FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

An Application Development Approach for the Time-Triggered System-on-Chip Architecture

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Bernhard Frömel
Matrikelnummer 0326077

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Peter Puschner
Mitwirkung: Projektass. Dipl.-Ing. Dr.techn. Christian El-Salloum

Wien, 01.08.2013

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

An Application Development Approach for the Time-Triggered System-on-Chip Architecture

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

by

Bernhard Frömel
Registration Number 0326077

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Peter Puschner
Assistance: Projektass. Dipl.-Ing. Dr.techn. Christian El-Salloum

Vienna, 01.08.2013

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Bernhard Frömel
Franz Rosskopfgasse 14, 3470 Kirchberg am Wagram

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Acknowledgements

I had the opportunity to conduct this thesis at the Institute of Computer Engineering, Vienna University of Technology.

I thank the advisor of the thesis, Prof. Dr. Peter Puschner. His invaluable comments and supportive guidance greatly contributed to the overall quality of this work. I also want to express my appreciation towards Prof. Dr. Hermann Kopetz who had fruitful input about meeting deadlines under challenging constraints.

Moreover, I want to thank my friend and colleague Roland Kammerer. His thorough proofreading considerably improved readability and quality of the thesis. The many discussions about real-time systems and our work on publications during the INDEXYS project prepared me well for this thesis.

Special thanks to Christian El-Salloum, Christian Paukovits and Bernhard Huber for the discussions and their thought provoking impulses during the TT-SoC project.

Finally, I like to give my gratitude to Maria Ochsenreiter for her good advice and for her excellent organizational support. Many thanks to Leo Mayerhofer who skillfully solved all the IT infrastructure challenges I presented to him during my work on the thesis.

— Bernhard Frömel
December 2013

Abstract

In the Time-Triggered System-on-Chip (TTSoC) Architecture *components* attached to a Time-Triggered Network-on-Chip (TTNoC) interact with each other via their Linking Interfaces (LIFs). This implies: *(1)* System behavior is determined by the interaction relations of its components and *(2)* component behavior is determined by the component LIF. Consequently, the definition of the interaction relations between components and the design of these component interfaces are the key to develop applications for the TTSoC Architecture. However, dynamic resource management provided by the architecture entails that both the component interfaces and the interaction relations are not necessarily static. Additionally, we are faced with a large available design space and the demand for denser integration: Todays chips offer hundreds of cores and for saving costs and other scaling benefits, we want to use them all.

Our goal is to formulate a generic application development approach that enables engineers to create distributed applications for the TTSoC Architecture.

To this end, we studied existing development methods intended for the Time-Triggered Architecture (TTA) and the general field of distributed systems. Based on these findings, we designed our development approach for the TTSoC Architecture and evaluated its viability in a case study. In our model-based development approach we defined a set of *service-oriented* models that represent the system at multiple abstraction levels. A series of model-transformations refine the system until it can be executed on a platform (e.g., a TTSoC Architecture conforming chip). For this purpose, we implemented a tool that assists the model transformations and automatically derives platform configuration data (e.g., schedules, mapping information, ...) for TTSoC Architecture elements, i.e., TTNoC, dynamic resource management, diagnosis and gateways.

In support of the development approach, we advanced existing work concerning the TTSoC Architecture both conceptually and regarding the implementation. We applied a recursive component concept to the TTSoC Architecture, i.e., components can contain other components and can establish links between components. The case study validates the developed architectural extensions and supports the viability of the proposed development approach.

The recursive component concept facilitates the formation of systems such that they scale with respect to complexity, reliability, performance, and power. By making the TTSoC Architecture accessible for application development we hope to encourage its further scientific evolution, its use in university courses and its deployment in industry.

Kurzfassung

In der Time-Triggered System-on-Chip (TTSoC) Architektur sind *Komponenten* an ein Time-Triggered Network-on-Chip (TTNoC) angeschlossen und interagieren miteinander über ihr Linking Interface (LIF). Daraus folgt: *(1)* Das Systemverhalten resultiert aus den Interaktionsbeziehungen zwischen den Komponenten und *(2)* das Verhalten der Komponente wird durch ihr LIF bestimmt. Daher ist die Definition der Interaktionsbeziehungen zwischen den Komponenten und das Design der Komponentenschnittstellen der Schlüssel zur Entwicklung von Applikationen für die TTSoC Architektur. Eine von der Architektur bereitgestellte dynamische Ressourcenverwaltung impliziert jedoch, dass Komponentenschnittstellen und Interaktionsbeziehungen nicht notwendigerweise statisch sind. Zusätzlich ist sowohl der verfügbare Design-Raum als auch das Verlangen nach immer höherer Integration groß: Heutige Chips enthalten hunderte Cores und alle sollen aufgrund von Skalierungsvorteilen auch genutzt werden.

Unser Ziel ist die Formulierung eines generischen Prozesses zur Entwicklung von verteilten Applikationen für die TTSoC Architektur. Dazu haben wir existierende Methoden sowohl für die Time-Triggered Architecture (TTA) als auch generell im Feld verteilter Systeme begutachtet. Basierend auf diesen Erkenntnissen haben wir einen Entwicklungsprozess für die TTSoC Architektur gestaltet und seine Brauchbarkeit anhand eines Fallbeispiels evaluiert.

Im Entwicklungsprozess haben wir mehrere *service-orientierte* Modelle definiert, die das System auf unterschiedlichen Abstraktionsebnen darstellen. Modelltransformationen verfeinern das System bis es auf einer Plattform (e.g., einem TTSoC Chip) ausgeführt werden kann. Das zu diesem Zweck entwickelte Tool unterstützt die Modelltransformationen und erzeugt automatisch eine Konfiguration für Elemente der TTSoC Architektur (e.g., TTNoC, Gateways, ...).

Im Zuge der Gestaltung des Entwicklungsprozesses wurden existierende Arbeiten zur TTSoC Architektur sowohl auf Konzept- als auch Implementierungsebene erweitert. Wir haben das Konzept rekursiver Komponenten auf die TTSoC Architektur angewendet, sodass Komponenten andere Komponenten enthalten können und als Gateways Verbindungen zwischen Komponenten herstellen können.

Ein Fallbeispiel validiert die entwickelten Erweiterungen der Architektur und zeigt die Brauchbarkeit des Entwicklungsprozesses. Das Konzept der rekursiven Komponenten erlaubt eine Systemgestaltung die bezüglich Komplexität, Zuverlässigkeit, Performance und Energie skaliert. Mit dieser Arbeit hoffen wir die TTSoC Architektur für weitere wissenschaftliche Evaluierung, ihren Einsatz in Lehrveranstaltungen und ihre Verwendung in der Industrie gefördert zu haben.

Contents

CHAPTER 1

Introduction

1.1 Motivation

These days we rapidly approach limits when we try to build more cost-effective, smaller, more powerful, and more energy-efficient systems. Advances in the semiconductor industry still follow Moore's law which already proved valid for the past half a century. It states that complexity (number of transistors in a single design) roughly doubles every two years. Right now (in 2013) there are designs of up to 4 billion transistors with a feature size down to 22 nm. One fundamental limit is a feature size in the order of 0.1 nm - the size of atoms. It is believed that we cannot go below the atomic scale. Hence, Moore's law must become invalid at some point in the future, but it is currently not the real problem: The ever increasing complexity becomes more and more unmanageable and leads to a rising productivity gap, also called *"design crisis"*. On the one hand, high density integration reduces costs per transistor/memory bit, but on the other hand traditional methodologies are just not able to explore the vastly increased design space fast enough to deliver an optimal solution w.r.t. time-to-market and cost. For example, in the avionic domain most Commercial Off the Shelf (COTS) available multi-core Central Processing Units (CPUs) are classified as "*Highly Complex COTS microcontrollers*" [3] which sets the certification efforts so high that it is more cost effective to operate multi-core CPUs as single-core CPUs by turning off all but one core.

Traditional designs often neglect time as a critical parameter: e.g., it is only important that a memory load operation eventually loads data from memory, but not when exactly. This makes worst-case analyses very difficult, especially if the system is optimized for the average case and uses techniques like interrupts, caches, instruction reordering or arbitrates access to shared resources (e.g., memory, buses, ...) on-demand. Further, for most processing cores there simply are no faithful timing models available which limits timing analysis to essentially evaluating a finite number of test cases. This leaves room for error. While tight worst-case timing analysis is already intractable for those average case optimized single-core architectures it is even worse in the context of multi-core systems [40] [2].

A time-aware abstraction during system design is key to manage the available design space and hold the cost/time-to-market ratio at competitive levels. Especially in the field of distributed (hard) real-time systems it is further critical to have architectures that provide analyzable timing.

The TTSoC Architecture [32] extends industrially well accepted concepts of the TTA [28] to the chip-level. The TTSoC Architecture organizes a *system* in *components* and delivers several *core services* (i.e., global timebase, time-triggered communication, resource management, diagnosis) to those components. Components (e.g., Intellectual Property (IP) cores) adhere to a LIF specification that defines their *message*-based communication with other components. Hence they can be composed such that they can interact side-effect free with each other.

Critical to this goal is a controlled separation of components and their access to shared resources (e.g., power, communication) in the spatial and the temporal domain. A Time-Triggered communication service, ruled by an a-priori communication schedule, establishes message passing between components on-chip (e.g., by a TTNoC) and off-chip (e.g., through a Time-Triggered Ethernet gateway).

In the past six years the TTSoC Architecture has been developed, implemented and successfully employed by the Institute of Computer Engineering, TU Vienna in several international projects with industrial and academic partners. Figure 1.1 shows these research projects and

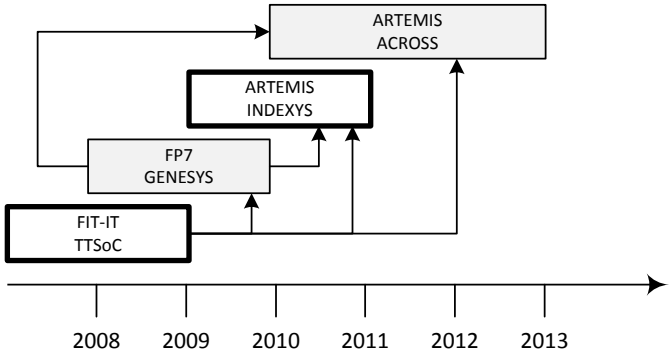


Figure 1.1: Research Projects

their dependencies on the timeline. The projects and their objectives are:

- **TT-SoC:** Time-Triggered System-on-Chip, FIT-IT, grant number: 813299/7852
  "to lay the foundation for a next-generation embedded system architecture that provides a predictable integrated execution environment for the component-based design of many different types of embedded applications"[1]

- **GENESYS:** Generic Embedded System Platform [34], FP7, grant number:FP7-213322
  "to develop a cross-domain reference architecture for embedded systems [...]"[2]

- **INDEXYS**: INDustrial EXploitation of the genesYS cross-domain architecture [24], ARTEMIS Joint Undertaking, ARTEMIS-2008-1-100021

[1] https://ti.tuwien.ac.at/cps/research/projects/ttsoc
[2] http://www.genesys-platform.eu/objectives.htm

"Cross-domain instantiation of GENESYS embedded system architecture for new platforms in Railway, Aerospace and Automotive domains."[3]

- **ACROSS**: ARTEMIS CROSS-Domain Architecture [39], ARTEMIS-JU, FFG, Funding ID: ARTEMIS-2009-1-100208
  "to develop and implement an ARTEMIS cross-domain reference architecture for embedded systems"[4]

This master's thesis was in part supported by the TT-SoC and the INDEXYS projects.

1.2 Contributions

In this thesis, we explore the following research questions:

- How would we design and implement distributed applications for the TTSoC Architecture?

- What application properties must be specified and what properties can be automatically derived at which level of abstraction?

In earlier work the TTSoC Architecture has been conceptualized, its services and interfaces have been specified, and finally the architecture has been implemented for FPGA based platforms. This thesis continues the line of research and defines an application development approach for the TTSoC Architecture.

To this end, we apply a recursive component concept to the TTSoC Architecture and use the gained expressiveness to describe virtual components on the one hand and components that are contained in another component on the other hand.

In the thesis, we present how the concept of components that are comprised of other components aids to automatically instantiate gateways between sets of components. Concerning virtual components that are intended to run on general purpose CPU based host components we illustrate an execution environment, called Virtual Component Runtime Environment (VCRE). The thesis describes the design and implementation of the VCRE as well as a reference implementation of a small real-time operating system that can provide the VCRE.

One central aspect of distributed applications in the TTSoC Architecture is their dynamic adaptability to changing demands. However, there exists no implementation of dynamic resource management for the TTNoC at hand. This thesis presents an algorithm to schedule communication channels for a TTNoC with arbitrary topology.

[3] http://www.indexys.eu/html/objectives.htm
[4] http://www.across-project.eu/

The major contribution of this thesis concerns our model-based application development approach for the TTSoC Architecture. The thesis describes the design flow, the models that represent the system under development at different abstraction levels, and the partly tool-assisted model transformations. Further, the thesis covers exemplary meta-models and describes a tool that derives a platform configuration (i.e., low level configuration data for the TTSoC Architecture implementation) from the model transformations.

Finally, we present a case study where we use the developed architectural elements and show the viability of the proposed application development approach.

1.3 Structure of the Thesis

In Chapter 2 we introduce important concepts and provide background information which we require during our thesis. We present the TTSoC Architecture in Chapter 3. Chapter 4 concerns advancements of the component concept used in the TTSoC Architecture. We also describe our proposed design and implementation advancements related to the current FPGA-based TTSoC Architecture implementation. Chapter 5 represents the core of this thesis. We discuss our model-based development approach which is inspired by concepts from the Service oriented Architecture (SoA). In Chapter 6 we apply the presented development approach in a case study that primarly gives an idea of the feasibility of our approach, but also demonstrates the TTSoC Architecture's capabilities concerning mixed criticality. We conclude our master's thesis in Chapter 7 by providing answers to our research questions and summarizing our contributions.

CHAPTER 2

Concepts and Background

The TTSoC Architecture is based on the Time-Triggered Architecture and as such contains many important concepts of distributed (real-time) systems. Also highly relevant to this thesis are concepts from Model-Driven Development (MDD). This chapter introduces these concepts and provides background information about the Time-Triggered System-on-Chip (TTSoC) Architecture.

2.1 Distributed Systems

A *distributed system* consists of spatially separated, essentially autonomous subsystems or components that interact with each other over a network. To build applications that adhere to possibly high dependability or quality of service requirements those distributed systems must be designed carefully. This section discusses important concepts related to distributed systems and their design.

Frameworks and Architectures

In our work we use the same general definition of a *framework* as Lee: *[..] a framework is a set of constraints on components and their interaction and a set of benefits that derive from those constraints* [30]. For example an operating system partitions hardware resources (e.g., CPU, memory, network devices, ..) to allow its components (i.e., processes and threads) to interact with each other and the outside environment. Benefits of this hardware partitioning are for example concurrency, safe interprocess communication, shared access to a TCP/IP stack or per-process protected (virtual) main memory. Another example of a framework outside of the context of computer science is a society. There, the components are people whose interaction is restricted by moral norms. Depending on those moral norms very different benefits arise: e.g., social intercourse, cooperativeness towards personal or group goals that may even lead to improved survival chances in the event of disasters.

An *architecture* on the other hand further constrains the components in a framework to a certain structure. In this context, a framework may or may not support a specific architecture. For example the TTA builds on the framework of time-aware distributed computing, where components are computing nodes that have access to a global time base and interact with each other over a network. The TTA constrains this interaction of its components to a time- and value precise interface specification (i.e., the *linking interface*) enabling the construction of large highly predictable and hence dependable applications.

In the framework of a society, a *nation* is an example for an architecture. Laws of the nation – usually in accordance to moral norms of the given society – introduce architectural components, structural classifiers for the components and regulate the interaction of such classified components. Structural classifiers are for example individuals, organizations and the government. Examples for architectural components of a state are the executive branch (e.g., police, military) or the judicature (e.g., courts). Some additional benefits of such an architecture are for example: freedom of speech, access to education or affordable public transportation.

Frameworks and architectures help in constructing large systems. Because of the introduced constraints and structure they often significantly contribute to a reduction of system complexity (i.e., the mental effort required for humans to understand and build complex systems).

We give one further example of an architecture that is supported by many frameworks: the SoA. We heavily used concepts originating from there to introduce our development approach (see Chapter 5) for the TTSoC Architecture. In a sense the SoA adds even more structure to the original TTSoC Architecture and allows for further important simplifications in the development methodology.

The Service oriented Architecture (SoA)

In the OASIS Reference Model [46], the SoA [23] is defined as: *"[..] a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains."* and services as *"The means by which the needs of a consumer are brought together with the capabilities of a provider."*. In a technical sense, *distributed capabilities* are for example computing nodes, sensors or actuators, but SoA also allows for non-technical capabilities like business departments or single persons. Further, the SoA acknowledges by *"under control of different ownership domains"* that those distributed capabilities are not necessarily homogeneous: large distributed systems are built by lots of diverse technology (e.g., originating from legacy subsystems) and it is not sufficient to target a uniform (technological) landscape.

Central to the SoA is the service: distributed capabilities (acting as a service consumer) might be in need of other distributed capabilities (acting as service providers) to be able to offer their capability as a service. For example, a controller for a heater element needs the services temperature and heater/cooler to provide feedback regulated temperature control to its environment. Hence, high interoperability and loose coupling of interacting entities are properties implied by the SoA. Conceptually, those two properties are realized by the Enterprise Service Bus (ESB). The ESB is not necessarily a physical network topology, but only reflects the fact that in principle any need can be satisfied with a matching capability.

An important representative of the SoA are *Web Services* where devices are available over the Web (Internet) and interact with each other using HTTP and XML.

Real-Time Systems

Processes in the physical world change entities over dense real-time, i.e., system evolution is tightly coupled with the progression of real-time. For example a car moves at a certain velocity which is a composed physical quantity: meters per second. Hence, in this Newtonian view[1], time is considered as a dimension (of space-time) and one of its many definitions characterizes it as *"the continuous passage of existence in which events pass from a state of potentiality in the future, through the present, to a state of finality in the past"*[2]. Further, time not only orders events in past, present and future, but time assigns also instants to events such that they can be partially ordered (there is no order relation between events that occur at exactly the same time instant). The time span between any two instants is called duration. Digital systems cannot represent and subsequently use 'raw' dense real-time, hence from now on we consider real-time as a discretized 'shadow' of dense real-time.

In real-time systems [25] overall correctness depends on value-correct outputs at the correct time instants. If the value-correct outputs have some utility after their deadline has passed (e.g., mouse movement of a personal desktop computer system under load), such a real-time system still operates correctly and we call it a soft real-time system where best effort response times and degraded peak-performance suffice. In case missing deadlines could have catastrophic effects (e.g., damage of property, endangering human life, ...), the real-time system is called hard. In hard real-time systems guaranteed response times and predictable peak-performance are essential correctness requirements. Concerning the control of pace (e.g., task activation, sensor sampling instants, ...) a hard real-time system usually employs the time-triggered paradigm where any system action is directly determined by the progression of time. Such systems allow guaranteed worst-case performance, while system resources may remain unused. In contrast, high throughput systems usually follow an event-triggered paradigm where events determine system load. Hence such systems allow for a high average-case performance.

Global Time

In a distributed real-time system it is important that events observed and generated at different nodes can be temporally ordered. To this end, the TTA introduces a discrete *global time base* [25] that can be established for example by synchronizing local clocks of the constituent nodes. In the reality, perfect clock synchronization is unattainable, hence such a global time base is required to be sparse, i.e., after every interval of activity there is an interval of silence which must be larger than the precision of the clock synchronization. Even within a chip a perfect clock synchronization remains unachievable. For example, consider a digital design of a chip where a single global clock signal drives for all constituent IP cores a local counter that represents the

[1] We only consider the Newtonian view because most distributed real-time systems are small enough that relativistic effects have no influence on their correct behavior. Naturally, this simplification does not apply to systems that are spatially very far distributed, like GPS or other systems that operate in space. Quantum mechanics offer another interesting view of time. There, one theory claims that time emerges from entanglement, which was apparently confirmed in a very recently conducted experiment [31]. This result may finally close the gap between quantum mechanics and general relativity (also known as 'the problem of time'), but is of course of no immediate relevance to this thesis.

[2] http://www.collinsdictionary.com/dictionary/english/time, Harper Collins 2013

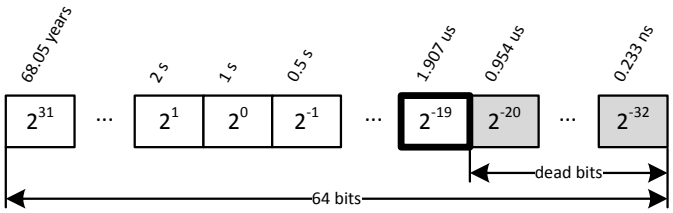


Figure 2.1: The TTSoC Architecture Digital Time Format

global time. There, the interval of activity is centered around the rising edge of the global clock signal and the interval of silence starts after the digital logic has settled and lasts until the next rising edge occurs.

Digital Time Format

The time-triggered paradigm (i.e., every action is initiated by the progression of time and not by any other event) is essential for hard real-time systems. Thus agreeing on a digital representation of time is crucial. This digital representation must characterize all important parts of the physical time in the digital world (i.e., cyberspace in terms of Cyberphysical systems). Such a digital representation of time is best described as a bitvector clock that counts upwards on every tick event (e.g., osciallation of a crystal). The granularity of this bitvector clock is the duration between two adjacent tick events. The horizon of such a digital clock is the number of different time stamps the bitvector can represent. Hence the horizon is determined by the size n of the bitvector and is exactly 2^n. Lastly, the epoch (i.e., the starting point of the clock) can be set as an arbitrary offset to any time scale like UTC or a chronoscopic (no discontinuities) time scale.

The TTSoC Architecture relies on a time format that is very similar to the format used in the Network Time Protocol (NTP), a network protocol for clock synchronization widely employed on the Internet. The time format is 64-bit wide, the upper 32 bits count full seconds and the lower 32 bits split seconds. In contrast to NTP, the TTSoC Architecture time format follows a chronoscopic time scale (similar to Time-Triggered Ethernet (TTE)) and the epoch is reset to zero when the system starts. Theoretically, this time format supports a granularity of approx. 233 ps, but for most applications a respective tick event rate of approximately 4.3 GHz is not desired (energy considerations, costs, ...). Hence, the digital time format allows the specification of *dead bits* which are stuck at zero. The actual granularity is determined by the first non dead bit. Figure 2.1 shows the 64-bit wide bitvector with an actual granularity of approx. 2 us (2^{-19}) and 13 dead bits. In the TTSoC Architecture, tick events are generated by the *macrotick* which is derived from a clock signal of appropriate frequency. The bit that is driven by the macrotick is also referred to as the *macrotick bit*.

