.eetimes.com/showArticle.jhtml?articleID=18309903`.

[Murthy and Manimaran, 2001] Murthy, C. S. R. and Manimaran, G. (2001). *Resource Management in Real-Time Systems and Networks*. MIT Press, Cambridge, MA, USA.

[Nilsson et al., 1998] Nilsson, U., Streiffert, S., and Torne, A. (1998). Detailed design of avionics control software. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS'98)*, pages 82–91, Madrid, Spain.

[Obermaisser et al., 2007] Obermaisser, R., Kopetz, H., Salloum, C. E., and Huber, B. (2007). Error containment in the time-triggered system-on-a-chip architecture. In *Proceedings of the International Embedded Systems Symposium (IESS'07)*, Irvine, CA, USA.

[Obermaisser et al., 2006] Obermaisser, R., Peti, P., Huber, B., and Salloum, C. E. (2006). DECOS: An integrated time-triggered architecture. *e&i journal (journal of the Austrian professional institution for electrical and information engineering)*, 3:83–95.

[Obermaisser et al., 2005a] Obermaisser, R., Peti, P., and Kopetz, H. (2005a). Virtual gateways in the DECOS integrated architecture. In *Proceedings of the Workshop on Parallel and Distributed Real-Time Systems 2005 (WPDRTS)*. IEEE.

[Obermaisser et al., 2005b] Obermaisser, R., Peti, P., and Kopetz, H. (2005b). Virtual networks in an integrated time-triggered architecture. In *Proceedings of the 10th IEEE Int. Workshop on Object-oriented Real-time Dependable Systems (WORDS2005)*, pages 241–253, Sedona, Arizona.

[OMG, 2002] OMG (2002). Smart Transducers Interface. Specification ptc/2002-05-01, Object Management Group. Available at `http://www.omg.org/`.

[OMG, 2003] OMG (2003). MDA Guide Version 1.0.1. Technical Report document number omg/2003-06-01, Object Management Group. Available at: `http://www.omg.org/docs/omg/03-06-01.pdf`.

[OMG, 2004] OMG (2004). Common Object Request Broker Architecture: Core Specification. Version 3.0.3. Technical Report formal/04-03-12, Object Management Group. Available at: `http://www.omg.org/docs/formal/04-03-12.pdf`.

[OMG, 2007] OMG (2007). Unified modeling language: Superstructure. version 2.1.1. Technical Report formal/2007-02-05, Object Management Group. Available at: `http://www.omg.org/docs/formal/07-02-05.pdf`.

[Owens, 2004] Owens, J. (2004). GPUs: Engines for future high-performance computing. Technical report, Lincoln Labs, Boston.

[Parkinson and Kinnan, 2006] Parkinson, P. and Kinnan, L. (2006). *Safety-Critical Software Development for Integrated Modular Avionics*. Wind River Systems Inc., Alameda, California.

[Pauli et al., 1998] Pauli, B., Meyna, A., and Heitmann, P. (1998). Reliability of electronic components and control units in motor vehicle applications. In *VDI Berichte 1415, Electronic Systems for Vehicles*, pages 1009–1024. Verein Deutscher Ingenieure.

[Pedram, 1996] Pedram, M. (1996). Power minimization in IC design: principles and applications. *ACM Transactions on Design Automation of Electronic Systems*, 1(1):3–56.

[Poledna, 1994] Poledna, S. (1994). Replica determinism in distributed real-time systems: A brief survey. *Real-Time Systems*, 6:289–316.

[Prisaznuk, 1992] Prisaznuk, P. J. (1992). Integrated modular avionics. In *Proceedings of the IEEE 1992 National Aerospace and Electronics Conference (NAECON'92)*, volume 1, pages 39–45.

[Ramamritham and Stankovic, 1994] Ramamritham, K. and Stankovic, J. A. (1994). Scheduling algorithms and operating systems support for real-time systems. *Proceedings of the IEEE*, 82(1):55–67.

[Ramsey, 2007] Ramsey, J. W. (2007). Integrated Modular Avionics: Less is More. *Avionics Magazine*. Available at `http://www.aviationtoday.com/av/categories/commercial/8420.html`.

[Randell et al., 1978] Randell, B., Lee, P. A., and Treleaven, P. C. (1978). Reliability issues in computing system design. *ACM Computing Surveys*, 10(2):123–165.