Composability

Composability is an architectural feature which allows the construction of large systems from smaller prevalidated sub-systems (i.e., components) without the occurrence of unintended side effects under well known limits. In our work we use Kopetz's definition of a component:*"A com-*

ponent is a self-contained hardware/software unit that interacts with its environment solely by the exchange of messages. A component is a unit of design and a unit of fault-containment" [25]. The benefits of good composability is that validation efforts of the composed larger system remain minimal. For example, Ethernet enabled components like personal computers, printers or storage devices can be easily integrated in an Ethernet segment to a work group setup, where multiple users can share data and print their documents. However, composability is limited in this example by the available Ethernet bandwidth and the number of devices. According to Kopetz, composability is based on 4 principles [25]:

- **Independent Development of Components:** This composability principle is established by the precise definition of component LIFs. Only then, components can be developed independently which also facilitates *component reuse*. The LIF is a contract a component must fulfill entirely before it can be integrated in the overall system. It consists of an *operational* and a *meta-level* specification. The operational specification concerns interworking issues, i.e., the transport of data (for example: electrical signalling, message structure). The meta-level specification deals with interoperability, i.e., the exchange of information.

- **Stability of Prior Services:** Component services that have been validated in isolation are preserved during system integration. This means that any previously validated component that delivers services through its LIF will continue to delivery these services in any possible integration configuration with other components. A negative example is a Controller Area Network (CAN) bus based system: Even though all CAN nodes may operate within their specification in isolation, there might be catastrophic failures if the nodes are integrated on a CAN bus. CAN is a Carrier Sense Multiple Access/Collision Avoidance (CSMA/CA) protocol. A component's ability to dispatch messages heavily depends on the state of the bus. This state is determined by all CAN bus participants. Hence every change of the number of bus participants or even a change of the behavior of a single bus participant requires a complete revalidation of the overall system.

- **Non-Interfering Interactions:** Any two groups of components where all components only interact within their respective group are independent. For example, a system consisting of four components whereas two disjoint sets of two components form two groups. All components share a single power supply and a single communication infrastructure. Nevertheless, the underlying architecture ensures that one group of components cannot interfere with the interactions of the other group - e.g., by enforcing allocated power and network resources.

- **Preservation of the Component Abstraction in the Case of Failures:** A composable architecture establishes that component internals have only influence on interactions with other components strictly in the scope of their component's LIF, even in the case of failures. Thus, a faulty component must be detectable only by observing the behavior at its LIF and it must be replaceable by any other component that adheres to the correct behavior of the faulty component's LIF.

Recursive Component Concept in the Time-Triggered Architecture

Based on Simon's view on *nearly decomposable systems* and non-formal hierarchies [43], Prof. Kopetz derived a recursive component concept [25] for the TTA. A set of components can be organized as a *cluster*. The cluster can communicate with other clusters by means of Gateway (GW) components. On a higher hierarchy or abstraction level, the inner details of a cluster are hidden and the whole cluster can be considered as a component again. Figure 2.2 illustrates the concept: On each recursion level a set of components includes a GW component that enables message exchange between the current recursion level and the next.

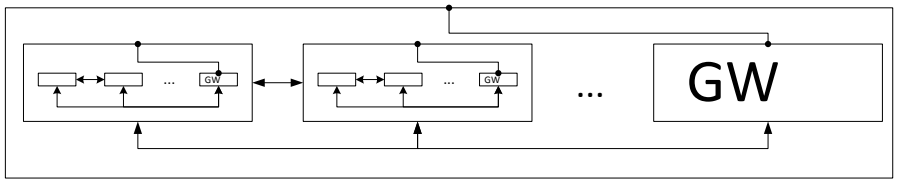


Figure 2.2: The Time-Triggered Architecture Recursive Component Model

Pulsed Data Stream (PDS)

A PDS [26] is a communication primitive intended for distributed real-time systems with the global clock feature. PDSes support the transport of highly jitter sensitive messages for continuous control applications as well as bulk-data transfers required for multimedia applications. Continuous control applications consist of a continuous process that is controlled by a digital system. The digital system periodically samples sensor values, calculates the new set points and apply them through actuators. This periodicity leads to the powerful abstraction of *a cycle* that is repeated infinitely often. Cycles are periodic, hence they can be assigned to a *period* (e.g., 1 second, $\frac{1}{2}$ second, ...). Every action (e.g., message transports, task triggers/deadlines, ...) is aligned relative to the start of the cycle. This alignment is called *the phase* or *instant* within a period. Another fundamental parameter is *the duration* of an action which determines latency constraints of the message transport system and deadlines at the communication channel endpoints. Finally, the *size* of a PDS determines the maximum size of the message a PDS can transport. With respect to a communication line, PDSes can be unrolled and mapped to a set of legacy Time-Division-Multiple-Access (TDMA) slots. Figure 2.3 shows the PDSes of a simple system consisting of two continuous control loops that shall run synchronized. We assume for this example that each control loop consists of 3 nodes: A sensor, an actuator and a control node. Hence it is necessary to communicate sensor values to the control node and set values from there to the actuator node resulting in two PDSes per control loop (in total there are 4). In this example, all PDSes have the same period, duration and size. Only the phase is different, because transport of sensor values shall take place in phase 1 and transport of the set values to the actuator nodes in phase 4. This situation demonstrates the potent feature of *interleaved pulses*: at the level of TDMA slots to which all cycles must eventually be mapped, slots can be used in any order as long as send instants are within the communication phase.

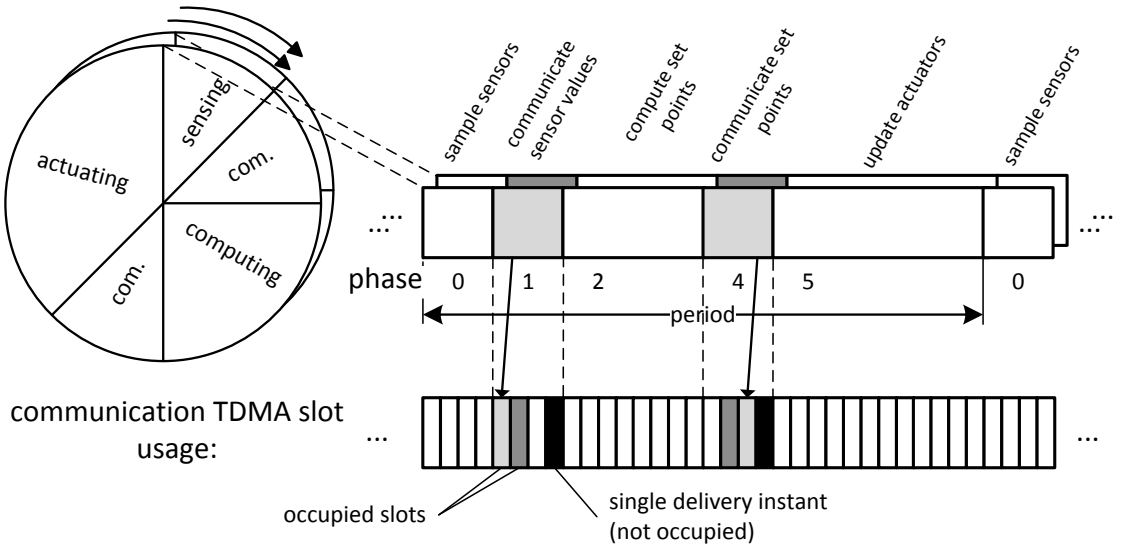


Figure 2.3: Two Pulsed Data Streams mapped onto TDMA Slots

Every PDS is characterized by the parameters period π, phase ϕ, duration d and size s. For the sake of complexity reduction in implementations and schedulers, all parameters are restricted:

- The period is a negative power of 2 and the number of negative powers of 2 is limited: e.g., $2^{-1}\text{s},\ 2^{-2}\text{s} \ldots 2^{-14}\text{s}$. Hence, in the implementation, the period can be simply associated with an index in the bitvector clock representation of the digital time format. We refer to this index as the *period bit*. Further, we can choose to skip certain powers of two by defining a minimum distance between any two period bits. We call this minimal distance *period delta*. For example, if the period delta is one, then no powers of two are skipped. In case of two, every second power is skipped.

- There are a fixed power of 2 phases per period. Again the phase can be expressed by the n-bits in the bitvector clock representation of the digital time format. Hence, we also refer to this parameter as *phase slice*.

- The maximum duration of any PDS is limited. The duration determines the amount of TDMA slots used, i.e., the amount of data the PDS transports.

We use the following notation[3]: π_n refers to the nth period which has a cycle of 2^{-n} seconds, and ϕ_k^n to the kth phase of period π_n. The actual offset of the pulse relative to the start of the period π_n represented by ϕ_k^n is determined by:

$$\phi_k^n = k\frac{\text{length}(\pi_n)}{\text{max\_phases}(\pi_n)}.$$

[3]In this thesis we sometimes represent the type of a variable explicitly. For example here, π denotes the type which is 'period' and n is the actual variable.

Consequently, in a high-level description a PDS with a unique identifier id is represented as a tupel:

$$\mathcal{P}_{\text{id}} = \{\pi_n, \phi_k^n, d, s\}.$$

In the low-level description, duration d and size s must be resolved to actual TDMA slots. Note that the transport system introduces delays. Thus, we also need to differentiate between phases at the origin (sender) and the destination(s) (receiver(s)).

2.2 Model-Driven Development (MDD)

In MDD [8][4], models not only describe aspects of a system (e.g., in specification documents), but actually rule the whole development process. For example, starting from models, tools can automatically derive implementation source code or documentation for which they may not even require any (human) engineering assistance at all.

Models and Metamodeling

A model is an abstraction of a system [44]. This abstraction has the following properties:

- **Mapping:** The model is based on the original system it represents.
- **Reduction:** The model only represents a subset of the original system's properties. This subset is determined by the purpose of the model.
- **Substitution:** the model can be used in the context of the model's purpose instead of the original system.

In the framework of formal languages, models are a language. For example, a specific Unified Modeling Language (UML) activity diagram that represents a system is a sentence conforming to the concrete syntax of an UML activity diagram. This concrete syntax is the notation/visualization (graphical diagram elements) of an abstract syntax. The abstract syntax or the metamodel defines language concepts (Activity, ActivityTransition, ...) and the language grammar (there must be an initial state, every transition must involve exactly two Activities, ...). Finally, models can be instantiated and such a model instance describes a snapshot of a system represented by a model.

Model-Driven Architecture (MDA)

The MDA is the Object Management Group (OMG)'s effort to standardize MDD. These standardization efforts are condensed in modeling languages like Meta-Object Facility (MOF), UML or XML Metadata Interchange (XMI). The original intent of the MDA is software design, but there is a large body of evidence that the MDA doesn't need to be limited to software only: [10] [5] [20]. Further, state of the art believe is that an unorganized separation of hard-

[4]This section is influenced by the excellent lecture about Model Engineering, WS2012, Prof. Kappel and the cited lecture literature.

and software during system design lead to a loss of predictability (because of widely employed throughput optimizations such as the use of caches, instruction reordering, ...) in most COTS devices and their programming models. That loss of predictability greatly contributes to artificial restrictions and unnecessary overhead in the context of real-time (embedded) system design [19] [29]. While a time-independent abstraction (i.e., computation does not depend on physical time constraints) allowed for great success in non-real-time computer systems (e.g., web applications, desktop operating systems, big data mining systems, ...), the design and verification of real-time systems has become extraordinary complex. Hence, it is of paramount importance to include the (physical) execution machine in design considerations of real-time systems. For example, a C program together with an input/output library can be considered as a model of a system. Unfortunately, there is no notion of time in the C standard and the execution time of any C program depends heavily on the executing CPU based system (e.g., used CPU architecture, system frequency, memory architecture, ...). Consequently, C programs certified to run correctly on one CPU-based system may show incorrect timing behavior on another system that uses for example a different CPU architecture.

Nevertheless, the MDA promotes interoperability across different technology platforms. However, this need not be problematic for real-time systems in case the technology platform remains time-analyzable and predictable. The interoperability across different technology platforms is expressed by a separation between the abstraction level that is independent of the platform and the abstraction level that is specific to the platform. The Platform Independent Model (PIM) is a system functionality description at the highest abstraction level and all other system descriptions at any lower abstraction level must adhere to its definitions of the system behavior. Model-to-model transformations together with Platform Models (PMs) allow the (automatic or human-guided) generation of Platform Specific Models (PSMs). PSMs can be directly deployed to platforms. For instance, CORBA, .NET or Java are platforms in the domain of software engineering. In our context, a hardware device containing TTSoC Architecture compliant chips is a platform.

Model Transformation

Model transformation is a formal method to arrive from one or more source model(s) at one or more destination model(s) that is/are consistent to the source model(s). For unidirectional transformations the definition of consistency is simple: A set of destination models is consistent to a set of its source models if it is produced according to the transformation. Bidirectional transformations need to be bijective where consistency is established if the source and destination set of models express exactly the same information. Model transformations are also classified according to their language closure: in case the set of source models and the set of destination models have the same meta-model, the transformation is said to be endogenous. Otherwise the transformation is exogenous. Popular means to conduct model transformations are graph transformations[5] and model transformation specific languages: e.g., Atlas Transformation Language (ATL), OMG's standardization efforts called Query View Transformation (QVT). Purposes of model transformation are mainly: model evolution (usually unidirectional, exo- or

[5] models are based on graphs

endogenous), automatic source code or documentation generation (usually unidirectional, exogenous) and realising interoperability by appliance of an *architectural style* (bidirectional, exo- or endogenous).

Executable Model

Similar to the C program example above, models without exact execution semantics remain an incomplete system behavior description. Accordingly, models need to include an execution semantics that describes how the system snapshots or model instances evolve over time. Important representatives for executable models are Executable UML (xUML), the OMG's standardization effort of an executable UML: Foundational Subset for Executable UML (fUML) [36], Ptolomey II [9], Mathwork's Matlab Simulink[6] and Esterel Technologie's Safety-Critical Application Development Environment (SCADE) [1]. Certainly, the listed executable models imply fixed models of computation and various manual and automatic transformations exist to obtain a PSM that may or may not require additional analysis to ensure consistency to the source model. For example, SCADE contains a qualifiable C code generator (KCG) that does not require unit-tests of generated C code, but still needs to be statically analyzed concerning its Worst Case Execution Time (WCET) and static memory requirements. Although, this execution time and stack usage analysis is simple because only a subset of the C language is used and library usage is restricted (e.g., no dynamic memory usage, dynamic concurrency, pointer arithmetic, recursions or unbounded loops).

[6] http://www.mathworks.com/products/simulink/

CHAPTER 3

The Time-Triggered System-on-Chip (TTSoC) Architecture

In this chapter we give a brief description of the TTSoC Architecture [32]. Main goals of the TTSoC Architecture are:

G0 **Move to an Integrated Architecture:** In federated architectures every function is realized in its own independent system. The TTSoC Architecture is an integrated architecture [33] that aims to share system commonalities among its subsystems. The TTSoC Architecture shall support the execution of several different (with respect to functional but also non-functional properties: e.g., criticality) distributed applications on a single chip.

G1 **Provide a Time-Triggered Execution Platform for Distributed Applications:** The TTSoC Architecture must provide means to execute applications that rely on the time-triggered execution paradigm.

G2 **Manage Chip Complexity:** Industry is able to provide very large chips which offer a vast design space for hundreds of IP cores where conventional bus based interconnects with limited control on available bandwidth and quality of service properties do not scale well. Complexity with respect to closing large designs under possibly critical non-functional constraints like energy budget, security and safety must be addressed by a scalable architecture.

G3 **Correctness by construction:** The architecture must support side-effect free composition of prevalidated and possibly heterogeneous components such that the resulting system still meets all its correctness requirements (i.e., does not deviate from specified behavior).

G4 **Dependability and Quality of Service Mechanisms:** The non-functional properties dependability and quality of service shall be provided right by the architecture. Applications shall focus on system functional properties only.

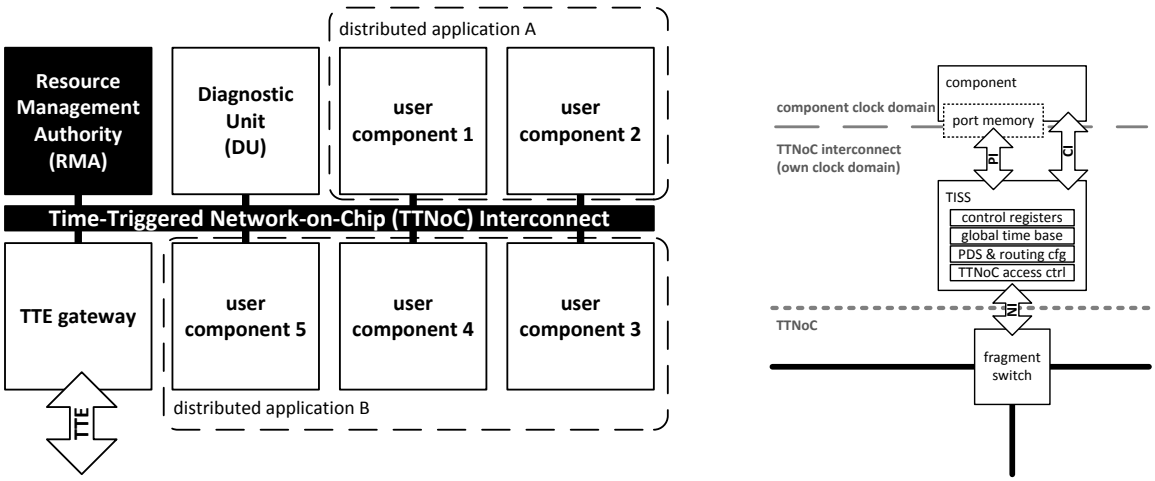
G5 **Monitoring without Probe Effects:** Probe effects would destroy composability features, hence the architecture must guarantee monitoring of components without modifying their interaction behavior. Monitoring allows for side-effect free system runtime verification.

G6 **Dynamic Resource Management:** The architecture shall be able to adapt its services according to changing needs. For example, depending on available non-critical network bandwidth it makes sense to design a movie playback system such that there are multiple levels of *service degradation* (e.g., reduced resolution or frame rate). Depending on other concurrently executing distributed applications an appropriate service degradation level can be selected.

The TTSoC Architecture tackles all these goals with a set of *architectural core services* and *architectural optional services* that are provided by architectural components.

3.1 Overview

Figure 3.1a gives an overview about the elements of the TTSoC Architecture. Components[1] are realized as *IP-cores* which are also the *primitive elements* (i.e., no further component recursion beyond IP-cores). A cluster of such components resides with the shared communication



(a) System Component Model

(b) TTNoC Interconnect

Figure 3.1: The TTSoC Architecture

resource, called the *TTNoC Interconnect* on a physical chip. Naturally, a component could also act as a gateway and implement an external LIF that abstracts the other components that are part

[1] In original TTSoC Architecture terminology the component was called 'micro component' or μcomponent.

of such a cluster. On the higher abstraction level, only the external LIF is visible and arbitrary complex recursive system hierarchies can be built: e.g., In Figure 3.1a the TTE [27] gateway hides all other internal components from the components connected to the TTE which perceive the gateway's external LIF as the LIF of the component *TTSoC Chip*.

The Resource Management Authority (RMA) is an architectural element realized as a component with special access rights. It is the only component allowed to carry out *reconfigurations* of component-shared resources like the TTNoC Interconnect or power. Hence, the TTNoC Interconnect and the RMA compose the *trusted subsystem*. Nevertheless, the RMA is an optional component, because for systems that don't require reconfiguration during runtime, the TTNoC Interconnect and other shared resources can be statically configured during design time.

All other components are *untrusted* in the sense that they need not to be trusted concerning correctness. They cannot carry out reconfigurations or use shared resources outside their allocated budgets. Consequently, the trusted subsystem should be certified up to at least the same criticality level of the highest critical distributed application running on the TTSoC chip.

The Diagnostic Unit (DU), like the RMA also realized as a special component, is responsible for observing on-chip component interaction and uses this information to deduce a chip health-state. The schedule of the TTNoC Interconnect can be reconfigured such that it delivers general error information and a copy of selected application messages (originating at user components) to the DU. If the TTNoC Interconnect reconfiguration is schedulable, the monitoring services of the DU do not cause any probing effects on the observed user component interactions.

The overall distributed system is structured into nearly independent subsystems that execute distributed applications[2]. In the remainder of this thesis, we treat a distributed application and the subsystem that are responsible for its execution as synonyms. A distributed application provides a service to the user (e.g., an airbag deployment system) and is further composed of closely interacting components (e.g., see the two distributed applications A and B in Figure 3.1a). The TTSoC Architecture ensures that distributed applications can operate in isolation of each other. Multiple distributed applications (with possibly different dependability requirements) can be located on the same chip or even span over gateways across multiple chips. Off-chip gateways like TTE gateways that offer similar temporal- and spatial-encapsulating properties like the TTNoC allow for off-chip component interaction.

3.2 Architectural Services

There are *core services* and *optional services* directly provided by the architecture. Core services are the smallest set of services required to provide optional services and enable user-defined distributed applications.

The two core services, established only by the trusted subsystem, are:

- **Global Time:** The TTNoC Interconnect replicates the global time base at each network endpoint (i.e., Trusted Interface Subsystem (TISS)). Both, the TISS and the component have read-only access including sophisticated time-trigger mechanisms available. For

[2] In original TTSoC Architecture terminology: Distributed Application Subsystems (DASes)

example, the component can configure periodic interrupts to schedule tasks. The TISS requires the global time to receive messages from and dispatch messages to the TTNoC.

- **Predictable Message Transport:** The TTNoC Interconnect allows for message-based Encapsulated Communication Channels (ECCs) among connected components. ECCs are logical channels on top of the physical communication resources. They are isolated in the temporal and the spatial domain, meaning in a fault-free schedule any two ECCs cannot interfere with each other. They are unidirectional and transport messages from its source endpoint to one or more destination endpoints according to the active PDS and routing configuration (i.e., the time-triggered schedule). An endpoint is called a port. ECCs need not to be contained solely on-chip, they can also pass gateways to off-chip components.

There are two more optional services that build on the core services:

- **Resource Management Service:** The RMA governs all shared resources. It can dynamically reconfigure the TTNoC Interconnect during run-time and is intended to monitor and control power distribution to on-chip components.

- **Diagnostic Service:** The DU is intended to do runtime verification of distributed applications. All on-chip ECCs can be set up such that the DU is able to monitor all component interactions. Also part of the diagnostic service is the watchdog that is implemented in every TTNoC Interconnect endpoint. The watchdog triggers the component reset in case the component fails to provide a life-sign. Finally, the DU collects and interprets generic error information disseminated by the TTNoC Interconnect.

Those optional services are independent of each other: i.e., conditional on the required features a TTSoC might contain all of them or an arbitrary selection .

3.3 Architectural Elements

This section describes integral elements of the TTSoC Architecture such that it is able to provide its core and optional services.

The TTSoC Component

In the context of the TTSoC Architecture a realization of a *TTSoC Component* (cf. Section 2.1 for the general definition) is shown in Figure 3.2. The component interfaces are partitioned in technology-dependent and technology-independent interfaces reflecting their dependence on a specific technology, e.g., physically available interfaces or component implementation internals [3]. The LIF and the configuration/planning (C/P) interface must be implemented by a component regardless of the used technology. The LIF and the local input/output (I/O) interfaces are application specific interfaces while the C/P interface is strictly a control interface. Finally,

[3] Prof. Kopetz only regards the debug interface as technology-dependent and the C/P interface as technology-independent control interfaces.

the debug interface is another control interface intended to give low-level insight into the component (e.g., a JTAG interface). This low-level insight shall primarily aid the engineer during component development.

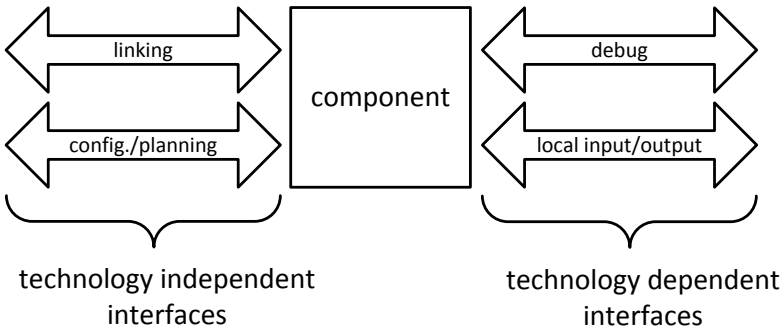


Figure 3.2: Generic Component Interfaces

All technology independent component interfaces are serviced by the *TTNoC Interconnect*. This interconnect consists of the TTNoC and network endpoints called TISSes that connect the component to the TTNoC (see Figure 3.1b).

Based on the concept of composability (see Section 2.1) and Salloum's definition of a job [13][4]we define a component as follows: **A component is a temporally and spatially partitioned self-contained hardware/software unit of a distributed application which interacts with other components singularly through its LIF.** Further, a component is also a unit of fault containment for design-faults, because the TTNoC Interconnect guarantees to deliver and forward only messages according to the time-triggered schedule. No component (excl. the RMA) can influence the time-triggered schedule, hence faults (e.g., babbling idiot) cannot propagate outside of the component.

From the implementation point of view, a component can be realized as an embedded system containing the port memory, RAM and a general purpose CPU that executes application software. However, the component must only handle the TTNoC Interconnect interfaces correctly. Hence, a highly application specific component might only consist of a port memory and combinatorial logic. Such a highly application specific component would not require any software at all.

We already introduced the local I/O interface of a component. This interface allows the component to access its environment (i.e., the outside world) through sensors, actuators or non-time-triggered systems (e.g., Ethernet, CAN, ...).

[4] In the original TTSoC Architecture terminology a *job* is considered as an atomic unit of behavior/software that can be allocated to a (μ)component. Multiple jobs may execute on a single component, but no job may be distributed to more than one component. The distinction between job and component became obsolete in our thesis, because we apply the recursive component concept of the TTA (see Section 4.1) also to the logical structure of distributed applications.

The Time-Triggered Network-on-Chip (TTNoC) Interconnect

For the sake of a more streamlined presentation, we introduce the TTNoC Interconnect (see Figure 3.1b) which consists of the TTNoC and the TISSes acting as the interconnect's endpoints. The TTNoC Interconnect is responsible for all communication relations between components. On the component side, there is the Port Interface (PI) and Control Interface (CI). The TTNoC Interconnect delivers or fetches messages as they arrive or respectively need to be transmitted (possibly in multiple fragments or chunks). The PI is designed to write/read directly into/from memory. Because the TTNoC Interconnect and the component may reside in different clock domains, we advise to dedicate a dual-clocked *port memory* to hold messages (or whole message queues). All port synchronization and other component configuration and control relevant details are handled over the CI. Also the CI must be properly synchronized to the component's clock domain. The whole TTNoC Interconnect operates in its own clock domain, separated from the component clock domains.

Time-Triggered Network-on-Chip (TTNoC)

The TTNoC is comprised of fragment switches that are interconnected according to a design-time fixed topology: e.g., a mesh. All connections are full-duplex meaning that there is a *lane* in each direction. A lane consists of a design-time configurable number of data and two control bits. The first control bit indicates *header* data and the second whether the data is *valid*. Messages are split up into packets, also called *fragments*. A fragment is a block of one or more continuously transmitted *flits*. A flit is the smallest unit of data that can be transported by the Network-on-Chip (NoC) during a single clock cycle. The fragment switches route fragments through the NoC according to the *wormhole* or *header-payload* routing method (also called source routing [48]). A flit is either header data or payload data, depending on the header control bit. Every fragment switch contains a small routing processor that processes header data and enables or disables outgoing lanes accordingly. Each time a fragment switch is passed, header data is reduced until the header flit is completely consumed. Fragment switches keep their lane configuration during the passage of non-header flits, hence once configured a channel remains open until new header data reconfigures a switch for the same input lane. Because of this per-input lane configuration and the full-duplex nature of connections between switches, 2 routes can pass the same switch bidirectionally or can even cross at a switch without collisions. Figure 3.3 gives a few samples of possible and impossible routes: Routes 0 and 1 are both valid. Both bidirectional routes (fragment switches $(1, 0)$ and $(2, 0)$), as well as crossing routes (fragment switch $(0, 0)$) can be realized, because they do not collide at any output lane of any switch. However, routes 2 and 3 do collide at fragment switch $(2, 2)$, because they both attempt to take the west exit of the fragment switch, but there is only one output lane available. Such a collision must be resolved in the time domain (e.g., by interleaving the two PDS that require the routes 2 and 3). Route 0 is also an example for a *multicast route* (one sender and multiple receivers). In bus based interconnects one sender can easily reach all participants, even at the same time instant. In a network, multicast becomes more sophisticated, because *(1)* the route must be setup in a way to reach all receivers and *(2)* the delivery instant must be the same for all receivers. The TTNoC Interconnect supports both requirements.

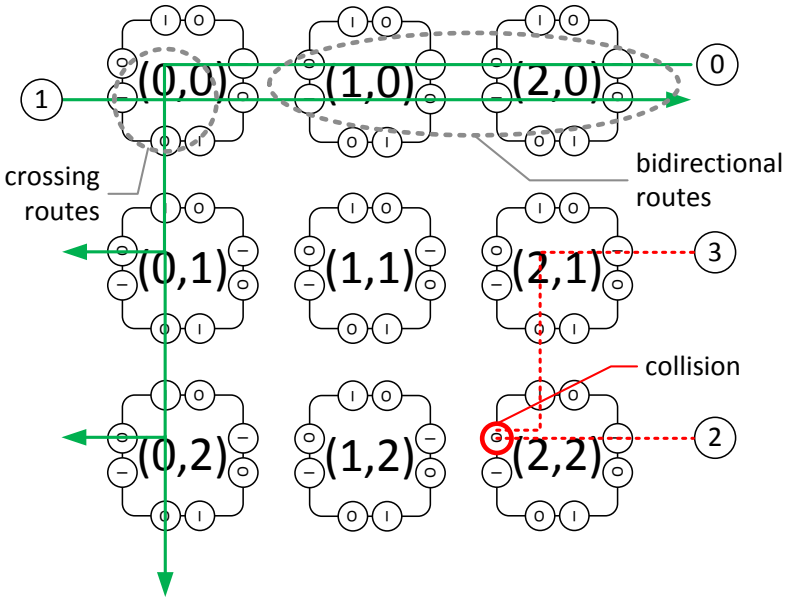


Figure 3.3: Example Routes in the TTNoC

The Trusted Interface Subsystem (TISS)

Each TISS replicates the global time in a 64-bit counter that represents the global time in the previously described digital time format (cf. Section 2.1). Figure 3.4 shows how the time format relates to PDSes. A global clock signal (i.e., the macrotick) of appropriate frequency drives each counter in each TISS. The TISS handles TTNoC access control according to a preconfigured PDS and routing configuration schedule. This time-triggered schedule controls how TTNoC suitable fragments are assembled and dispatched over the Network Interface (NI) as single flits.

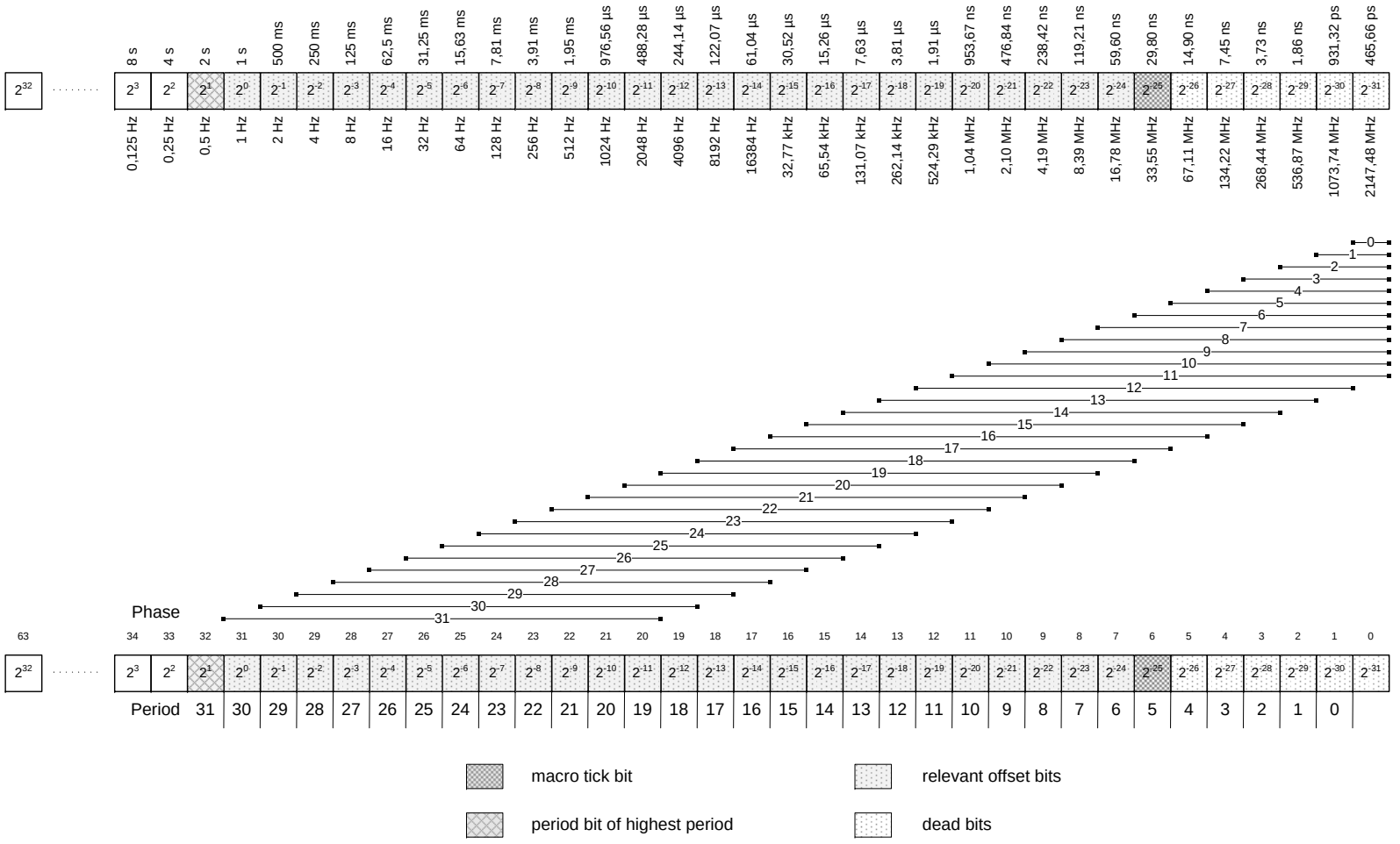


Figure 3.4: The Time-Format w.r.t. PDSes [37]

The NI connects the TISS to a fragment switch. It also consists of two lanes with the same layout and semantics as a connection between two fragment switches in the TTNoC. A component is connected to the TISS via the Uniform Network Interface (UNI) which is structured into the PI and the CI. The CI allows the component to access component relevant configuration registers (e.g., global time, message complete interrupt control, ...) within the TISS. The TISS fetches messages from the component and delivers messages to the component through the PI.

Encapsulated Communication Channels (ECCs)

An ECC is a unidirectional channel between one sender and a nonempty set of receivers. Every ECC is associated with a PDS and routing information. The TTSoC Architecture supports two types of messages: state and event messages. State messages contain the complete state (e.g., a temperature value) and event messages contain only the change from the state of the past to the state of the present (e.g., the temperature increased by 2 degrees Celsius). While the actual message transport through the TTNoC is independent of this message type, the channel endpoints or ports implement different fetch and deliver mechanisms that match the semantics of the respective message type:

- **Periodic Ports:** In original TTSoC Architecture terminology they have been called state ports. This is still an accurate description, because the term 'state' refers to the main application of periodic ports. State information is complete in the sense that it contains all information necessary to reconstruct the state itself. Such information is best sent periodically regardless of any actual updates of the state. It is sometimes not essential to receive or process all state messages, because past state messages are not required to reconstruct the current state: the currently received or processed state message is already the complete state. The TISS provides shadow-buffering to synchronize the transport of periodic messages with the component.

- **Sporadic Ports:** In contrast to periodic ports, sporadic ports (originally called event ports) are not intended to transfer state information, but event information. Such information is also used to (re)construct the state, but now the state is determined by the initial state and all observed events (i.e., changes over time) up to now. Thus it is important to transmit and receive all events in order. No event information must be lost. Hence a message is only sent if there was an event (i.e., a new message is put into the port memory). The TISS provides send and receive queues to synchronize sporadic message transport with the component.

Gateways

Gateway components with time-triggered off-chip interfaces enable deterministic off-chip component communication and are a means to meet otherwise unachievable requirements:

- **Ultrahigh Reliability:** A failure rate in the order of one failure per 10^9 hours cannot be achieved with a single component like a microchip. On a single die it is not possible to establish more than one Fault Containment Unit (FCU) [25] with reasonable independence

assumptions (usually chips have common power and clock sources). Ultrahigh-reliable systems can only be built with replicated components such that failing components can be masked (e.g., by employing Triple Modular Redundancy (TMR)).

- **High-Performance:** A single chip might not deliver the required performance. Hence, off-chip communication enables the distribution of parallelizable workloads to several TTSoC chips. Depending on the nature of the application, the off-chip interface of a gateway to achieve high-performance requirements might not need to be time-triggered. In that case, we consider the gateway as a component which has high-performance computing power attached to its local I/O interface.

The gateway's main task is to provide an *internal* and an *external LIF* and translate messages appropriately between them. The internal LIF is the 'usual' component LIF that every component needs to implement in a cluster A of components. The external LIF points outside the cluster A and might connect to another cluster B of interconnected components. From the perspective of cluster B the role of the gateways internal and external LIFs switch.

The Resource Management Authority (RMA)

The RMA is responsible for processing *resource requests* from certain components during runtime. Distributed applications may offer different modes of operations: e.g., a multimedia distributed application might dynamically use TTNoC resources depending on the number of input video channels the user is interested in to display simultaneously. In order to process resource requests, the RMA first needs to find a conflict-free communication schedule suitable for the TTNoC Interconnect. Only then the change request is carried out at a configurable *reconfiguration instant*. At the configuration instant all TTNoC Interconnect endpoints (i.e., the TISSes) switch simultaneously from the current configuration to the new one. Originally, in the TTSoC Architecture dynamic resource management was carried out in two steps: Another architectural element called the Trusted Network Authority was the sole component allowed to reconfigure the TTNoC Interconnect. Instead of the RMA, the Trusted Network Authority (TNA) formed together with the TTNoC Interconnect the trusted subsystem and the RMA remained an untrusted component. The TNA's purpose was to merely check the schedule created by the RMA before deployment. Checking a schedule is much simpler than creating one, hence also certification efforts are lower for the TNA than for the RMA. In favor of a more streamlined design, we dropped the notion of the TNA. In case certification of the resource scheduling algorithm in the RMA becomes an issue, the RMA can be redesigned to isolate the schedule checking from the schedule creation. For example, the schedule can be created on another processing core attached to the local I/O interface of the RMA.

Architectural Plugins

Architectural plugins or middleware are extensions located between components and their network interfaces. Those extensions are provided by the architecture, i.e., they can be instantiated by components and are platform independent (i.e., every platform needs to provide them). Their

purpose is to provide standardized extensions of core and optional services of the TTSoC Architecture on the basis of messages. Architectural plugins are not part of the trusted subsystem, because they are located behind the UNI. Still, their criticality requirements are at least as high as the criticality requirements of the attached components. For example, a security related architectural plugin transparently encrypts messages at the sender and decrypts messages at the receiver. Further, architectural plugins can be chained: e.g., a security plugin can first decrypt a message and then hand it over to a voter plugin. Plugins, especially if chained introduce delays or increased message sizes that need to be considered during the application design.

3.4 The Model-Based Development Process

In his dissertation [13] Salloum proposes a model-based development process founded on earlier work [20] for the TTSoC Architecture (cf. Figure 3.5). The process consists of a series of model transformations that starts at an abstract system representation and ends at a model that can be deployed to a platform (e.g., a TTSoC chip). To this end, Salloum outlines at the start of the process the Abstract Application Model (AAM) which is some TTA-unspecific, more abstract representation of the system under development: e.g., there might be no functional decomposition into components yet. The AAM and its transformation to the Fully-Specified Interface Model (FIM) can be defined and implemented by a domain-specific TTSoC Architecture compliant tool supplier. Through refinement this abstract representation is transformed to the FIM, a TTA-specific representation where components realize the system's functionality. These components are defined by their LIFs. However, messages are only fully-specified with respect to their syntactic, semantic and temporal properties. In the context of semantic and syntactic properties, 'fully-specified' means that these properties are completely defined (e.g., the message size is 4 byte and the first 2 byte are a 16bit signed integer that carries the temperature in degrees Celsius). In the context to temporal properties, 'fully-specified' is weaker: messages are only time-constrained (e.g., in terms of PDSes, the phase is constrained to a specific interval). There are two views on the FIM that represent two different abstraction levels. The Unified Fully-specified Interface Model (UFIM) is a model that defines the component LIF at the level of the TTNoC Interconnect UNI. In its meta model, the UFIM only contains the elements *UFIM channel* and *component*. A UFIM channel is an ECC between two or more components attached to the same interconnect. In contrast to the UFIM, the MFIM is a high-level, possibly domain-specific representation of the FIM. The meta-model of the MFIM contains *macros* that can be elaborated via automatic model transformation into UFIM constructs: e.g., a bidirectional channel for transactions is resolved to two separated UFIM channels going into opposite directions. Multiple MFIMs can peacefully coexist and tailoring them to a specific domain might help to decrease required efforts to build an AAM to MFIM transformation. In his dissertation Salloum describes a selection of macros. We use those as a basis and introduce a service-oriented MFIM in Chapter 5.

At the end of Salloum's development process is the model transformation from the FIM to the PAM. The PAM describes the platform (e.g., an embedded system) and a mapping between platform components and UFIM components. The transformation also takes care about attaining a complete temporal specification that is satisfiable by the physically available platform and the

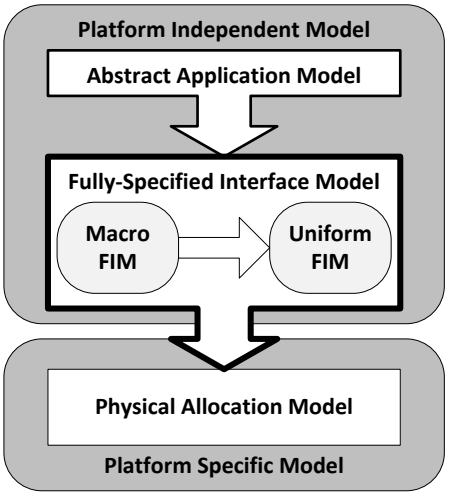


Figure 3.5: Model-Based Development Process

given constraints in the FIM: e.g., the FIM constraints a period of a component to deliver a sensor value between 500 and $650\mu s$. We assume the totality of the physical platform (i.e., processor, operating system, system frequency, ...) allow for sending said sensor value with a period $\geq 600\mu s(\pm 50\mu s)$, so in the PAM the period is fixed to $600\mu s$.

In terms of OMG's standards, the AAM and the FIM are considered both as a PIM and the PAM is a PSM.

3.5 Implementations

In 2007 Engleder [15], Seiger [41] and Weirich [49] developed in their master's theses an early prototype of the described TTSoC Architecture which served as a valuable scientific milestone. This prototype was used heavily to evaluate single theoretical considerations, although mostly in isolation. For example, PDSes have been implemented by Engleder and could be analyzed on a real physical device w.r.t. message scheduling (cf. Weirich and Huber [21]). A second version [37] of the proposed TTSoC Architecture has been implemented by Paukovits during his PhD thesis and the project TTSoC in 2008. For the first time, all the major concepts of the TTSoC Architecture have been put together in a single implementation. From an engineering point of view this implementation provides synthesizeable hardware for building a TTNoC Interconnect with arbitrary network topology and connect components (IP-cores) to it. These components need to implement the PI and CI such that they can interact with the architecture's services (e.g., global time, communication). Naturally, concrete instances of the implementation require a lot application- and component-specific configuration data (e.g., concerning schedule, routing, ports) which is cumbersome to create, modify or debug without tools that adhere to a methodical development approach. For example, configuring even the most simple communication primitive, i.e. a unidirectional ECC between two on-chip components using the TTNoC Interconnect requires to repartition the port memories of the respective components, configure

port semantics, schedule a PDS that does not disturb existing PDSes and add information about the route of the new PDS. However, because of Paukovits' elegant design and documentation efforts his implementation served as an excellent starting point for our own work. We extended the code base to accommodate for the suggested architectural changes (cf. chapter 4) and the proposed development approach (cf. chapter 5).