[Rong and Pedram, 2006] Rong, P. and Pedram, M. (2006). Power-aware scheduling and dynamic voltage setting for tasks running on a hard real-time system. *Proceedings of the 2006 conference on Asia South Pacific design automation*, pages 473–478.

[RTCA, 1992] RTCA (1992). *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*. Radio Technical Commission for Aeronautics, Inc. (RTCA), Washington, DC.

[Rubini and Corbet, 1998] Rubini, A. and Corbet, J. (1998). *Linux Device Drivers (Nutshell Handbook)*. O'Reilly.

[Rushby, 1999] Rushby, J. (1999). Partitioning for avionics architectures: Requirements, mechanisms, and assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center. Also to be issued by the FAA.

[Rushby, 2001] Rushby, J. (2001). Bus architectures for safety-critical embedded systems. In Henzinger, T. and Kirsch, C., editors, *Proceedings of the First Workshop on Embedded Software (EMSOFT'01)*, volume 2211 of *Lecture Notes in Computer Science*, pages 306–323, Lake Tahoe, CA, USA. Springer-Verlag.

[Sangiovanni-Vincentelli and Martin, 2001] Sangiovanni-Vincentelli, A. and Martin, G. (2001). Platform-based design and software design methodology for embedded systems. *IEEE Design and Test of Computers*, 18(6):23–33.

[Scharnhorst et al., 2005] Scharnhorst, T., Heinecke, H., Schnelle, K.-P., Fennel, H., Bortolazzi, J., Lundh, L., Heitkämper, P., Leflour, J., Maté, J.-L., and Nishikawa, K. (2005). AUTOSAR-Challenges and Achievements 2005. *VDI BERICHTE*, 1907:395.

[Scheidler et al., 2000] Scheidler, C., Puschner, P., Boutin, S., Fuchs, E., Grünsteidl, G., Papadopoulos, Y., Pisecky, M., Rennhack, J., and Virnich, U. (2000). Systems engineering of timetriggered architectures – the SETTA approach. In *Proceedings of the 16th IFAC Workshop on Distributed Computer Control Systems (DCCS'00)*, pages 55–60.

[Shen et al., 1993] Shen, C., Ramamritham, K., and Stankovic, J. A. (1993). Resource reclaiming in multiprocessor real-time systems. *IEEE Transactions of Parallel Distributed Systems*, 4(4):382–397.

[SIA, 2005] SIA (2005). International technology roadmap for semiconductors - executive summary. Technical report, Semiconductor Industry Association (SIA).

[Sifakis, 2005] Sifakis, J. (2005). A framework for component-based construction. In *Proceedings of 3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM'05)*, pages 293–300.

[Sonics, 2002] Sonics (2002). Sonics uNetwork technical overview. Available at: `http://www.sonicsinc.com/`.

[Stallings, 1998] Stallings, W. (1998). *Operating systems (3rd ed.): Internals and Design Principles*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[Stankovic and Ramamritham, 1991] Stankovic, J. A. and Ramamritham, K. (1991). The spring kernel: A new paradigm for real-time systems. *IEEE Software*, 8(3):62–72.

[Steinhammer, 2006] Steinhammer, K. (2006). *Design of an FPGA-based Time-Triggered Ethernet System*. PhD thesis, Vienna University of Technology, Institute of Computer Engineering, Treitlstr. 1-3/182-1, 1040 Vienna, Austria.

[Steinhammer, 2007] Steinhammer, K. (2007). *TT Ethernet Communication Controller IP. Preliminary Draft.* Vienna University of Technology, Institute of Computer Engineering, Treitlstr. 1-3/182-1, 1040 Vienna, Austria.

[Steinhammer et al., 2006] Steinhammer, K., Grillinger, P., Ademaj, A., and Kopetz, H. (2006). A time-triggered Ethernet (TTE) switch. In *Proceedings of 2006th Conference on Design, Automation and Test in Europe (DATE'06)*, Munich, Germany.

[Suri et al., 1995] Suri, N., Walter, C. J., and Hugue, M. M. (1995). *Advances In Ultra-Dependable Distributed Systems*, chapter 1. IEEE Computer Society Press, 10662 Los Vaqueros Circle, P.O. Box 3014, Los Alamitos, CA 90720-1264.

[Swanson, 1998] Swanson, D. L. (1998). Evolving avionics systems from federated to distributed architectures. In *Proceedings of the 17th AIAA/IEEE/SAE Digital Avionics Systems Conference*, volume 1, pages D26/1–D26/8, Bellevue, WA,USA.