CHAPTER 4

Evolution of the Time-Triggered System-on-Chip (TTSoC) Architecture

In this chapter we present some changes of and additions to the TTSoC Architecture and its prototype implementation that considerably simplify application development while not violating any primary design principles of the original TTSoC Architecture. We call these changes *evolution*, because they emerged during actual application development over multiple years, i.e. hands-on working with prototypes and running distributed applications on them while searching for features/changes that improve application development (see Chapter 5). Those changes range from imposing more structural constraints, over introducing a runtime environment for software components, to proposing an algorithm to solve TTNoC Interconnect scheduling problems.

4.1 Recursive Component Concept

The original TTSoC Architecture limits its logical system structure to a two level hierarchy: At the higher abstraction level there are distributed applications (consisting of components) and at the lower level there are physical components (IP-cores) executing possibly multiple tasks (originally called jobs) in parallel. We propose to remove this artificial limitation and apply the original TTA recursive component concept (see Section 2.1). We start at the abstraction level of physical components attached to the TTNoC Interconnect and distinguish between two vertical abstractions and one horizontal abstraction:

- **Recursion Towards Lower Abstraction Levels:** All components and the TTNoC Interconnect are located on the chip, but every physical component could in principle act as a gateway and host many more *virtual components* executing in virtual machines. For example, see Figure 4.1a: a (physical) host component is an IP-core and consists of an

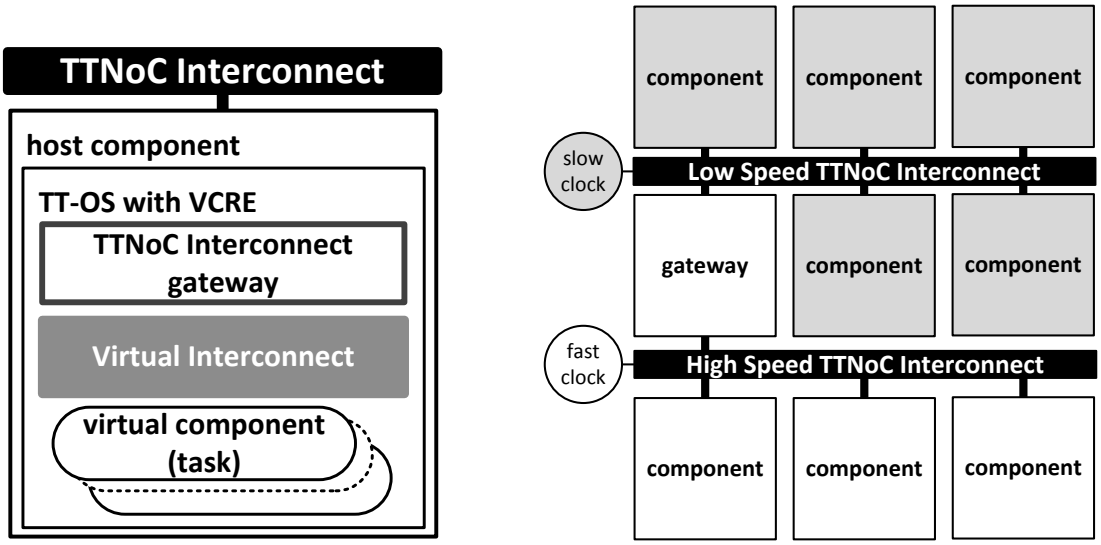
embedded processor system that runs a time-triggered operating system with a runtime environment for virtual components (VCRE). The time-triggered operating system manages the port memory and is ideally able to temporally and spatially isolate tasks and to provide those tasks controlled access to the TTNoC Interconnect services. This host component is a gateway: it has an internal LIF (as seen from the perspective of the host component attached to the TTNoC Interconnect) and an external LIF to the tasks its operating system manages. If we call these tasks 'jobs', we arrive at the original TTSoC Architecture's logical component structure. Another simplification is that we do not require further concepts (e.g., 'interjob' communication) to express communication between virtual components hosted on the same physical component. Similarly to the TTNoC Interconnect, a Virtual Interconnect establishes communication channels among virtual components.

- **Recursion Towards Higher Abstraction Levels:** A set of components can contain a gateway whose external LIF represents an interface definition of the whole set of components. This effectively abstracts the set of components that is hidden behind the gateway's external LIF to a single component.

- **Linking at the Same Abstraction Level:** A physical component could also act as a gateway and connect to other TTNoC Interconnects on the same chip. There are two immediate advantages: Firstly, the total on-chip network bandwidth can be increased by partitioning the overall chip design in multiple TTNoC Interconnects and attaching components to them. Figure 4.1b presents a situation where two TTNoC Interconnects operating at different clock speeds are interconnected by a gateway component. Naturally, an NoC operating at a higher clock requires more power than an NoC operating at a slower clock, hence linking at the same abstraction level allows for appropriately, fine-grained power and performance scaling. Secondly, horizontal abstraction enables a more robust on-chip TMR, where also the trusted subsystem and clock inputs of the chip can be replicated. Figure 4.1c illustrates such a TMR setup. Of course, beside the replicated chip pin signals and the possibility to harden against Single Event Upsets (SEUs) by physically placing the replicas at far ends of the chip die, there is still dependency with respect to failures among the replicas: e.g., common power source, common die. Hence off-chip TMR remains the only solution to reach ultra-high reliability requirements (or at least to be reasonably sure about that [7]). Finally, combined with *recursion towards higher abstraction levels*, we can build arbitrary topologies of interconnected components.

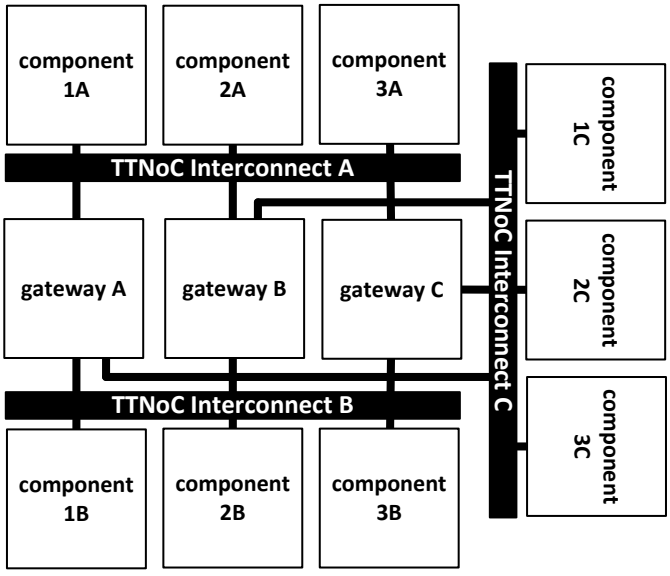
This leads to the following definition of a component *recursion property*: A component is either composite, i.e. a cluster composed of other components or primitive, i.e., it is not composed of other components. Composite components help to structure their constituent components. At the highest abstraction level only a single component remains. Naturally, this is the most powerful abstraction, because having only one component, we are not faced with a distributed system anymore.

Concerning the relationships of components (i.e., their interactions), we can apply a similar recursive concept: Two primitive components connected to the same shared communication resource (e.g., the TTNoC Interconnect) can interact with each other over a primitive encapsulated communication channel (e.g., established over a channel associated to a PDS on the



(a) Virtual Components

(b) NoC Bandwidth and Power Scaling



(c) Improved On-Chip TMR

Figure 4.1: Examples utilizing the Recursive Component Concept

TTNoC Interconnect). For any other interaction of primitive or composite components across one or more levels of abstraction the involved gateway components serve as bridges or hops, similar to Internet IP packet routers.

4.2 A Time-Triggered Operating System for Virtual Components

We already introduced the notion of a virtual component by applying the recursive component concept to the TTSoC Architecture. In this section we propose a light-weight, real-time operating system specifically tailored to provide a runtime environment for virtual components on general purpose CPU systems hosted by components directly attached to the TTNoC Interconnect. We refer to such components that provide this runtime environment as host components or simply *hosts* for virtual components. We call our operating system the Time-Triggered Operating System for Virtual Components (TTOSVC) and the runtime environment VCRE. See Figure 4.1a for a high-level overview.

While there is already a great variety of real-time operating systems available – closed and open-source alike – we still decided to create a custom and minimal solution from scratch to take care of special scheduling requirements and make use of the additional hardware functionality provided by the TTNoC Interconnect (global time, message service, timers, ...).

However, in principle any operating system that fulfills the requirements (R1-R8) given in this section is suitable. For example, we also implemented the VCRE for a Linux kernel based operating system and successfully deployed virtual components to it (see Chapter 6).

Requirements

We identify the main requirements of the TTOSVC:

R0 **Provide a Hardware Abstraction Layer (HAL):** The TTOSVC must abstract hardware access such that multiple (embedded) CPU architectures are supported by most of the same code base.

R1 **TTNoC Interconnect Interaction:** The TTOSVC must interact with the TTNoC Interconnect to exchange messages with other components on the TTNoC Interconnect.

R2 **Task Management:** The TTOSVC must support execution of multiple tasks "in parallel" on a single CPU.

R3 **Time-Triggered Task Activation:** The TTOSVC must support execution of time-triggered tasks. Tasks may depend on incoming or outgoing messages in order to process them with lowest delay, a mechanism must exist that activates tasks upon the time-triggered message reception or transmission.

R4 **Priority-Based Preemptive Scheduling:** The TTOSVC must support a preemptive priority-based scheduling strategy such that non time-critical tasks can utilize idle CPU time.

R5 **Memory Management:** The TTOSVC must provide means to manage available memory for tasks.

R6 **Inter-Task Communication:** The TTOSVC must provide synchronisation mechanisms such that tasks can safely and resource-efficiently exchange information.

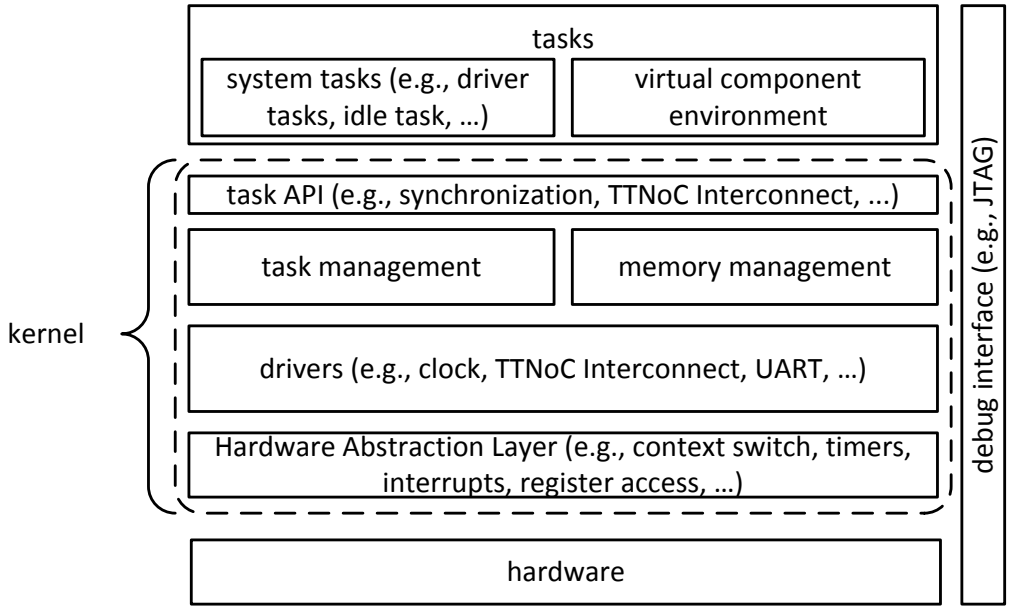


Figure 4.2: High-Level Overview of the TTOSVC

R7 **Driver Framework:** The TTOSVC must support a generic driver framework with generic interfaces to tasks.

R8 **Virtual Component Runtime Environment:** The TTOSVC must provide a runtime environment for virtual components. Virtual components implemented with general purpose hardware as a software program should be able to interact with other (TTSoC) components, virtual or not. Hence a runtime environment must offer an Application Programming Interface (API) to the TTNoC Interconnect services.

R9 **Hot Virtual Component Deployment:** The TTOSVC should support the deployment (upload and execution) of virtual components during runtime. For highly flexible systems, deployment during runtime is favorable.

R10 **Partitioning of Shared Resources:** The TTOSVC should partition shared resources (e.g., CPU time, memory) such that tasks cannot interfere with each other outside their given APIs. This (optional) requirement is similar to the requirement of the TTNoC Interconnect (trusted subsystem) and brings similar benefits: Partitioning of shared resources allows to deploy multiple virtual components with different criticality requirements on the same physical component, if the TTOSVC is certified up to at least the same level of criticality of the virtual component with the highest criticality requirement.

Design and Implementation

Figure 4.2 gives a high-level overview of the TTOSVC design.

Drivers access the hardware of the host (i.e., the CPU system contained in the physical component attached to the TTNoC Interconnect) indirectly through a HAL which gives a uniform interrupt handling mechanism, implements CPU architecture specific context switching and performs endianess and machine word translations. Task and memory management is built above the driver layer and provides the task API. Tasks are structured in system tasks and each virtual component within the virtual component environment also has at least one main task. Through all layers a debug interface is available that enables assertion checking and message passing to the outside world (e.g. through JTAG or serial interfaces).

Task Management

Tasks are described by the task structure which is also often called the process control block [45]. Most importantly, it contains task state information, the task's context and a pointer to its reserved stack memory, scheduling parameters and synchronization/inter-task communication information. A task is always in one of the following available task states: `READY`, `RUNNING`, `TERMINATED` or `WAIT`. Transitions are issued by the TTOSVC scheduler and functions that suspend or resume tasks. Only a single task can take the state `RUNNING` at any time (single CPU core assumption) and only the scheduler is allowed to move tasks from `READY` to `RUNNING`. The task's context, defined in the architecture specific HAL, is able to hold an image of the processor's registers. This enables context switches: i.e., freeze a task, store the processor state to the task's context and resume it later on. In the same frame of reference is the task's stack memory. The HAL must provide a dedicated function to reserve a fixed size of system memory so it can be assigned to a task as stack memory. Depending on which scheduling strategy is used (in the TTOSVC schedulers are pluggable modules that can be exchanged, if desired), it must keep track of scheduling parameters per task: e.g. priorities, maximum slice usage, execution time, deadlines, ... The TTOSVC offers several synchronization mechanisms that also need to store state information on a per task basis.

Task Organization

Tasks are managed in two lists: the *ready* and the *wait list*. Both are sorted and doubly-linked lists implemented in static memory. During task creation, the task is inserted into the ready/wait list by means of insertion sort – the scheduler module provides a compare function that uses the scheduling parameters per task to determine the order relation. On top of the ready/wait list there is always the task that needs to be run next, respectively needs to be resumed next. Concerning static memory usage, all tasks are organized in an array where the array's index corresponds to the unique task id. If a task is moved from one list to the other (it cannot be in both lists simultaneously), no data is actually copied, but only list element pointers are modified. With the current organization we achieve the time complexity described in Table 4.1 (the array index equals the task id, hence search operations are very cheap).

Operation	Time Complexity
task insertion	$O(n)$
find ready task that should run next	$O(1)$
find wait task that should resume next	$O(1)$
move task (ready to wait list or wait to ready list)	$O(1+n) = O(n)$

Table 4.1: TTOSVC Task Management, Time Complexity

Scheduling

The TTOSVC offers support for exchangeable *scheduler modules*. A scheduler module defines the layout of the scheduling parameters (i.e., the C structure), implements the reschedule mechanism and provides a compare function that is used during task insertions into the ready list to establish the sort order of tasks. Each timer event and some of the system calls that modify the ready/wait list triggers the scheduler to reschedule. Rescheduling means taxing the running task and checking whether the top position of the ready list has a higher priority than the task currently running. For our scheduling module, we employ the following simple strategy:

- the task with the highest priority runs
- if there are several tasks with equal priority, apply a round robin scheme and respect time slice configurations

The task scheduling parameters are illustrated in Listing 4.1.

```
struct schd_prio_str
{
    /** priority */
    uint8_t priority;
    /** configured slices */
    uint8_t scfg;
    /** consumed slices */
    uint8_t susage;
    /** total time slice usage */
    uint32_t susage_total;
};
```

Listing 4.1: Scheduler Priority Structure

While the decision about which task should run next is simple, finding the next timeout of the tickless timer is more involved: If there is only one task with the highest priority, no timeout is set up. Otherwise, the next timeout is set to `scfg` slices multiplied with a configurable slice size (`SLICESIZE`). Concerning task accountancy, the reschedule mechanism keeps track of when it has been called last and adds this time (converted to slices) to the running task's scheduling parameters. After a task that shares the same priority with other tasks has used all of its available slices (`scfg`), `susage` is reset to zero and the task is reinserted to the ready list at the end of its priority group (results in round robin). Algorithm 4.1 represents the described scheduler also in pseudo-code. In case the algorithm initiates a rescheduling (resched is set to *true*), a context switch takes place.

constant: The length of a slice based on the used time format: SLICESIZE
global : The last time the rescheduling algorithm was called: $\mathrm{time}_{\mathrm{last}}$
input : The current time: $\mathrm{time}_{\mathrm{cur}}$,
Scheduling parameters of the currently running task: $\mathrm{schedp}_{\mathrm{run}}$,
Scheduling parameters of the top positioned task in the ready list: $\mathrm{schedp}_{\mathrm{ready}}$
output : The reschedule decision resched, The next timeout: $\mathrm{time}_{\mathrm{next}}$

```
1  resched ← false;
2  time_diff ← time_cur − time_last;
3  /* tax currently running task */
4  time_last ← time_cur;
5  schedp_run.susage ← time_diff / SLICESIZE;
6  /* enforce that task with highest priority runs */
7  if schedp_ready.priority < schedp_run.priority then
8  |  resched ← true;
9  |  schedp_ready.susage ← 0;
10 |  time_next ← time_cur + schedp_ready.scfg ∗ SLICESIZE;
11 |  schedp_run.susage_total ← schedp_run.susage_total + schedp_run.susage;
12 else
13 |  if schedp_ready.priority = schedp_run.priority then
14 |  |  if schedp_run.susage < schedp_run.scfg then
15 |  |  |  time_next ← time_cur + schedp_run.scfg ∗ SLICESIZE;
16 |  |  else
17 |  |  |  resched ← true;
18 |  |  |  schedp_ready.susage ← 0;
19 |  |  |  time_next ← time_cur + schedp_ready.scfg ∗ SLICESIZE;
20 |  |  |  schedp_run.susage_total ← schedp_run.susage_total + schedp_run.susage;
21 |  |  end
22 |  else
23 |  |  time_next ← ∞;
24 |  end
25 end
```

Algorithm 4.1: Task Scheduler in the TTOSVC

For reaching a decision, the scheduler has a time-complexity of $O(1)$. If we consider that insertion sort needs to be carried out to place a task back into the ready/wait list, the complexity is $O(n)$.

Intertask Communication and Synchronisation

The TTOSVC offers a few mechanisms to allow safe message exchange (shared memory/message queues) among tasks. There is also the possibility to turn interrupts off/on and do atomic,

guaranteed uninterruptible work (functions in the HAL). We extensively use disabling and reenabling of interrupts for the following synchronization mechanisms.

- **Semaphore:** A semaphore [11] consists of an integer variable and two functions to atomically increase or decrease the integer variable. The user can set the integer variable to a positive value or zero during initialization. In case the integer variable of the semaphore is zero, and a task T tries to decrease the integer variable further, the semaphore implementation suspends task T. Task T gets only resumed in case another task increases the integer variable of the semaphore again. The TTOSVC provides strong semaphores [6], meaning that the task that has waited the longest for the release of a semaphore will be resumed first (FIFO fairness). Internally the semaphore data structure contains a doubly-linked wait list which is processed each time a task increases the integer variable of the semaphore to zero.

- **Mutex:** A mutex can be considered as a semaphore initialized to 1. If a task obtains a mutex, the mutex gets locked and another task that wants to obtain the same lock is suspended until the mutex has been released. However, we implemented the mutex synchronization mechanism not with semaphores in order to save some resources. A mutex has an owner and manages a wait list. If the owner is a valid task, the mutex is locked and other tasks are inserted into the wait list. After the owner releases the mutex, the task that waited the longest becomes the new owner (FIFO fairness).

- **Event:** Tasks can wait on events and/or emit events to resume other tasks. The event data structure contains a 32-bit wide event bit field which is used to represent 32 events. They can be set or cleared by the event set function. The event wait function on the other hand takes an event mask parameter and suspends the calling task which eventually gets resumed when the wait mask matches the event status (event mask & event status = event mask). As events can also be emitted in an interrupt context, events are an elegant method to implement blocking system calls, like read and write functions. We use events extensively for this purpose in all drivers that support those system calls.

Tickless Timer

Instead of the traditional approach, where a periodic timer interrupt (e.g. every 10 ms) advances the operating system's time-base, we use the concept of tickless timers [42]. A high precision timer in output compare mode (e.g., as provided by the TTNoC Interconnect) is only programmed to fire if there is actually something to do, for example rescheduling. While there is slightly additional work to calculate timeouts, in general we expect reduced interrupt load, increased idle times that can be spent in power-saving modes and overall more accurate virtual timers. Currently all timeouts are accurate to the subsecond (highest time resolution the underlying architecture supports). It may prove useful to differentiate between work that requires highest possible accuracy and work that can be grouped to have a common timeout (rounded timeouts). The tickless timer offers an internal API to setup a timeout. Upon every timer event, the next timeout is reset to unlimited. If a subsystem requires a shorter timeout, the next timeout is updated, otherwise the next timeout remains at the old value. That way, the subsystem

requiring the shortest timeout, will actually set the tickless timer. Algorithm 4.1 also shows this concept: the next timeout is set to a value until the scheduler has to be executed again such that no scheduling constraints are violated. In case there is only one task at the highest priority running, the scheduler sets the next timeout $\mathrm{time}_{\mathrm{next}}$ to infinity (see line 23).

There are three sources that influence the next timeout of the tickless timer:

- **Scheduler:** The running task may have a limited run time.
- **Wait List:** A task in the wait list may have a limited wait time.
- **Virtual Timers:** A virtual timer may expire.

System Calls

The TTOSVC provides a set of functions that a task may use to access operating system features. Even though the TTOSVC does not implement memory protection and there is no kernel- or userspace, we refer to those functions as system calls. There are 5 classes of system calls:

- **Time Related:** `ttosvc_time`, `ttosvc_vtimer_register`, `ttosvc_vtimer_unregister`
- **Debug Related:** `ttosvc_printf`
- **Task Management:** `ttosvc_task_create`, `ttosvc_task_sleep`, `ttosvc_task_terminate`
- **Synchronization:** `ttosvc_sem_init`, `ttosvc_sem_v`, `ttosvc_sem_p`, `ttosvc_mtex_init`, `ttosvc_mtex_lock`, `ttosvc_mtex_release`, `ttosvc_event_init`, `ttosvc_event_set`, `ttosvc_event_get`, `ttosvc_event_wait`
- **HAL:** `hal_crit_enter`, `hal_crit_leave`

For details we refer to the Doxygen documentation within the source code.

TTOSVC Internals

We briefly describe some of the TTOSVC internals like startup, how we manage drivers and files, and most importantly how we handle TTNoC Interconnect access.

Startup

The HAL needs to set up a certain environment before the TTOSVC can boot: The function `ttosvc_init` initializes all essential subsystems (e.g. file and driver framework, ready/wait lists, ...). Before the function `ttosvc_start` is called, the HAL needs to initialize important kernel structures and set up the tickless timer. The file and driver framework can be already used before the call to `ttosvc_start`, but only after `ttosvc_init` has been called. This

is useful, if timers are accessed through device drivers: e.g. this approach is used in the Nios II HAL. The HAL must call `ttosvc_start` last. There, all tasks that are known to the system at compile-time are initialized and added to the ready or wait list. Then, the start function elevates the first ready task to the `RUNNING` state and the scheduler is called to eventually setup a timeout for the task in the tickless timer. Finally, the TTOSVC switches to multi-tasking mode by carrying out a bootstrap context switch to the task in the `RUNNING` state. The bootstrap context switch is a special context switch where the old context is not saved.

Drivers and Files

In the TTOSVC drivers are modules that use the macros `TTOSVC_DRIVER_INIT(x)` and `TTOSVC_DRIVER_EXIT(x)` to make initialization and exit functions known. They are put in a special linker section (pointer array), so that they can be called without any additional configuration hassle: if a driver is linked to an `TTOSVC` kernel, it is automatically available. The driver init and exit functions should take care about device initialization and device registration, respectively device closing and device deregistration. We refer to this design-pattern as "self-register mechanism" which is also employed for tasks. Drivers are maintained in a doubly-linked list which makes it easy to traverse them and notify them about important system events: e.g. low power modes, reboots, ...

The term *file* refers to nameable objects where a set of operations is available: open, close, seek, read, write and in-/output control(IOCTL) commands. Files are organized in a linked list of configurable size in static memory. They can be created by the `osttvc_file_add` system call and removed by the `ttosvc_file_remove` system call. When a file is created, it is associated with a unique id, the file descriptor. An open call by `ttosvc_file_open` will first try to find the file in the linked list by string comparison and if successful, return the file descriptor. All other file operation calls (`ttosvc_file_read`, `ttosvc_file_write`, `ttosvc_file_ioctl`, ...) take the file descriptor directly and forward the call to the correct file object. File operations, at least the mandatory open and close system calls, need to be implemented by a file system or a device driver. Currently, there is no file system in the TTOSVC, but we use files extensively for the timer, serial and TTNoC Interconnect drivers.

TTNoC Interconnect Driver

The TTOSVC TTNoC Interconnect driver handles TTNoC Interconnect register access and partitions the port memory. The driver makes great use of the TTNoC Interconnect library that we also developed in an effort to reuse code among different architectures (and operating systems). This library is independent of computing architectures and uses HAL macros to access TTNoC Interconnect registers. Here we will concentrate on TTOSVC specifics. For details concerning the TTNoC Interconnect library, we refer to the Doxygen documentation of our implementation.

The Interconnect Control Device During driver initialization, the Interconnect base address, the port memory base address and the Interconnect interrupt index is obtained from the hardware registry. The driver initializes the Interconnect library and adds the control file: "dev.frontend.ctrl"

to the TTOSVC. Then the Interconnect library's interrupt handler is installed to process Interconnect interrupts. The Interconnect control file implements the following system calls: open, close, ioctl. All other calls will return an error. The open and close functions keep track about call occurrences and ensure that the Interconnect is only opened a single time. The ioctl system call allows to issue the commands described in Table 4.2.

IOCTL	**Description**
TTNOCIC_IOCTL_CMD_SETUP	Setup the current port configuration.
TTNOCIC_IOCTL_CMD_GLB_TIME_GET	Get the global time.
TTNOCIC_IOCTL_CMD_TIMER_SET	Set the timer pattern and mask serviced by the TTNoC Interconnect, TISS.
TTNOCIC_IOCTL_CMD_TIMER_GET	Get the timer pattern and mask serviced by the TTNoC Interconnect, TISS.
TTNOCIC_IOCTL_CMD_TIMER_ENABLE	Enable the timer.
TTNOCIC_IOCTL_CMD_TIMER_DISABLE	Disable the timer.
TTNOCIC_IOCTL_CMD_WTD_LIFESIGN	Give the lifesign.
TTNOCIC_IOCTL_CMD_WTD_PERIOD	Get the watchdog period.
TTNOCIC_IOCTL_CMD_SETUP_APPHNDL	Setup Interconnect interrupt handlers for e.g., reconfiguration, timer, port operation complete, ...
TTNOCIC_IOCTL_CMD_ENABLE	Enable communication service.
TTNOCIC_IOCTL_CMD_DISABLE	Disable communication service.

Table 4.2: TTOSVC TTNoC Interconnect Driver, Control Device IOCTL commands

All ioctl calls are forwarded to the Interconnect library. During the port configuration ioctl call, all previously created port device files are removed and, according to the new port configuration, a port device file per port is created. The naming scheme is: "dev.drontend.ports.[port_id]" whereas `port_id` is the physical port id (see Section 5.3 about port identifiers).

Port Devices Port device files implement the following system calls: open, close, ioctl, read and write. Again the open/close pair enforces that each port is opened at most a single time concurrently. Additionally, the open system call enables the associated port and its port operation complete interrupt. From now on, a read or a write system call, depending on the port's direction, triggers the port operation complete interrupt which is forwarded by the Interconnect library to the driver via a callback hook. If a task wants to read/write a message from/to a port and the port is not ready yet, the task suspends by default and waits on a specific TTOSVC event. When the expected port operation complete interrupt occurs, the hook function in the driver sends the TTOSVC event to unblock a potential reader or writer task, because the port operation complete interrupt indicates that a new message can be sent or received. This mechanism elegantly makes time-triggered task activation available: the task is blocked until a new message can be processed. Message processing is postponed right to the period and instant that is defined in the time-triggered communication schedule. The read/write system calls also offer non-blocking

behavior if the option `OPT_NOBLOCK` is supplied: Then those functions don't block the calling task, but immediately return and success can be checked by evaluating the return value.

The TTOSVC does not have memory protection, so there is no user or kernel space and tasks can write directly to the port memory. For complete control of the Interconnect we provide IOCTL commands for port devices as described in Table 4.3.

IOCTL	Description
`TTNOCIC_IOCTL_CMD_MSG_GETPTR`	Get a pointer to the port memory where a message can be stored or read.
`TTNOCIC_IOCTL_CMD_MSG_COMMIT`	Commit the message: for incoming port this will consume the message, for outgoing ports this will send the message.

Table 4.3: TTOSVC TTNoC Interconnect Driver, Port Device IOCTL commands

Handling Reconfiguration Instants The RMA notifies each component about a possibly pending reconfiguration ahead of the actual reconfiguration instant. This gives the component time to gracefully shutdown old services, get new services ready and prepare the next port memory layout to keep service shutdown and startup times as close to the actual reconfiguration instant as possible. In a new configuration, ports could get added, modified in size (e.g. by degradation) or removed. Each time, the port memory layout needs to accommodate for the change: There always must be enough space for all ports. We identified two extremes to handle the reconfiguration instant with respect to ports: the *revolutionary approach* where the whole port configuration including the port memory layout is wiped and rebuilt for every new configuration, or the *evolutionary approach* where we apply only changes to the current configuration. While the first one is radical and disrupts communication, the second one contains the risk of unpredictable port memory fragmentation over time: Imagine a layout of two ports where the first one is removed from the configuration and another one that requires more space than the first one is added: the free port memory space before the port memory reservation of the second port may never get used again and effectively becomes dead port memory space. This may result in a sudden inability to fit a specific port memory layout into the fragmented port memory, while there would be no problem if there were no gaps between port memory reservations. We decided to strike a balance between those two approaches and distinguish between total and small reconfigurations. A total reconfiguration is carried out according to the revolutionary approach, whereas we treat small ones differently: there, all the configuration including the port memory layout remains constant and ports are enabled/disabled as required. This strategy avoids fragmentation and unnecessary communication disruption, but requires additional information about the reconfiguration.

System Tasks

System tasks provide functionality related to the kernel and have predefined priorities with respect to scheduling. They are required for measuring the system load, keeping scheduling as-

sumptions valid and allowing user interaction with the TTOSVC.

- **Idle Task:** The scheduler always requires to have a task running or ready to run. The idle task has the lowest available priority in the system so it will only run if all other tasks are in the wait list. The idle task is also used to determine system CPU load: During the initialization of the system the task is not interruptible by other tasks and we count how many delay loops are executed until 250ms have passed. We store the result in `idle_cnt_max`. When the system completes initialization and enters the multi-tasking mode (after the `ttosvc_start()` function call), the idle task still counts how many idle loops are taken in 250ms and calculates the overall CPU usage by using the following formula:
$$100 - \frac{\text{idle\_cnt\_cur} \cdot 100}{\text{idle\_cnt\_max}}$$
- **Console Task:** The console task is not required for correct system operation. It prints messages from the kernel ring buffer to an output file. This output file can be a data file in a memory or a serial device file. In case of a serial device file, the console task also provides a small command line interface for debugging and task interaction.

Virtual Component Runtime Environment

The VCRE is a set of library routines and callbacks a virtual component can rely on. The purpose of the VCRE is to encapsulate generic functionality and ease the migration of virtual components among different host components.

Within the context of host components we differentiate between general purpose and special purpose application computers, whereas in this section we only concentrate on general purpose ones. There we expect to have at least a CPU with sufficient memory available that can execute compiled C-programs. The herein described VCRE is implemented for the TTOSVC and as a Linux userspace program that makes use of the Linux TTNoC Interconnect driver. Through the Virtual Component Programming Interface (VCPI) a virtual component can exchange messages with other components (defined in the LIFs), be configured (C/P interface) and offer diagnosis information (D/M Interface). The local I/O Interface and the VCRE directly communicate with the underlying operating system, specifically with its drivers of locally attached devices that can interact with the environment.

Virtual Component Programming Interface

The VCPI is summarized in Table 4.4. The local I/O interface of the virtual component is not part of the VCPI, because we do not want to constrain this interface.

We recommend that virtual components cleanly separate the local I/O interface into their own C-modules, such that only these C-modules need to be ported in case the virtual component is migrated to a different host.

Function	Description
`void VC_INIT(x)`	This macro takes the virtual component initialization function as its single argument and marks it to be placed in a special linker section. This is part of the virtual component self-registration mechanism.
`int vc_register(vc_t* vc)`	Register a virtual component. This is part of the virtual component self-registration mechanism.
`void vc_com_enable(void)`	Enable the message based time-triggered communication service.
`void vc_com_disable(void)`	Disable the message based time-triggered communication service.
`int vc_port_open(port_id_t port)`	Open a communication endpoint.
`int vc_port_close(int port_fd)`	Close a port with file descriptor `port_fd`.
`int vc_port_read(int port_fd, void* buf, const uint32_t size, uint8_t options)`	Read `size` bytes into `buf` from port with file descriptor `port_fd`. By default this call blocks until a message is available. This can be overridden by supplying `VC_OPT_NOBLOCK` as the options parameter.
`int vc_port_write(int port_fd, void* buf, const uint32_t size, uint8_t options)`	Write `size` bytes from `buf` to port with file descriptor `port_fd`. By default this call blocks until a message is available. This can be overridden by supplying `VC_OPT_NOBLOCK` as the options parameter.
`void vc_dbg_printf( const char* fmt, ...)`	Pass a formated debug message to the virtual component runtime environment.
`void vc_time_local(vc_time_t* t)`	Get local time.
`void vc_time_global(vc_time_t* t)`	Get global time.
`int vc_timer(uint32_t ms, uint8_t periodic, void (*timeout_func)(void))`	Register a callback function `timeout_func` to be called either in or periodically every `ms` milliseconds.
`void vc_panic(void)`	Notify the virtual component runtime about a failed critical assertion.

Table 4.4: Virtual Component Programming Interface for General Purpose Hosts

Virtual Interconnect

Virtual components on the same host can also interact with each other via the VCPI. However, messages are not sent/received via the TTNoC Interconnect driver, but are handled directly by the VCRE via intertask communication. We call the part of the VCRE that handles this intertask communication Virtual Interconnect, because it closely resembles the functionality of the TTNoC Interconnect. During a reconfiguration issued by the RMA, the VCRE reserves space for messages within a virtual port memory, directly located in the host's main memory. Message delivery is handled according to the time-triggered schedule of the encapsulated channel's associated PDS and port type (cf. Section 3.3).

- **Sporadic Ports:** In case the queue is not full, the message is copied into the virtual port memory at its correct offset and the sending virtual component's task is not blocked. Otherwise, depending on the send call options (`VC_OPT_NOBLOCK`), the task is blocked until the queue is not full anymore, or the task is not blocked (and the message is not enqueued as well). The receiving virtual component's task is – also depending on the call options – blocked until a sporadic message arrives, or the task is not blocked and continues without receiving a message.

- **Periodic Ports:** Here the reserved memory in the virtual port memory is twice the size of the total message length to reserve space for receiving virtual components and space for the sending virtual component. Those two locations are flipped (pointers, no actual copy) according to the time-triggered schedule of the encapsulated channel's associated PDS. After the flip, all receiving components are able to read the message that the sending component has previously written there. Neither sending nor receiving virtual components are blocked during send and receive operations. Proper synchronization takes care about the message consistency.

If two or more virtual components on the same host are involved in an encapsulated communication channel with other off-host components, the Virtual Interconnect also employs the TTNoC Interconnect driver. However, depending on the processing speed of the host, the virtual components receive messages with (depending on the operating system possibly bounded) delay and respectively need to send messages in advance.

Results and Discussion

Table 4.5 summarizes the lines of C source code that we have written for the implementation of the TTOSVC and the VCRE. We used the tool cloc[1] and did not include blank lines in our count. We implemented support for NiosII, LeonIII and a posix compliant system, hence the rather high number of lines of source code we required to implement the HAL.

In this section we presented the requirements and our implementation of the TTOSVC and the VCRE. Our C-based implementation covers all mandatory requirements R1 to R8 (see Table 4.6). We finished our prototype version during the TTSoC project and successfully deployed it within the mixed criticality use case described in Chapter 6.

[1] http://cloc.sourceforge.net

Module	Files	Comments	Code
HAL	17	2 160	2 596
kernel, system tasks, HAL	74	6 156	6 481
drivers	19	1 394	3 121
VCRE	5	324	556
total (removed duplicates)	93	7 397	9 838

Table 4.5: Source Code Statistics

A conceptually similar implementation of the optional requirements R9 and R10 has been explored in the ACROSS project: R9 is realized by a *boot service* and R10 by using a *partitioning operating system* that guarantees task isolation. Our implementation currently supports only design-time deployment of virtual components and due to the missing task isolation, all virtual components (and of course also the TTOSVC) must be certified to at least the same criticality of the virtual component with the highest criticality requirement.

Another interesting feature heavily exploited by various virtualization architectures is code mobility [17]. The ability to move virtual components between different host components possibly located in different TTSoC chips during runtime and without service downtime could bring a lot of benefits: e.g., masking physical component faults, system repair without downtimes or load balancing.

requirement	rationale/notes	result
R0	The TTOSVC uses for all hardware accesses C preprocessor macros that can be defined on a per-architecture basis. Further, startup code, context switching and interrupt handling is also on a per-architecture basis configurable.	✓
R1	The TTNoC Interconnect driver provides means to interact with all the accessible functionality (i.e., most importantly the time-triggered message transport service) of the TTNoC Interconnect.	✓
R2	Multiple tasks are supported to run in 'pseudo' parallel. Tasks that have the same priority and are not waiting on events receive processor time according to their time-sharing configuration in a round-robin scheme.	✓
R3	The TTOSVC employs a priority preemption scheduling strategy and provides events as a means to synchronize tasks. Virtual components enjoy a high priority and the TTNoC Interconnect driver is able to block tasks until time-triggered messages are ready for transportation. As soon as a time-triggered message can be sent or received (port complete interrupt is directly tied to the periodic control system), the waiting task is activated, hence the activation is time-triggered.	✓
R4	The TTOSVC support priority-based preemptive scheduling.	✓
R5	Memory can be dynamically allocated by using a third-party libc (e.g., newlib). Low-level support is provided in the HAL (`sbrk()`).	✓
R6	The TTOSVC supports semaphores, mutex, events and shared memory for inter-task communication.	✓
R7	The TTOSVC has a generic driver framework which makes devices available as files to tasks.	✓
R8	The TTOSVC contains an implementation of the VCRE which provides the VCPI. Hence virtual components can be executed.	✓
R9	There are no means to deploy virtual components during runtime.	×
R10	The TTOSVC is not a partitioning operating system. In favor to support simple microcontrollers and CPU system, the TTOSVC does not require or use memory protection.	×

Table 4.6: Validation of Requirements

4.3 Topology Invariant Scheduling of Pulsed Data Streams (PDSes) in the TTNoC Interconnect

In the TTSoC Architecture ECCs represent data transport relations between components. An ECC is a logical, unidirectional communication channel between one sending and one or multiple receiving components. Encapsulation, i.e., temporal and spatial isolation of these channels is of paramount importance: Any two ECCs must not interfere. Hence, it is one of the most critical requirements of the trusted subsystem to provide physical resources and a conflict-free communication schedule to establish the prescribed ECCs among components. The trusted subsystem consists of the physical communication resource[2] and the elements involved to create and verify conflict-free schedules. For example, in a single TTSoC chip, the TTNoC Interconnect and the RMA are the trusted subsystem. In case there are multiple TTSoC chips that are interconnected by off-chip communication through TTE gateways, then those gateways and the whole TTE infrastructure attached to the gateways are also included in the trusted subsystem.

In this section we concentrate on scheduling PDSes through the TTNoC Interconnect. This means we only solve the scheduling problem for on-chip ECCs. We refer to these ECCs as *single-hop* ECCs, because only a single time-triggered communication resource is involved. In chapter 5 we explore the possibility to build *multi-hop* ECCs where multiple time-triggered communication resources and gateways are involved.

The Scheduling Problem: Parameters and TTNoC Interconnect Constraints

The communication schedule required by the TTNoC Interconnect consists of a set of PDSes. A PDS is defined by its, period, a common delivery instant (or phase) and a set of fragments[3] where each fragment has its own send instant and a route from sender to receivers. These parameters associate a single PDS to a specific set of TDMA slots within the TTNoC Interconnect and it is the purpose of the scheduling algorithm to take high-level described PDSes (see Section 2.1) that are only constrained in the time and space domain and find a conflict-free low-level description.

We define the input parameters for the scheduling problem as a set of PDSes whereas we require for each PDS the parameters:

- **Unique Channel Identifier:** Channels need to be uniquely distinguishable while solving the scheduling problem.
- **Priority:** Different criticality requirements can be expressed by a scheduling priority parameter. Resource change requests of high-critical distributed applications might result in a partial solution where some low-priority channels of a non-critical distributed application are not schedulable. In that case, a degraded mode of the non-critical distributed application might work or in the extreme case the non-critical application is shut down.
- **Sender:** The sender is the origin of the PDS.

[2] Strictly speaking, also other shared resources (like power) and their management are part of the trusted subsystem, but with respect to this thesis we are concerned with shared communication resources only.

[3] In the context of PDSes we use the terms fragment and burst as synonyms.

- **Set of Receivers:** The set of receivers are the destination of the PDS.
- **Period:** A fundamental parameter that determines the PDS's pulse periodicity.
- **Send Instant:** This parameter needs to be given as a relative period offset or phase. The send instant determines the deadline at the sender when a message needs to be put into the port memory.
- **Receive Instant:** This parameter needs to be given as a relative period offset or phase. The receive instant constraints the maximum TTNoC Interconnect transport delay. A large delay gives more freedom to schedule the PDS in several bursts or fragments.
- **Length in Flits:** This parameter determines the size of the message.

The output of the scheduling algorithm is a set of scheduled PDSes that additionally contain a:

- **Set of Fragments Associated to each Schedulable PDS:** The set of fragments has a total number of flits that is equal to the length of the PDS. In case send and receive instant allow to fragment a PDS into multiple bursts, different PDSes can be *interleaved*.
- **Route for each Fragment:** A route is a list of fragment switches that should be taken to reach all receivers from the sender.

Thus, a route gives spatial freedom and fragmentation gives temporal freedom during the process of scheduling.

Besides input parameters, the scheduling algorithm needs to respect implementation specific constraints. The periodic control system of the TTNoC Interconnect has the following design-time configurable parameters related to the digital time format which must be obeyed by the set of input PDSes:

- **Supported Periods:** Naturally, only PDSes that are supported by the periodic control system within the TISS can be scheduled.
- **Highest Supported Period Bit $\mathcal{N}$:** This is the index of the most significant bit of the highest period that the periodic control system in the TISS supports.
- **Period Delta δ:** This parameter is the distance between two period bits which determines how many lower periods are contained in a higher period.
- **Width of Phase Slices:** Phase slices determine the periods phase granularity. Valid phase slice sizes for any period are larger than zero.
- **Macrotick Bit $\mathcal{M}$:** As introduced in Section 2.1, this bit position in the time format is toggled by the macrotick. All bits right to the macrotick bit $\mathcal{M}$ are never updated and are stuck to zero, i.e., they are dead bits.

In case a phase slice crosses the macrotick bit, its width is reduced by the number of bits that are right to the macrotick bit. In Figure 3.4 the period delta $\delta = 1$ and supported periods are all periods between $\pi_6 = 2^{-24}$ (phase slice size is one) and $\pi_{31} = 2^1$. The highest supported period bit is 32. The phase slices are 12 bits wide, unless they start to cross the macrotick bit. For example, for period π_{16} the phase slice is only 11 bits wide.

Concerning the NoC and routes, we need to respect the constraints described in Section 3.3. Most importantly:

- No two input lanes of a single fragment switch can be set to the same output lane simultaneously.
- Each TISS (i.e., the endpoint of the TTNoC Interconnect) can either exclusively receive or send, but can not do both operations simultaneously.
- Routes can only fork within fragment switches directly attached to TISSes.