[Swingler and McBride, 1998] Swingler, J. and McBride, J. W. (1998). The synergistic relationship of stresses in the automotive connector. In *Proceedings of the 19th International Conference on Electric Contact Phenomena*, pages 141–145.

[TT-SoC, 2007a] TT-SoC (2007a). Design of the NoC-FPGA. Project Deliverable D2.2, Vienna University of Technology, Institute of Computer Engineering, Vienna, Austria.

[TT-SoC, 2007b] TT-SoC (2007b). Requirements and overall system architecture. Project Deliverable D1.1, Vienna University of Technology, Institute of Computer Engineering, Vienna, Austria.

[Unsal and Koren, 2003] Unsal, O. S. and Koren, I. (2003). System-level power-aware design techniques in real-time systems. *Proceedings of the IEEE*, 91(7):1055–1069.

[Veríssimo and Rodrigues, 2001] Veríssimo, P. and Rodrigues, L. (2001). *Distributed Systems for System Architects.* Kluwer Academic Publishers, Norwell, MA, USA.

[Wolf, 2004] Wolf, W. (2004). The future of multiprocessor systems-on-chips. In *Proceedings of the 41st annual conference on Design automation (DAC '04)*, pages 681–685, New York, NY, USA. ACM Press.

[Yano et al., 1990] Yano, K., Yamanaka, T., Nishida, T., Saito, M., Shimohigashi, K., and Shimizu, A. (1990). A 3.8-ns CMOS 16×16-b multiplier using complementary pass-transistor logic. *IEEE Journal of Solid-State Circuits*, 25(2):388–395.

[Yeo and Buyya, 2007] Yeo, C. S. and Buyya, R. (2007). Pricing for utility-driven resource management and allocation in clusters. *International Journal of High Performance Computing Applications.*

[Zhao, 2005] Zhao, L. (2005). Designing application domain models with roles. In Aßmann, U. and Aksit, M., editors, *Model Driven Architecture*, volume 3599/2005 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin/Heidelberg.

List of Publications

[1] Bernhard Huber and Wilfried Elmenreich. Wireless time-triggered real-time communication. In *Proceedings of the 2nd Workshop on Intelligent Solutions in Embedded Systems (WISES'04)*, Graz, Austria, June 2004.

[2] Bernhard Huber and Wilfried Elmenreich. Wireless time-triggered real-time communication. *Telematik Magazin*, 10(3/4):44–50, December 2004.

[3] Bernhard Huber, Philipp Peti, Roman Obermaisser, and Christian El Salloum. Using RTAI/LXRT for partitioning in a prototype implementation of the DECOS architecture. In *Proceedings of the 3rd Workshop on Intelligent Solutions in Embedded Systems (WISES'05)*, Hamburg, Germany, May 2005.

[4] Ingomar Wenzel, Raimund Kirner, Martin Schlager, Bernhard Rieder, and Bernhard Huber. Impact of dependable software development guidelines on timing analysis. In *Proceedings of the International Conference on Computer as a Tool (EUROCON)*, volume 1, pages 575–578, 2005.

[5] Bernhard Huber, Roman Obermaisser, and Philipp Peti. MDA-Based development in the DECOS integrated architecture - modeling the hardware platform. In *Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-time Distributed Computing (ISORC'06)*, pages 43–52, Gyeongju, Korea, April 2006.

[6] Christian El Salloum, Roman Obermaisser, Bernhard Huber, Hermann Kopetz, and Neeraj Suri. Supporting heterogeneous applications in the DECOS integrated architecture. In *Proceedings of the International DECOS Workshop at the Mikroelektroniktagung 2006*, pages 183–193, Vienna, Austria, October 2006.

[7] Wolfgang Herzner, Martin Schlager, Thierry Le Sergent, Bernhard Huber, Shariful Islam, Neeraj Suri, and András Balogh. From model-based design to deployment of integrated, embedded, real-time systems: The DECOS tool-chain. In *Proceedings of the International DECOS Workshop at the Mikroelektroniktagung 2006*, pages 204–213, Vienna, Austria, October 2006.

[8] Roman Obermaisser and Bernhard Huber. Model-based design of the communication system in an integrated architecture. In *Proceedings of the International*

Conference on Parallel and Distributed Computing and Systems (PDCS 2006), Dallas, USA, October 2006.