The output of the scheduling algorithm, i.e., the set of scheduled PDSes cannot be arbitrarily large. At each TTNoC Interconnect endpoint a subset of the whole communication schedule is deployed to configuration memories. The subset is determined by the endpoint's involvement as a sender or receiver. In case of a sender, also at least one route must be stored. Naturally, the number of flit bursts (i.e., the fragments) and the number of routes is limited to design-time configurable memory sizes.

Further, there are fundamental latencies (i.e., not configurable during design-time) in the implementation of the TTNoC Interconnect that need to be considered for scheduling and schedule verification. Table 4.7 summarizes them with respect to the current prototype implementation [37]. Latencies are listed in TTNoC Interconnect clock frequency ticks.

operation	**latency**
receive burst	7
send burst	7
fragment switch passing (hop)	1
routing flits	1

Table 4.7: Latencies in the TTNoC Interconnect

The smallest unit of data that is transported by the TTNoC Interconnect is a flit of size b bits. In the TTNoC Interconnect fragment switches transport fragments (i.e., packets composed of a series of flits) from a source to one or more destinations. A fragment might contain a header of routing flits. A routing flit consists of routing processor opcodes that configure the output multiplexers of the input port where the routing flit arrived. In a mesh network topology every fragment switch has n ports, the size of a single routing opcode is exactly n. Hence we require a routing flit per every $\lceil \frac{b}{n} \rceil$ hops within the NoC.

Finally, we make the following assumptions concerning the TTNoC Interconnect:

- **Limited Macrotick Rate Adaption:** Due to time synchronization, the macrotick may be slowed down or accelerated by an integer divisor of the macrotick. This divisor must

be limited per macrotick: For example, assume the macrotick is generated by a clock $\mathcal{C}$ which is 16 times faster than the macrotick. In this case a sensible rate adaption limitation would be to allow a macrotick rate adaptation of at most two clock ticks of $\mathcal{C}$. Then every 14-18 ticks of $\mathcal{C}$ a macrotick must occur.

- **No Route Length Constraints:** The receive window size of all receiving parties ensures correct reception regardless of the route length or the size of the NoC. The receive window must remain open long enough to compensate for all introduced latencies, while send and receive instants also must remain in the same macrotick.

TDMA Slot Allocation

The scheduling algorithm needs to operate at the level of TDMA slots, hence, methods are required to allocate TDMA slots for a PDS. In the time domain, all available bandwidth is partitioned in TDMA slots, whereas the size of a single TDMA slot is exactly one macrotick. Of course, many flits might fit in a single TDMA slot, because the TTNoC Interconnect clock rate is usually much higher than the macrotick rate. Depending on the PDS's properties, one periodic TDMA interval needs to be allocated per PDS fragment. The start of the first TDMA interval is determined by the:

- PDS period π_n and instant ϕ_k,
- highest supported period N, i.e., the highest supported period is π_N,
- highest supported period bit position $\mathcal{N}$ in the digital time format,
- macrotick bit $\mathcal{M}$, i.e., the bit position of the macrotick in the digital time format,
- period delta δ, and
- phase slice width of a period π_n, denoted as Δ_n.

The phase slice width Δ_n of a given PDS period π_n defines the exact TDMA slot distance between two consecutive instants of the same period. We refer to this distance as the *TDMA slot granularity* γ_n for a period π_n. Note that the TDMA slot granularity is different for each period, as the phase slice width Δ_l stays at most constant (until the macrotick bit $\mathcal{M}$ is crossed) across all periods, while the periods themselves get smaller/lower/faster when we move towards the macrotick bit $\mathcal{M}$. If a phase slice crosses $\mathcal{M}$, the passing bits are cut off, so that the granularity γ_n cannot get lower than a single TDMA slot. Formula 4.1 describes how the granularity for a specific period can be calculated. The function $\mathrm{bitpos}(\pi_n)$ considers the period delta δ and returns the correct bit position in the time format of a period π_n.

$$\gamma_{\mathrm{n}} := 2^{[\mathrm{bitpos}(\pi_n)-\mathcal{M}-\Delta_n+1]}. \tag{4.1}$$

For example, if you consider Figure 3.4 again, the TDMA granularity for π_{16} is:

$$\gamma_{16} = 2^{17-6-12+1} = 1.$$

Every supported period that is shorter than or equal to π_{16} results in a granularity of 1. As a last example, for π_{17} the granularity is 2, i.e., not every natively supported TDMA slot can be 'addressed' at this (and any higher) period anymore.

The TDMA start slot is the granularity at period π_n multiplied by the instant ϕ_k (cf. Formula 4.2).

$$\mathrm{tdma\_start\_slot}(\pi_\mathrm{n}, \phi_\mathrm{k}) := \gamma_n \cdot \phi_k. \tag{4.2}$$

Note that we only obtain the first occurrence within the defined periodic control system. Each PDS period length there is another appearance of the same PDS burst. The number of occurrences of a period at the level of TDMA slots can be calculated with Formula 4.3.

$$\mathrm{tdma\_period\_reoccurrences}(\pi_\mathrm{n}) := \frac{2^{N-\mathcal{M}}}{2^{N-\mathrm{bitpos}(\pi_\mathrm{n})}} = 2^{\mathrm{bitpos}(\pi_n)-\mathcal{M}}. \tag{4.3}$$

Formulas 4.2 and 4.3 return for any given PDS fragment the start of each TDMA allocation. The length of such an allocation is determined by the relation of the TTNoC Interconnect clock frequency to the macrotick clock frequency and the size of the PDS fragment.

Managing Pulsed Data Streams (PDSes)

For the scheduling algorithm, we want to organize already scheduled PDSes such that we are able to obtain the TTNoC Interconnect usage for any given time duration, i.e., an interval of consecutive TDMA slots. Hence, we need to track the complete interconnect state and the used TDMA slots for each PDS. We solve this problem with two lists: the *TDMA burst list* where we keep track of all PDSes on the TDMA level and the doubly-linked *TDMA usage list* which we require to correctly model the burst's periodicity. There is a 1 to n relationship between elements in these two lists. Each element in the TDMA burst list contains:

- the length of the TDMA interval,
- the fragment switch configuration (TTNoC Interconnect utilization) of the burst's route,
- a link to the corresponding PDS, and
- a flag that indicates whether the burst has been successfully scheduled.

The length of the TDMA interval is the total amount of TTNoC Interconnect clock ticks that a flit requires for transport from sender to all receivers. As we have to account for different latencies (cf. Table 4.7), we need to keep track of the TDMA interval sizes in TTNoC Interconnect clock ticks.

We store the *fragment switch configuration* in an array that contains all fragment switches within the TTNoC Interconnect. Each fragment switch is encoded according to Figure 4.3 with a series of two bits describing incoming and outgoing lanes (shown as arrows) of a single switch port: e.g. a 4-port switch has four such 2-bit series representing the state of the lane and bit 4 represents the incoming lane of port 2. If the bit is cleared, the lane is free, otherwise it is in use. The sequence of the ports is clockwise around the switch. We do also store in the fragment switch configuration the switch utilization during the progression of the route itself: Each passed

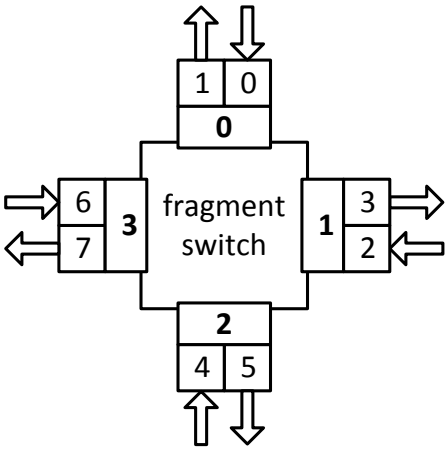


Figure 4.3: Encoding Scheme for Port Usage of 4-port Fragment Switch

switch gets a TTNoC Interconnect clock tick timestamp that is relative to the start macrotick of the TDMA interval. The timestamp marks the start of the TDMA interval from the switch's point of view. In the TDMA usage list, we keep track about what TDMA intervals are occupied by which TDMA burst. This list is sorted by the start of the TDMA interval in ascending order. Each element contains:

- the start of the TDMA interval and
- a link to the corresponding TDMA burst element.

Our input is a communication schedule broken down to its *PDS fragments*. A message in the concept of a PDS is a single pulse consisting of one or more PDS fragments that are in the same period. For each PDS fragment we add an element to the TDMA burst list. The TDMA interval is calculated according to the size of the PDS fragment, send/receive setup time, network latencies and safety space to accommodate for an eventual macrotick rate adaption. During the insertion into the TDMA burst list, we also amend the TDMA usage list: We add one or multiple TDMA usage elements consistent to the burst's periodicity (e.g., if a period is repeated n-times in the longest supported period, there are n elements inserted at the correct positions). Each start of the TDMA intervals is calculated according to the previously given formulas.

Network-on-Chip (NoC) Utilization

After all PDS fragments from the input set of PDSes are added to both TDMA lists, we require a way to obtain the NoC utilization for any given TDMA interval. By means of binary search we first try to find a TDMA usage list element which points to a TDMA burst that is already scheduled and overlaps with the given input interval. If there is no overlapping element, the NoC utilization is zero. Otherwise, we traverse the sorted list starting from our match, both upwards and downwards; until we encounter the end of the list or an element that does not overlap with the given input interval. For example, Figure 4.4 depicts a situation, where we search for all

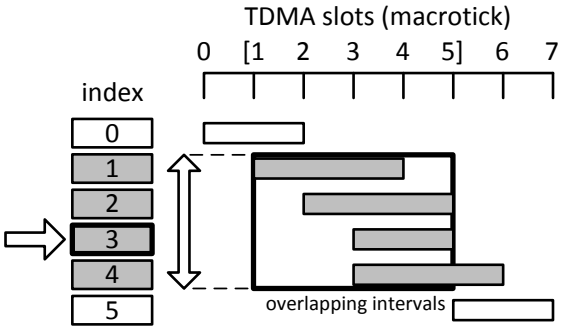


Figure 4.4: TDMA Slot Usage Search Procedure

bursts that overlap with macrotick 2 to 5. In this example the search returns the elements 1 to 4 of the TDMA usage list. When we traverse the list, we only consider so far scheduled bursts as contributions to NoC utilization which saves us some effort ($\frac{n(n-1)}{2}$ instead of $n(n-1)$, but still $O(n^2)$). Depending on the size of the NoC, available RAM and CPU power of the RMA, there are different options available to combine all affecting TDMA intervals to a complete view on NoC utilization:

- **Diffuse:** We use binary OR to combine each single NoC utilization information (switch configuration array) from the matches and do not care about individual switch TDMA intervals. While this is the fastest option with the smallest memory requirements, there is the disadvantage of not being able to tell where exactly within the TDMA interval a switch port is occupied. With this restriction, routes which would collide in the space domain cannot be placed ”too close” in the time domain, because they appear as collisions to the scheduling algorithm. Ultimately, the tradeoff is that the scheduling algorithm cannot use all available margins in the time domain.

- **Accurate:** In contrast to the diffuse option, each contributing TDMA interval is added to a NoC utilization list where all information about single switch TDMA interval offsets remain available. The scheduling algorithm can use all margins in the time domain and is able to place routes as close as possible. Later, when we check switch utilization against a desired route for collisions there is the drawback of a considerable comparison effort.

Figure 4.5 illustrates the difference between the two approaches: the solid and the dashed lines are routes that represent PDSes with the size of a single flit that have the same period and instant (i.e. they start at the same macro tick). When we choose the diffuse option, a (false) collision is detected, because the information in the white circles (TTNoC Interconnect clock tick offsets from the start TDMA slot) is lost. The accurate option has the clock tick offsets available and although the routes cross in the lower left fragment switch, they do so at different time instants. Hence it is determinable that the two routes are collision-free.

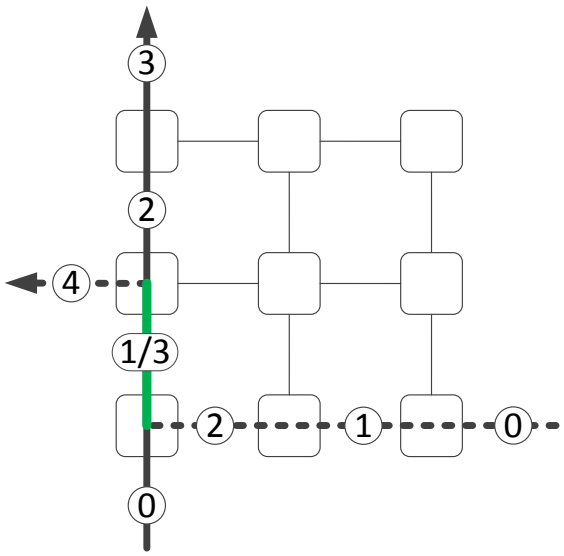


Figure 4.5: Diffuse versus Accurate NoC Utilization Representation

Routing and Collision Checks in the Network-on-Chip (NoC)

We model the complete NoC by means of a traversable, directed graph that can represent an arbitrary topology. Each directed edge acts as a single lane and each node represents a fragment switch or a TISS in the TTNoC Interconnect. Lanes contain information about source and target switch ports that they connect. We need this information to be able to generate routing opcodes. To find the shortest-path between two given TTNoC Interconnect endpoints, we chose the Dijkstra algorithm [4] because of its beneficial time complexity and the custom weight function feature where a graph edge weight can be dynamically calculated by a custom weighting function. With the Dijkstra algorithm, we remain NoC topology independent and are able to route PDSes in the space and the time domain. The Dijkstra algorithm will find optimal unicast routes in polynomial time and with a small extension to the algorithm, multicast routes as well. Multicast routes in the current NoC implementation are severely limited by wormhole routing where no forks are allowed. Hence, we cannot just build a shortest paths subgraph from the source node to all destination nodes and expect that the Dijkstra algorithm will avoid forks. Instead, our algorithm takes the shortest path to the nearest receiver and continues from there to the next nearest receiver until all receivers are reached. During this operation the NoC utilization must be updated such that the resulting overall route remains valid. In case the algorithm is unable to route a PDS, a *scheduling conflict* is created. This scheduling conflict contains all conflicting routes and their root cause sorted according to their size. A possible conflict resolution strategy is to split larger PDSes' bursts such that smaller PDSes' bursts can fit in between (i.e., pulse interleaving, see Section 2.1).

The Scheduling Algorithm

Algorithm 4.2 outlines our scheduling algorithm which makes use of the previously introduced data structures and utility algorithms. Our implementation of the Dijkstra algorithm calls the weight function on each attempt to take a hop to a directly connected node (fragment switch or TISS). Now, the weight function is able to influence routing by either reporting that the next hop is cheap, expensive or even blocked. The Dijkstra algorithm will find an optimal route with respect to any constraints we may impose. The most important constraints originate from collision checks: Before we attempt to place a route, we can find out about all other previously scheduled PDSes that make up the NoC utilization during the TDMA interval of the current PDS. Then during scheduling, the weight function gets called by the Dijkstra algorithm with the source node, the lane and the destination node. For collision-free scheduling, we need to check whether the outgoing port on the source switch and the incoming port on the destination switch are both unused. Here we can also verify that a TISS is not involved in simultaneously sending and receiving operations, by checking that source or destination nodes that are TISSes have only either their incoming or their outgoing lane active (but never both). In case of a collision, the weight function blocks the hop. In case a PDS should be only verified, the weight function blocks all lanes beside those that are defined in the proposed scheduling information. If one of the receivers is not reached or there is a block on the suggested route because of a collision, the route is invalid and schedule verification fails.

In case we try to lay a unicast route through the NoC, NoC utilization determines blocked and unblocked lanes and the Dijkstra algorithm will find an optimal route through the NoC. However, as we only make optimal routing choices per added PDS burst, we might not end up with the overall possible lowest weight of all routes (local versus global optimality). With the multicast extension, the Dijkstra algorithm will find optimal non-forking paths where multiple receivers are present.

global : TTNoC Interconnect utilization: nocutilization (used by `WeightFunction`).
input : Period ordered set of time-constrained PDSes $\mathcal{I} = \{\mathcal{P}_1, \mathcal{P}_2, \ldots\}$,
TTNoC Interconnect parameters: P.
output: Set of scheduled PDSes: $\mathcal{O}$.

```
1  O ← {};
2  noc_graph ← BuildNoCGraph (P);
3  InitList (tdma_usage); InitList (tdma_burst); InitList (conflict);
4  for p in I do
5      pds ← p; InitList (pds.bursts);
6      for b in p.bursts do
7          periodrepeats ← TDMAPeriodReoccurrences (b.period);
8          periodspacing ← P.allTDMASlots/periodrepeats;
9          periodoffset ← γ_[b.period] · FindFreeInstant
           (b.period, b.instant_constr, b.size);
10         nocutilization ← NoCUtilization (tdma_usage, periodoffset, b.size);
11         if |p.receivers | = 1 then
12             new_burst.route = DijkstraShortestPath (noc_graph,
               WeightFunction, p.sender, p.receivers);
13         else
14             new_burst.route = FindMulticastRoute (noc_graph, WeightFunction,
               p.sender, p.receivers);
15         end
16         if route is found then
17             for i in 1 to periodrepeats do
18                 new_usage.burst ← new_burst;
19                 new_usage.start ← periodoffset;
20                 AddListElemSorted (tdma_usage, new_usage);
21                 periodoffset ← periodoffset + periodspacing;
22             end
23             AddListTail (pds.bursts, new_burst);
24             AddListTail (tdma_burst, new_burst);
25         else
26             O ← ResolveConflict (O, tdma_usage, tdma_burst, new_burst,
               nocutilization.bursts);
27         end
28     end
29     O ← O ∪ {pds };
30 end
31 return |O | = |I |;
```

Algorithm 4.2: The Topology Invariant Scheduling Algorithm

Employing the Scheduling Algorithm for Schedule Verification

In case we want to verify whether a set of PDSes is scheduled correctly, the very same scheduling algorithm can be employed with superficial changes.

First we extract all TDMA bursts from the proposed schedule and add them to the TDMA burst list and TDMA usage list. Then we start to iterate through the TDMA burst list and verify each entry against already verified routes by trying to follow the suggested path through the NoC: For every TDMA burst, we know the total active TDMA interval and can obtain the NoC utilization around that time. The weight function uses the NoC utilization data for collision checks. If the TDMA burst does not collide with any other already processed burst, verification has been successful so far. In case the algorithm verifies all PDSes against all others, the whole verification is successful.

Implementation and Validation

We implemented the proposed scheduling algorithm with the exception of conflict resolution and the limitation that instants are given and not only constrained. Our first implementation for early testing and for calculating static schedules is realized in Java. We also implemented the scheduling algorithm in C and successfully deployed it as a virtual component on the RMA. Additionally to the sole scheduling algorithm our implementation also contains the functionality to structure the found schedule according to the TTNoC Interconnect requirements such that the RMA can carry out reconfigurations. Concerning validation, we created a small test system that generated both systematically and randomly input PDSes for the same TTNoC Interconnect parameters that we use in Chapter 6 for the case study: i.e., 3x2 mesh topology, 4-port fragment switches, 10 components, $N = 16$, $\delta = 1$, $\mathcal{M} = 11$, $\mathcal{N} = 32$. We used the systematic generation for runtime estimates and the random generation for rudimentary scheduler validation. At first glance at the scheduling Algorithm 4.2 one might think that time complexity is in the order of the Dijkstra algorithm. However, we have multiple runtime determining input parameters: the NoC graph and the number of pulses which might exponentially grow when allocated as TDMA slots. Table 4.8 gives a small impression about this circumstance. We measured the runtime of the algorithm on a systematically generated input set of 1 500 PDSes with random sender and receiver (unicast only). For systematic generation we constrained all PDSes to be evenly distributed throughout the whole period (all PDSes are $\frac{\text{PHASES\_MAX}}{1500}$ apart). For random

π_n	0xf	0xe	0xd	0xc	0xb	0xa	0x9
execution time (s)	0.02	0.04	0.17	0.64	2.57	10.12	43.21

Table 4.8: Exponential Runtime in Period to TDMA Slot Allocation

generation, we used PDSes where the period, instant, size, sender and receiver(s) are random (uniformly distributed). We added as many PDSes as possible before a collision occurred and checked this collision manually. We repeated this test more than 40 times with different seed values for the pseudo random number generator. Further, we collected all found routes and val-

idated them with a simple route checking tool that just verifies if a route in the schedule indeed goes from sender to all receivers.

These results support that our scheduling algorithm is able to produce valid schedules. However, we did not test extensively and reserve this for future work. Another challenge – also reserved for future work – is to find a feasible conflict resolution strategy and look into stochastic modeling/heuristics to guide the search for valid instants within the given instant constraints. Additionally, the TDMA allocation is in the worst case exponential and increases the number of bursts that need to be considered for collision checking tremendously. A more intelligent strategy (e.g., only check relevant TDMA usage elements) or a cleanup strategy (e.g., combine smaller overlapping TDMA usage elements to larger ones) might be beneficial. However, solving the scheduling problem for arbitrary topologies and multiple periods remains a **difficult** problem. Relevance of our proposed solution might only be in building an optimal or near-optimal offline scheduler to verify topology depending solutions that run reasonably time-constrained on-chip.

4.4 Discussion

We applied the recursive component concept to the TTSoC Architecture and introduced the concept of virtual components. The main benefits of the recursive component concept are to elegantly structure systems such that they scale with respect to complexity, reliability, performance and power. Then we introduced Time-Triggered Operating System for Virtual Components (TTOSVC) where we also presented our Virtual Component Runtime Environment (VCRE). An interesting feature of the TTOSVC is its tickless timer implementation that is also used in the Linux kernel. The VCRE allows for the execution of multiple virtual components on a single host component. Lastly, we covered the problem of scheduling ECCs for the TTNoC Interconnect. We presented the design and a proof-of-concept implementation that although solving the problem also outlines the difficulties in doing so. Unconstrained runtime scheduling of critical distributed applications might never be feasible. However, this is not a problem: The TTNoC Interconnect topology can safely be assumed to be static. A practical implementation might only include a few base periods and/or structure the overall TTSoC chip design in a way such that there are multiple smaller TTNoC Interconnects tailored exactly to the connected component's period requirements instead of one large TTNoC Interconnect that needs to support all periods at once. Additionally, the set of PDSes of critical applications can be scheduled before runtime. Then, a runtime scheduler only needs to schedule non-critical applications around the critical application. Consequently, the RMA can take its time to build a schedule for mode switches of non-critical applications and only switches when a valid schedule is found.

In conclusion, we provided some advancements of the TTSoC Architecture that mostly concern its implementation and aim towards practical application development. In fact the development approach that we present in the next chapter, specifically the tool support described there, produces configuration and C stubs that can be directly used to build virtual components for our VCRE implementation and our scheduler located in the RMA.

CHAPTER 5

Application Development Approach

In the TTA, interface design is central to building composable systems. Precisely defined interface specifications determine the interaction between components and indirectly also determine the behavior of components. This interface design can be done at multiple abstraction levels, ultimately leading to the complete definition of the system behavior. Hence, the TTA design methodology [28] suggests to carry development out in two consecutive phases: architecture design and component design. The architecture design concerns functional decomposition of the overall system into components and their LIFs. The component design regards how a component implements the previously defined LIFs. For larger designs with focus on integration and dynamic reconfiguration we believe that lessons learned from the Service oriented Architecture (SoA) are beneficial to manage the increasing complexity.

In this chapter, we describe a service-oriented development approach tailored to the TTSoC Architecture. Parts of this approach are only TTA specific and might be useful for other TTA based architectures.

5.1 Requirements

Based on the goals of the TTSoC Architecture (see Chapter 3), we derive the main requirements for our development approach:

R0 **Integration of Multiple Distributed Applications:** Usually, the set of distributed applications that can be integrated on a TTSoC is not designed by a single vendor. Naturally, vendors employ many domain- and even company-specific development approaches/processes that are suitable to support designing their specific distributed application. Thus, the development approach shall aid the integration of multiple, heterogeneously developed distributed applications.

R1 **Creation of Time-Triggered Distributed Applications:** All distributed applications must eventually execute under the time-triggered execution paradigm. The development approach shall primarily consider the engineering of time-triggered distributed applications.

Distributed applications that are not designed as time-triggered applications must be re-designed.

R2 **Manage Shared Resources:** Usage of shared resources like power, the TTNoC Interconnect or chip area must be managed at a global scale such that all requirements of individual distributed applications are satisfied. The development approach shall aid in the creation of necessary schedules and their deployment as resource configurations on TTSoC based systems.

R3 **Support Separate Development:** The development approach shall support the (re)use of existing, prevalidated components as building blocks to build larger systems. Suppliers like Original Equipment Manufacturers (OEMs) should be able to offer COTS products that contain their preverified IP as a library to the engineer's disposal during application development.

R4 **Instantiate Non-Functional TTSoC Architecture Features:** The development approach shall employ TTSoC Architecture mechanisms to fulfill dependability and quality of service requirements in a generic way. Ideally, non-functional requirements of distributed applications deployed on TTSoC based systems are only specified but not designed.

R5 **Support for System Runtime-Verification:** The TTSoC Architecture is able to collect general error-information (e.g., missed deadlines to send/receive a message, missed watchdog deadline, ...), but also to collect application messages. The development approach shall support the creation of application specific run-time verification mechanisms or strategies.

R6 **Support for Dynamic Resource Management:** The RMA can process resource change requests to reconfigure shared resources. The development approach shall support the creation of applications with multiple operational modes, i.e., applications that can adapt to changing demands of their environment.

5.2 The Model-Based TTSoC Architecture Development Approach

Besides a vast solution space for the previously stated requirements, there is also the difficulty to evaluate development approaches with respect to properties like efficiency, usability or more general 'quality'[1]. Unlike many algorithmic problems where some metric of optimality is available, development approaches are methodologies or sets of processes and guidelines that help (typically human) engineers to produce an object of creation with desired characteristics. These methodologies arise through experience – in a way they are a manifestation of preserved experience, often even domain-unspecific experience. For example, the process of modeling where reality is reduced to a model. A model is an abstraction of the modeled reality and contains only details relevant to the purpose of the model. The modeling process is a powerful tool for

[1] We like to refer to Pirsig's epiphanies [38] concerning his inability to define what quality exactly is.

knowledge acquisition, making predictions, and – most relevant to us – engineering. Luckily, in our situation we do not need to start from scratch. The earlier history of development approaches [28] [20] suggested for application development in other versions of the TTA and Salloum's work concerning a development process in the TTSoC Architecture [14] already provide a good starting point. It remains for us to adopt and extend them for designing distributed applications for the TTSoC Architecture where integration of many distributed applications and their possible runtime reconfigurability are more prominently supported than in earlier TTA based architectures.

We give a short summary of the methods contained in the existing development approaches:

- **Dominating Two-Phases Approach:** In the *architecture design phase*, the system is decomposed into components and their communication relations. Based on this decomposition the LIF of each component is derived. The LIF serves as a component's contract with the system and the architecture. In the *component design phase*, component internals are built such that the component's contract is fulfilled, i.e., the component is in accordance to its LIF.

- **Employing Model-Driven Development (MDD):** Huber, Obermaisser and Peti adopt in their work [20] the Model-Driven Architecture (MDA) methodology for the TTA. They propose a way to describe system functionality (i.e., the distributed applications) as a PIM and a meta-model that allows to design TTA compliant platforms. Salloum later contributes in his thesis [14] a more detailed view on the Platform Independent Model (PIM) which supports his proposed naming scheme and refines the PIM up to a level where it barely remains platform independent. All those techniques are based on a logical two level abstraction hierarchy (see Section 4.1): DASes and jobs.

Another interesting (software) development approach is the work of Hutchinson et al. about a service model for component-based development [22]. They propose to implement requirements raised by viewpoints in services. The services are mapped to components. In their work, a service is an entity with a certain behavior and non-functional constraints that drive the mapping to COTS, possibly black box software components. For our thesis, the most important benefits of their proposed service-oriented model are:

- There is traceability of every service and component back to the originating requirement.

- Functional and non-functional requirements are integrated.

- There is a powerful partitioning of the system architecture into services at a very high platform independent level.

- The service behavior specification is flexible with respect to how this description is accomplished.

- Executable service models can be used to verify system behavior.

- Mixing of third-party and self-developed components is possible.

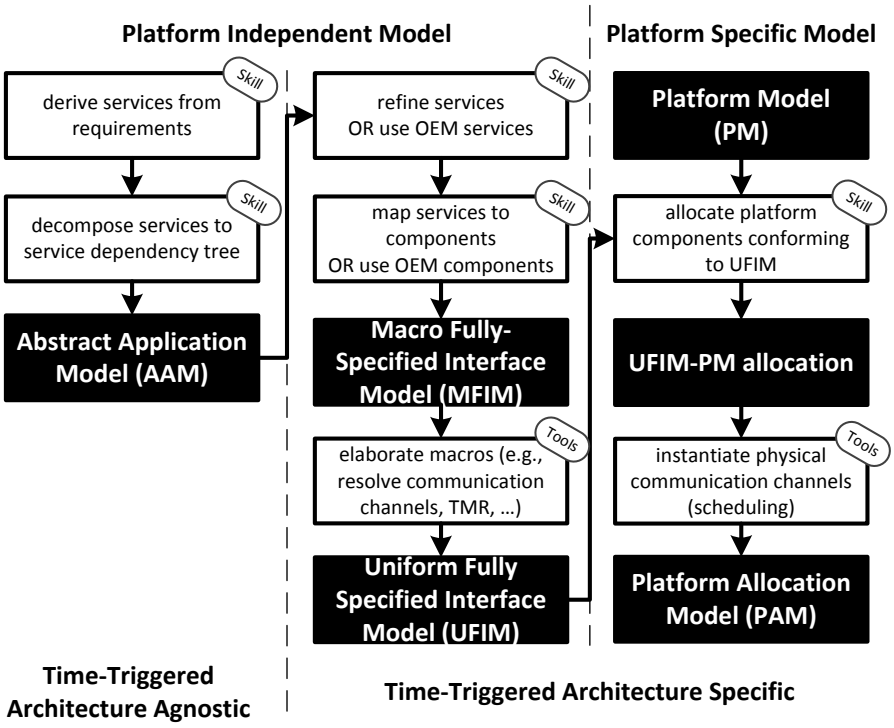


Figure 5.1: The TTSoC Architecture Development Approach Design Flow

Based on the previous TTA development approaches, our proposed recursive components extension in Section 4.1, concepts from the Service oriented Architecture (SoA) (see Section 2.1) and the work of Hutchinson et al. about service oriented software component-based development, we suggest the design flow illustrated in Figure 5.1.

Starting after the *requirements analysis*, the engineer derives (complex) *services* from system requirements by employing his or her skill and experience. The *behavioral description* of a service might allow for a functional decomposition into subservices that the behavioral description can reference. The engineer applies this method until he or she arrives at a *service dependency tree*. This system model is still very sketchy and barely more refined than the requirements we have started with. Hence, we refer to this model as the Abstract Application Model (AAM) which is still agnostic with respect to any TTA. In the next step the engineer either refines the decomposed services towards the TTA (i.e., basically defines a LIF for the service), or the engineer selects and parameterizes a *parameterizeable service* from a supplier library such that the parameterized service is in conformance to its representation in the AAM. Following to the service refinement step, the engineer maps a group of services to single components. According to the TTA recursive component concept, components can be nested. The service dependency tree can guide this component nesting. Again, the engineer might decide to use ready-made *parameterizable components* from a supplier library instead of designing components from scratch. After the mapping, the engineer can formalize the system with the *service-oriented* Macro Fully-specified Interface Model (MFIM). The service-oriented MFIM

is an MFIM (see Section 3.4) whose macros operate on services provided and requested by components, i.e., a model transformation is able to expand those macros automatically to a Unified Fully-specified Interface Model (UFIM). Finally, the engineer can take a Platform Model (PM) and allocate platform components that adhere to the constraints given in the UFIM. The result of this process is the *UFIM-PM allocation*. From there, tools can automatically transform the UFIM, the PM and the UFIM-PM allocation to the Platform Allocation Model (PAM) which represents the system implementation and can be deployed to any platform conforming to the used PM.

In the following subsections we explain our development approach in more detail.

Modelling Distributed Applications as Services

We previously defined a distributed application as an entity that provides a service to the user (cf. Section 3.1). In terms of the SoA, a service is a set of capabilities that might require other services to be able to provide those capabilities. This view rules our suggested development approach. We start at the AAM and derive a set of services from a set of given requirements that should be met by a TTA based application. A service at the AAM is an entity that has the properties:

- **Name:** A unique name shall serve as an identifier of a service. However, the same name shall be given if two services have the same operational modes, behavioral description and non-functional annotations. The case of intended multiple instantiation of the same service can be handled with non-functional annotations and/or to prevent redundancy by referring to the instantiated service.
- **Operational Modes:** A service can offer different behaviors depending on the currently selected operational mode. Services start in a default mode and can change their mode according to their behavioral description.
- **Behavioral Description:** This description can be represented in natural language and contains high-level details how the service intends to fulfill the functional requirement(s) it is supposed to meet. In case the service has multiple operational modes, the behavioral description should also contain the description how mode changes are carried out.
- **Non-Functional Annotations:** Non-functional properties like dependability requirements shall be specified here.

For example, we can derive the airbag service from the requirement to protect the driver of a car in the event of a frontal crash. In a top-down process these usually rather complex services should be decomposed to a behavioral description that uses – and in return depends on – other services (see Figure 5.2, decomposition). This process is similar to object oriented programming and the final result depends on the engineer's skill and experience. In our airbag service example, a not-yet decomposed behavior description could be in natural language: *If a frontal crash is detected, deploy airbag at the correct time instant*. The decomposed behavior description: *If the frontal crash detection service recognized a crash, use the airbag deployment service to deploy the airbag at the correct time-instant*. Typically, non-functional properties of a decomposed

service are inherited to depending services. For example, the airbag service has a high criticality level, hence all its depending services should have at least the same criticality level. The decomposition process naturally supports behavior refinement: For example, the frontal crash detection service description might include constraints about sensor parameters that support the underlying requirements. During the process of decomposition, we build a service dependency tree (i.e., an acyclic directed graph) where the service provider and service requesting relationships become visible. The leaves of this tree must be service providers and in a reactive real-time application, those leaves might be services that directly interact with the environment (i.e., sensors and actuators) or databases where information can be stored and retrieved. The root of the tree is a service that only provides capabilities to the user, hence no other service depends on it. We deem the term 'distributed application' as a suitable synonym for a root in the service dependency tree. During service decomposition it might occur that two or more services require the same services, i.e., their decomposition reveals similarities. Naturally, the engineer should exploit these similarities. In the service dependency tree such a situation shows as subtree duplicates and all but one of these subtree duplicates can be removed. The edges that lead originally to the removed subtree duplicates should be linked to the remaining subtree duplicate (which is now a unique subtree). In terms of the SoA the service dependency tree expresses a structured Enterprise Service Bus (ESB): Every service's need for capabilities is satisfied by services lower in the tree. However, the structure of the tree prevents the interaction of unrelated services.

From the AAM to the Service-Oriented MFIM

Now, we take the service dependency tree as the input for an AAM to *service-oriented MFIM* model transformation. The engineer performs the transformation from the AAM, again relying on his or her skill and experience, because refinement of the information contained in the AAM is highly non-trivial: In the first step (see Figure 5.2, refinement) new parameters need to be derived from the existing service description and formalized per service:

- **Formal Behavioral Specification:** Here, the earlier possibly informal behavioral description should be formalized in an appropriate model of computation. For example an engineer can realize this specification in SCADE Suite, Matlab Simulink or even as a C program, compiler and a model of the executing CPU. The important aspect of a formal behavioral specification is to not only describe algorithms, but also the executing, possibly abstract, machine (see Section 2.2 concerning executable models).

- **Message Ports:** Service communication is message-based, hence input and output message ports must be fully specified with respect to syntax and semantics for each operational mode of the service. The temporal behavior of messages shall be constrained and impose also restrictions on the formalized behavior. If we look at the airbag example from before, for the frontal crash detection service we could define an output message with a periodicity between $2\mathrm{ms}$ to $10\mathrm{ms}$, containing a bit that indicates if a crash has been detected. This constraint also imposes limits on the formal behavior specification: the WCET of the specified behavior must accommodate the required message periodicity/deadlines. A message port has the properties:

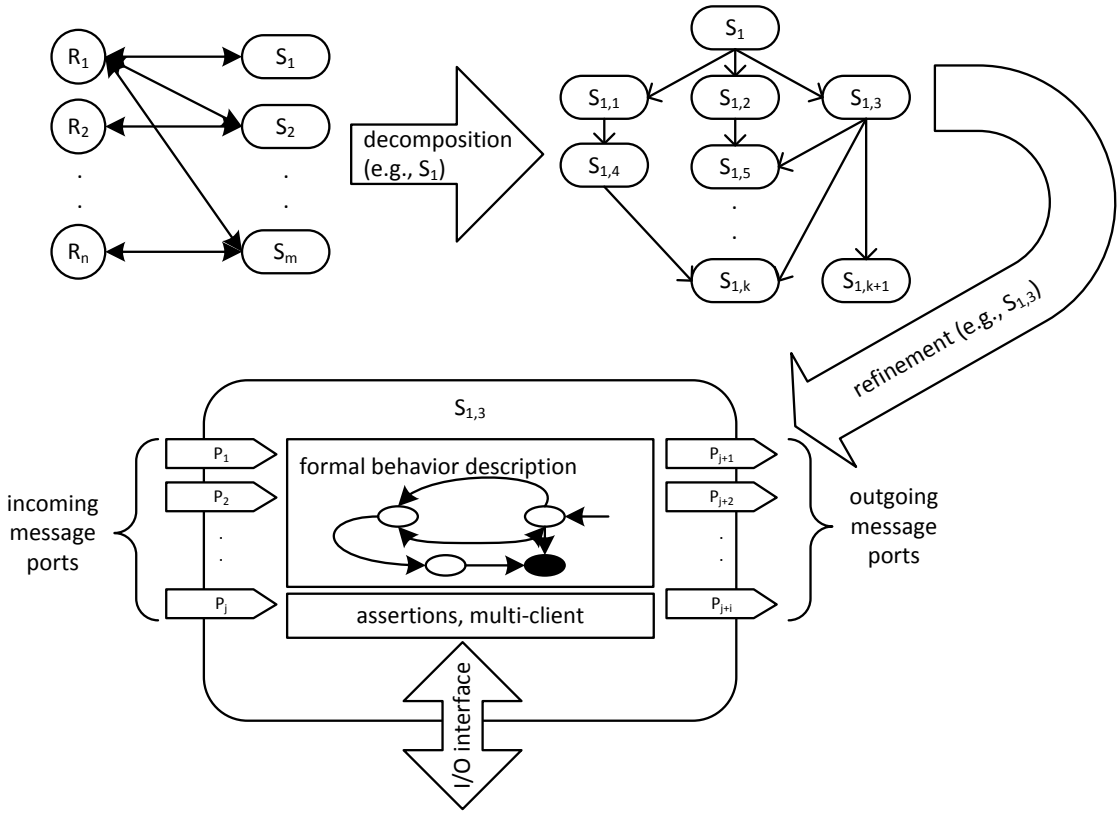


Figure 5.2: From TTA agnostic AAM to a TTA specific Model

- **Unique Name:** Within a single service, the name of the port must be unique.
- **Direction:** A port is either an *incoming port* or an *outgoing port*.
- **Type:** A port either transports state messages, i.e. is a *periodic port* or transports event messages, i.e., is a *sporadic port*.
- **Size:** The message size in bytes.
- **Period:** The period in seconds: e.g., $2^{-4}s$.
- **Phase Constraints:** Sometimes it is necessary to define that messages arrive before other messages in the same period. A list of phase constraints can restrict the phase between a very coarse-grained constraint down to defining absolute-valued phases.
- **Maximum Event Rate:** The event rate in events per second determines the queue size for event ports.
- **Active Modes:** A port is only active during the modes defined here.
- **Assertions:** Value assertions (e.g., range, change rate, ...) can be defined here and are subsequently checked by the DU if the service is monitored.

- **Multi-Client Awareness:** This is a flag that indicates if the service must be available for multiple clients (i.e., service requesters): e.g., a database service that responds to individual queries from different services. In case the flag is set then all outgoing message ports are replicated per client. Otherwise, all outgoing ports are multicast ports where transmitted messages are received by all clients. Incoming ports are replicated per client regardless of the flag (see Section 5.3 for details).

- **Local I/O Interface:** The formal behavioral specification might obtain information from the environment or apply changes to it through the local I/O interface.

- **Assertions:** Assertions related to the behavior of the service should be defined here. These assertions can be used to verify derived implementations or they can be used for error detection during runtime (diagnosis).

In the second step of the AAM model transformation, services are mapped to components. For this last step of the model transformation the engineer has the full power of the recursive component concept that we described for the TTSoC Architecture in Section 4.1. In case a service dependency subtree of size larger than one service is mapped to a single component, the subtree collapses to the single service of the root of the subtree. In this case, we waive the additional information gained through the service decomposition. Concerning our airbag service example, we simply map the three services to three separate components. We give a more involved example in Chapter 6. The information from the service dependency tree and the component-mapping determines the LIF and the non-functional properties of each component. The component LIF consists of the *service providing LIF* and the *service requesting LIF*. In fact, the component interaction relations (i.e., channels) remain indirect in our service-oriented MFIM. The message port description can be seen as a port template that needs to be instantiated during MFIM-to-UFIM transformation. The service requesting LIF is the union over all message ports of the services the component depends on and the service providing LIF is the union over all message ports of the services assigned to and subsequently provided by the component. The component's non-functional properties are the union over all non-functional properties of its services. The component's local I/O interface is determined by the union over all local I/O interfaces of its services and the transformation to our service-oriented MFIM is complete.

From there, tools can expand the macros (see Section 5.5) and transform the MFIM to the UFIM. The UFIM is not a service-oriented model anymore: There are only components interconnected by a set of UFIM channels. Both, components and UFIM channels may still have non-functional annotations (e.g., reliability related), but some of those (e.g., TMR, encryption, ...) are already resolved during the MFIM-to-UFIM transformation.

From the FIM to the PAM

The FIM-to-PAM transformation is the last refinement step from the PIM to the PSM. For this transformation the engineer needs to allocate for each component of the FIM a platform component that is available in the PM which is called the UFIM-PM allocation. Of course, the platform component needs to satisfy every constraint of the originating component in the FIM.

The PM characterizes the available hardware of the entire TTSoC Architecture based system. Specifically, this includes the structure of each TTSoC chip, the gateways, the available architectural plugins, the local I/O interface of each component, and a set of parameters that defines each component with respect to its implementation (e.g., configurable FPGA logic, application computer that can host virtual components, ...).

Having the UFIM, the PM and the UFIM-PM allocation, tools can take over and complete the transformation to the PAM. For example, in this final step of the transformation, the tool instantiates for each UFIM channel an appropriate physical communication channel: e.g., a TTNoC Interconnect or a TTE channel. Those physical channels must fulfill every spatial requirement and temporal constraint. In TTSoC Chips where no dynamic reconfiguration is available, all temporal constraints need to be resolved such that the channel definition is fully-specified with respect to semantics, syntax and time (off-line scheduling). In case of the TTNoC Interconnect this includes routing information. If dynamic reconfiguration with online scheduling is available, the channel can remain constrained with respect to time, sender, and receivers only.

Validation of Distributed Applications

Our approach supports the validation of distributed applications right from the beginning of their development. Validation can be employed at every stage of the design flow and even after application deployment, i.e, during application runtime. Starting at the Abstract Application Model (AAM) which is agnostic to the TTA, the engineer can trace every service back to the set of requirements the service originates from. As the design of the distributed application becomes more and more refined in the TTA specific PIM the engineer is already in the position to formally verify the behavior of each service in isolation. Their interaction can be validated in a generic TTA model of computation (e.g., realizable in Lee's Ptolemy framework [12]). At the UFIM, there exists a TTSoC Architecture System C implementation [18] which models the TTNoC Interconnect semantics. The System C implementation is useful for simulation and model checking. As soon as development has advanced to the PSM, the DU can verify the distributed application at various levels of detail. At the highest detail, the DU can validate the distributed application on a per-service basis against the high-level behavior models defined in the PIM. At a lower level of detail, the DU can verify services against service assertions (e.g., communication patterns, expected responses, ...). At the lowest level of detail the DU verifies messages on a per-channel basis against previously defined service port assertions. Hence, the DU is of tremendous help concerning validating the PSM against the PIM during development. Later, the DU's responsibility is to continuously verify monitored applications.