[9] Roman Obermaisser, Philipp Peti, Bernhard Huber, and Christian El Salloum. DECOS: An integrated time-triggered architecture. *e&i journal (journal of the Austrian professional institution for electrical and information engineering)*, 3:83–95, March 2006.

[10] Wolfgang Herzner, Bernhard Huber, György Csertan, and András Balogh. The DECOS tool-chain: Model-based development of distributed embedded safety-critical real-time systems. *ERCIM News*, 67:22–24, October 2006.

[11] Christian El Salloum, Roman Obermaisser, Bernhard Huber, Harald Paulitsch, and Hermann Kopetz. A time-triggered system-on-a-chip architecture with integrated support for diagnosis. In *Proceedings of the Design, Automation and Test in Europe (DATE'07)*, Nice, France, April 2007.

[12] Hermann Kopetz, Christian El Salloum, Bernhard Huber, and Roman Obermaisser. Periodic finite-state machines. In *Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-time Distributed Computing (ISORC'07)*, Santorini, Greece, May 2007.

[13] Hermann Kopetz, Roman Obermaisser, Christian El Salloum, and Bernhard Huber. Automotive software development for a multi-core system-on-a-chip. In *Proceedings of the 4th International ICSE Workshop on Software Engineering for Automotive Systems (SEAS'07)*, Minneapolis, USA, May 2007.

[14] Roman Obermaisser, Hermann Kopetz, Christian El Salloum, and Bernhard Huber. Error containment in the time-triggered system-on-a-chip architecture. In *Proceedings of the International Embedded Systems Symposium (IESS'07)*, Irvine, CA, USA, May 2007.

[15] Bernhard Huber and Roman Obermaisser. Model-based development of integrated computer systems: Modeling the execution platform. In *Proceedings of the 5th Workshop on Intelligent Solutions in Embedded Systems (WISES'07)*, Madrid, Spain, June 2007.

[16] Wolfgang Herzner, Rupert Schlick, Martin Schlager, Bernhard Leiner, Bernhard Huber, András Balogh, György Csertan, Alain LeGuennec, Thierry LeSergent, Neeraj Suri, and Shariful Islam. Model-based development of distributed embedded real-time systems with the DECOS tool-chain. In *Proceedings of the 2007 AeroTech Exhibition and Congress*, Los Angeles, CA, USA, September 2007.

[17] Bernhard Leiner, Martin Schlager, Roman Obermaisser, and Bernhard Huber. A comparison of partitioning operating systems for integrated systems. In *Proceedings of the 26th International Conference on Computer Safety, Reliability and Security (SAFECOMP'07)*, Nuremberg, Germany, September 2007.

Curriculum Vitae

Dipl.-Ing. Bernhard Huber

Edla 9
3261 Steinakirchen/Forst
phone: +43 699 11748949
email: mail.huberb@gmx.at

date of birth: 1980-02-21

EDUCATION

2004/10 – ongoing	**Vienna University of Technology – Institute of Computer Engineering Real-Time Systems Group** PhD thesis: “Resource Management in an Integrated Time-Triggered Architecture”.
1999/10 – 2004/10	**Vienna University of Technology** Studies of computer science. Master thesis: “Wireless Real-Time Communication for Smart Transducer Networks”. Graduation with distinction.
1994/09 – 1999/06	**Engineering School for Communications Engineering and Computer Science in St. Pölten**

PROFESSIONAL & PRACTICAL EXPERIENCE

2004/09 – ongoing	**Vienna University of Technology – Institute of Computer Engineering Real-Time Systems Group** Research and Teaching Assistant (fulltime) - Working on the European FP7 project GENESYS (Generic Embedded Systems Platform) - Working on the national FIT-IT project TT-SoC (Time-Triggered System-on-a-Chip) - Working on the European FP6 project DECOS (Dependable Embedded Components and Systems)
2002/10 – 2004/07	**Vienna University of Technology – Institute of Computer Engineering Embedded Computing Systems Group & Real-Time Systems Group** Study Assistant for - Embedded Systems Engineering - Electrical Engineering for Computer Engineering
2002/02 – 2002/09	**IS4O – Integrated Services for Organizations** - Server maintenance (Windows NT 4.0 Server, Microsoft Exchange) - Mail backend solutions (Free-BSD)
2001/08 – 2001/09	**Siemens Austria** Internship - Port of telecommunications software using object-oriented programming techniques
2000/02 – 2001/02	**Rotes Kreuz Österreich – Dienststelle St. Pölten** Civil service