Suppliers and Legacy Components

We described a process starting at some AAM to obtain logical components described in our service-oriented MFIM and a series of transformations that end at platform components represented in the PAM. Those components are created from scratch and are exactly tailored to fulfill a specific set of requirements by its services. However, our approach does not preclude the use of COTS components – in fact our solution offers very fine grained reuse capabilities at different abstraction levels. Starting at a TTA specific PIM, parameterized services that can be mapped to

components result in parameterized components in the MFIM. Hence, suppliers such as OEMs can build IP at the level of services in the service-oriented MFIM or whole components down at the PAM and offer that in a supplier library. While services in the service-oriented MFIM offer more design flexibility for the engineer, physical components offer the supplier more benefits with respect to quantity manufacturing, IP protection and responsibility delegation in the event of failures. A physical component still retains some level of flexibility, because it can host a wide range of services that are only active depending on the component's instantiation parameters and defined service operational modes.

Suppliers that only want to offer physical components are not even required to change their development process. The documentation of the supplied component only needs to include sufficient information to create a faithful representation in the MFIM. In some cases, mostly concerning legacy components, a direct integration might still not be possible because of a property mismatch (e.g., a component reports the temperature in degrees Fahrenheit while the remaining system expects degrees Kelvin). In this case, a *wrapper component* needs to be designed around the legacy component. Naturally, this wrapper component might contain multiple legacy components. Important is that the gateway component that connects the wrapper component with the remaining system handles any possible property mismatch, i.e., the gateway component acts as a translator between legacy/third party system and the remaining TTA based system. We express a property mismatch as a non-functional annotation of a legacy component such that the gateway instantiation can be automatized during the MFIM-to-UFIM model transformation.

5.3 The service-oriented MFIM

In this section we present an early prototype implementation of the service-oriented MFIM and describe the available macros.

Meta Model

Figure 5.3 shows the *MFIM* meta-model in Eclipse Ecore, an implementation of OMG's Essential Meta-Object Facility (EMOF). This model describes the modeling elements available in MFIMs. We do not explicitly model the local I/O interfaces, but collect these properties along with system, component, service, and port parameters in `CfgParameter`. For convenience reasons, we also added the attribute `id` to some classes. Finally, the operational mode of a component does only indirectly determine the active services. In our meta model, only single ports are active or inactive during a specific set of modes. This design choice reflects the fact that changed service behavior only becomes visible through a change in the component's message-based interaction with other components.

The rest of the meta model closely resembles the described service oriented MFIM in Section 5.2. It remains to explain the available macros and how to expand them such that we can transform the MFIM to the UFIM.

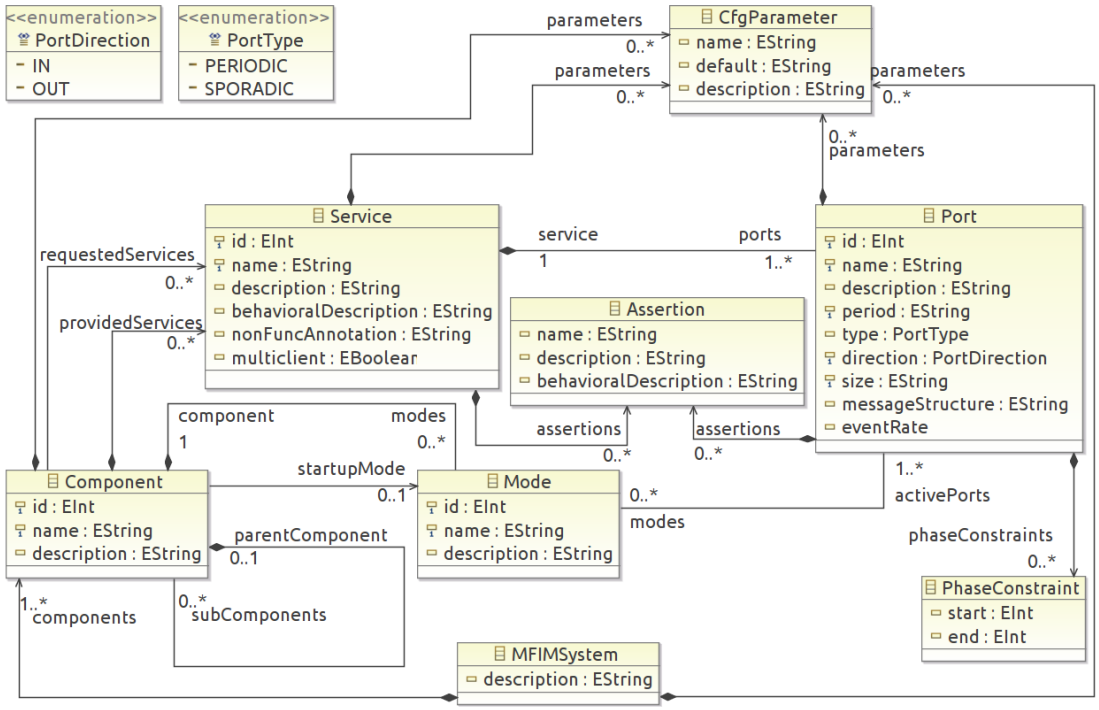
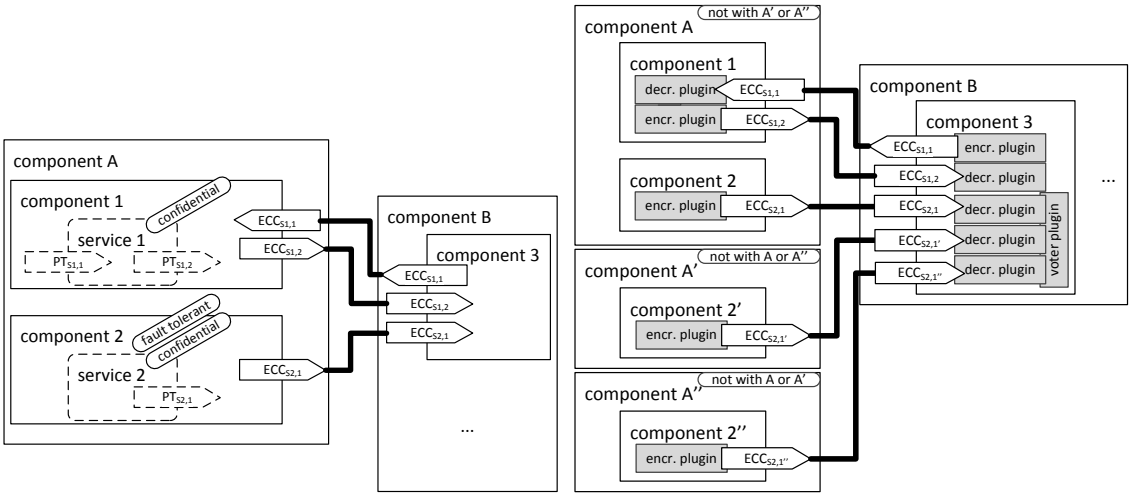


Figure 5.3: The service-oriented MFIM Meta-Model

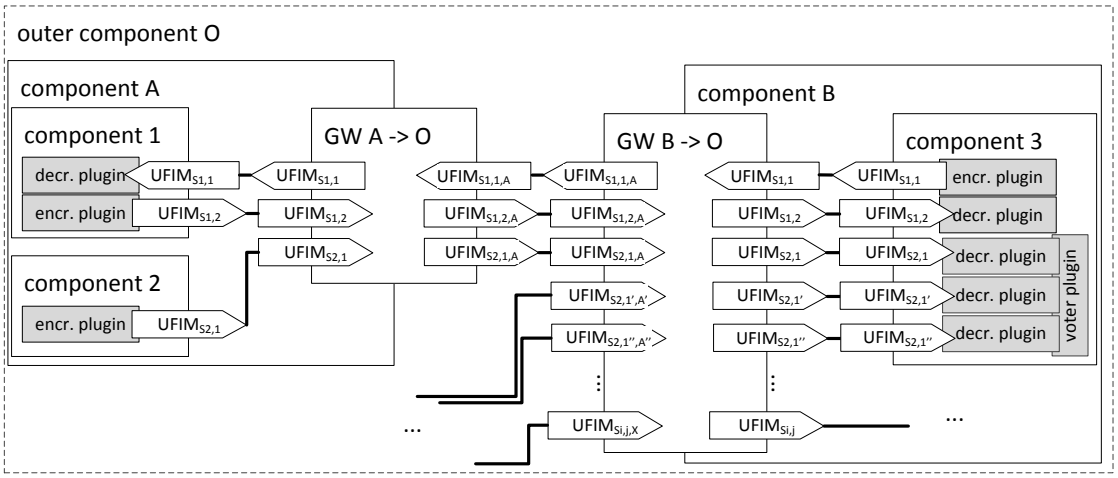
MFIM-to-UFIM Transformation

This transformation consists of multiple steps. First, all *interim ECCs* are assembled by evaluating the information of the service dependency tree, the message port template and the multiclient flag of each service mapped to a component. An interim ECC is an ECC that might not be final yet because other macros in the MFIM could still transform them further. In case there are services with multiple operational modes, the transformation also instantiates RMAs wherever required (at most one per component hierarchy level). Now, the model transformation takes care about the evaluation of most non-functional properties, specifically those non-functional properties which concern the instantiation of (platform independent) architectural plugins. This first concerns the elaboration of confidentiality related non-functional properties. For each affected channel, the transformation instantiates an encryption plugin at the sending endpoint and decryption plugins at the receiving endpoints. Each security plugin instantiation is augmented with minimum encryption requirements that the platform later needs to meet. Other macros (e.g., adding redundancy data, DU independent assertion checks, ...) related to meet component requirements should be taken care of at this point of the transformation. Like the RMA, the DU is instantiated wherever required and also at most once per component hierarchy level. The next step in the model transformation is the evaluation of non-functional reliability properties to increase the reliability of components by TMR (e.g., an ultra-high reliability requirement). This macro replicates these components two more times, instantiates voter plugins at each re-

ceiving channel endpoint and applies necessary changes to the *interim ECCs*. The voter plugin is chained after all other plugins (e.g., the decryption plugin). Last, the transformation instantiates gateway components and resolves the remaining (and now final) ECCs to one or more UFIM channels concatenated by gateways. However, the information regarding ECCs is not lost during the resolving process, but incorporated as *translation configuration* (e.g., information regarding UFIM port mapping, property translation, ...) in the instantiated gateway component.



(a) Service Port Template Expansion to interim ECCs (b) Expansion of Non-Functional Annotations



(c) Transformation to UFIM

Figure 5.4: Service-Oriented MFIM-to-UFIM Transformation

Service Port Templates

In the MFIM, services are convenient means to group related message ports within a component. A port template also includes the direction of the port relative to the providing component. For example, an incoming port is a receiving port at the service providing component, but a sending port at the service requesting component. Depending on whether the service should support multiclients, the macro elaboration, which results in the set of interim ECCs, behaves differently:

- **Multicast Services:** Here, the model transformation instantiates for each service requesting component and for each incoming port an ECC with the service requesting component as the sender and the service providing component as the single receiver. For each outgoing port the transformation instantiates a multicast ECC with one sender which is the service providing component and multiple receivers which are the service requesting components.

- **Multiclient Services:** Like before, the transformation also instantiates for each incoming port and for each service requesting component a single unicast ECC. In contrast to multicast services, outgoing ports are resolved differently. Now, for each outgoing port and each service requesting component a unicast ECC with the service providing component as the sender and the respective service requesting components as the receiver.

For multiclient services the model transformation replicates the port template for each service requesting component. Hence, each of the service requesting components can be identified and served in isolation (i.e., in a session) of all other components. Multiclient services establish arbitrarily complex bidirectional channels between components. For example, a storage component can offer a transaction based shared storage service for its service requesting components. Multicast services on the other hand allow for efficient information push to multiple receivers that are still allowed to give feedback (e.g., a force-feedback gas pedal service).

The end result of the service port template macro expansion is a set of interim ECCs. This part of the transformation does not refine/modify any time constraints: each ECC inherits its time constraints from the port template. Hence, services with ports in the port template that are fully-specified w.r.t to time cannot be scheduled: there would be more than one message port with exactly the same PDS at a single component. Figure 5.4a demonstrates in a simple example how we carry out the service port template macro expansion (see the dashed service port template description from which we create the three ECCs).

In case a service S defines more than one operational mode, the service port template macro expansion also instantiates an RMA at the same component hierarchy level of the component that provides S and an RMA for each component that requests S in the UFIM. In total the transformation instantiates only a single RMA per component hierarchy level. The RMA takes care about possible reconfigurations initiated by a service. Hence, the transformation creates an ECC from the service S providing component to all affected RMAs to request mode changes. Also there is a single ECC multicast channel to each component that is involved with service S to announce mode changes. Naturally, the reconfiguration instant must be synchronized among all affected components.

Service and Port Assertion Checking

For validation or for runtime verification the engineer can define assertions on services and ports to be monitored and checked by a DU. In case there is such an assertion defined, the transformation instantiates a DU component in the UFIM. The behavior of this new component is the union over all behavioral descriptions defined in the single assertions. The transformation adds the DU as a receiver of all interim ECCs that are part of a service or port assertion. Thus, all affected ECCs end up as multicast channels.

Resolving Non-Functional Confidentiality Annotations

The transformation instantiates an encryption plugin at the sender and a decryption plugin at each receiver port of all ECCs that require confidentiality. Non-functional annotations concerning confidentiality are inherited by ECCs that an expansion of a specific port template of a service created. Additionally, the message size of the involved ECCs might increase and/or additional ECCs are created for key exchange. It is not part of the transformation to actually choose an adequate cryptographic algorithm or even define keys, but only to take care about possibly additional communication requirements. The FIM-to-PAM transformation is responsible to select adequate plugin implementations conforming to the available message sizes, ECCs and confidentiality annotations. Figures 5.4a and 5.4b demonstrate how we carry out this transformation (see component 1 which has a non-functional annotation concerning confidentiality).

Resolving Non-Functional Reliability Annotations by TMR

For every component which needs TMR the transformation instantiates two replica components. In case of off-component (e.g., off-chip) TMR, the two replica components are instantiated in another component that gets automatically annotated to be sufficiently independent of the other two replicas. For example, such an annotation would prevent the deployment of these components as virtual components on the same host component during the FIM-to-PAM transformation. The macro expansion also affects ECCs where the replicated component participates:

- **As a Sender:** For every sending interim ECC port two more ECCs are instantiated, whereas each respective replica component is the sender, and the receivers remain the original receivers of the interim ECC. The chain of plugins attached to the replicated component's sending ECC port is also instantiated at the replicas. During the macro expansion, we instantiate the chain of plugins attached to the receiving interim ECC port of the replicated component also for the receiving ports of the two new ECCs. It follows that three different ECC ports end up at each receiver. These three ports form the input of a voter plugin that the model transformation instantiates at the end of the plugin chain.

- **As a Receiver:** For every receiving interim ECC port the macro expansion adds the two replicas as receivers to the interim ECC. In case the ECC previously was a unicast ECC it is now a multicast ECC which includes the replicated component and its replicas as receivers.

Figures 5.4a and 5.4b demonstrates how we carry this transformation out (see component 2 which has fault tolerance and confidentiality non-functional annotations).

Gateways

Finally, there is the problem of resolving ECCs to UFIM channels. A UFIM channel is a unidirectional channel at the lowest abstraction level. Such a channel can be directly mapped to a physical TTA-conforming communication platform element (e.g., to the TTNoC Interconnect) and no gateway components are required in between. In the following we give an inductive description about how we instantiate gateways and UFIM channels from an arbitrarily given component structure and arbitrary ECC channels among those components. The transformation is applied repeatedly until all ECCs are expanded to gateways and UFIM channels.

- A single component does not need a gateway to interact with other components at the same component hierarchy level. Hence, every set of components within an *outer component*, can communicate directly. This means the transformation instantiates for each ECC between such components an UFIM with the same properties (same sender, set of receivers, time constraints, ...).

- An outer component O where a set of components C within O have a set of ECCs E between other components that are not contained in O requires the instantiation of a gateway within O. Any channel from E needs to pass through the gateways in O as follows:

  - If the sender is contained in C, the model transformation instantiates an UFIM channel originating at the component in C. The set of receivers of the newly instantiated UFIM channel contains all receivers from the originating ECC that are also in C plus the gateway. Then the model transformation instantiates an ECC channel that originates at the gateway and includes all the receivers of the originating ECC without those receivers that are part of C.
  - In case receivers of a channel of E are contained in C, then the transformation instantiates an UFIM channel from the gateway to the receivers and instantiates an ECC between the sender of the original ECC (behind the gateway) and the gateway plus all other receivers of the original ECC that are not part of C.

  In both cases, the original ECC is replaced by a combination of UFIM and ECC channels. Of course, ECC channels need to be transformed to UFIM channels at a later stage in the overall transformation. Also, the transformation can conveniently keep track of gateway semantics, i.e., port lookup configurations (how UFIM channels from the inside relate to UFIM channels from the outside), possible message translation to ensure interoperability among third party supplied components (originating from property mismatch non-functional annotations), ...

Figures 5.4b and 5.4c demonstrate how we apply this transformation on a small example. Note that there is always an outer component O even if we do not explicitly design it (cf. Figure 5.4b). According to the recursive component concept (see Section 4.1), we can always reduce the

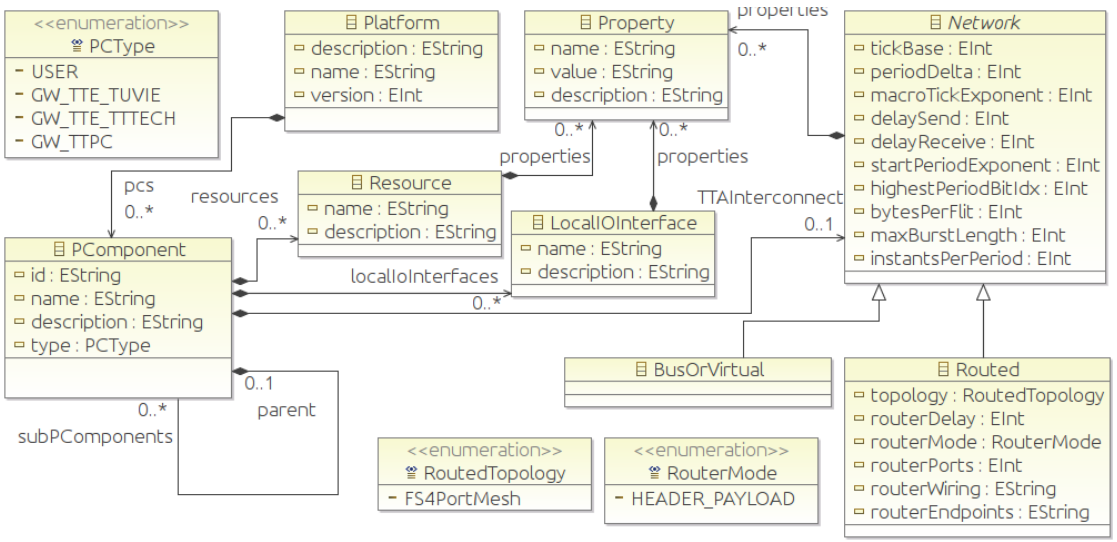


Figure 5.5: The Platform Meta-Model

whole system to a single component. We also present in the figures a simple naming scheme for channels to ensure traceability back to services and unique channel names that do not require a centralized naming authority. The first part of the channel name is always the service name, followed directly by the port number of the service. Whenever a channel leaves a service providing component (channel direction is irrelevant) the component name is attached to the channel name. For example, the ECC $\mathrm{ECC}_{\mathrm{S2},1}$ in Figure 5.4a refers to the first port of the service S2. At the transformation to the UFIM (cf. Figure 5.4c) this ECC is resolved to three UFIM channels whereas two of the three UFIM channels retain their name. Only the UFIM channel originating at the instantiated gateway of component A gets the name of component A attached, i.e., the UFIM channel $\mathrm{UFIM}_{\mathrm{S2},1,A}$.

5.4 A Simple Time-Triggered Architecture (TTA) PAM

This section gives a simple example of a PAM suitable for TTA compliant distributed applications. Focus is on the TTSoC Architecture and TTE based systems. However, this PAM can be easily extended to any other TTA compliant architecture.

Platform Meta Model

In Figure 5.5 we give a TTA *platform meta-model*. This meta model contains elements to describe TTA systems. It supports the recursive component concept, gateways, components with PDS-based interconnects and components with local I/O interfaces. Each of the architectural elements can be augmented with properties. Specifically, the meta-model allows to describe TTSoC chips with TTNoC Interconnects and components that possibly host virtual components. Other architectural elements like the RMA or DU are not explicitly visible in the platform model, be-

cause they are just components. Currently, the meta model only supports some selected gateway component types and one specific routed topology. Architectural plugins are represented by the class `Resource` only. However, this meta model only serves as an example that already covers the TTSoC Architecture implementation and TTE based systems.

Allocation

Now we have on the one hand the UFIM and on the other hand the PM conforming to our PM meta model. It remains to allocate the UFIM components on platform components, i.e., the UFIM-PM allocation. This allocation is a simple mapping where the platform independent structures are instantiated on the platform. Of course, this is also a refinement model transformation where information is added to the design. However, we prefer to express the relation between the UFIM and the PAM as links or mappings, because the platform is usually given (e.g., a mass produced general purpose TTSoC chip with a set of standard components and a set of standard TTNoC Interconnects) and allows many possible allocation choices. The allocation process itself only needs to respect all the constraints implied by the PIM.

Scheduling and Dynamic Resource Management

In contrast to the PIM, in the PAM all component interfaces need to be fully-specified, also with respect to time. In a strict interpretation this means, we need to statically schedule each of the UFIM channels for the respective time-triggered interconnect such that the precise time instants of transported messages are fixed. However, such a static schedule would severely limit dynamic resource management. Hence, conceptually, we allow the TTA to modify its PAM during runtime. These modifications are limited to the constraints specified in the FIM. Only the RMA can apply such modifications at previously defined reconfiguration instants.

5.5 Tool Support

As a proof of concept, we developed a tool that operates on models conforming to a subset of the meta-models and transformations presented in this thesis. Specifically, the tool supports only a two level-hierarchy of nested components and no architectural plugins are supported. However, we could already employ our tool in our case study in Chapter 6 to tremendously simplify the development process. The tool (see Figure 5.6) takes the MFIM, the PM and the UFIM-PM allocation as inputs and produces the UFIM and platform configuration data. Naturally, the platform configuration data is very platform specific. We assume that TTSoC chips mostly contain general purpose CPU based components that are capable of executing programs written in C. Hence, our tool produces the configuration data as C source code. Additionally, our C-based VCRE (see Section 4.2) is able to directly make use of the produced configuration data.

Our tool processes the input models exactly as described in the previous sections. The implementation language is Java and besides the model transformations we also included code

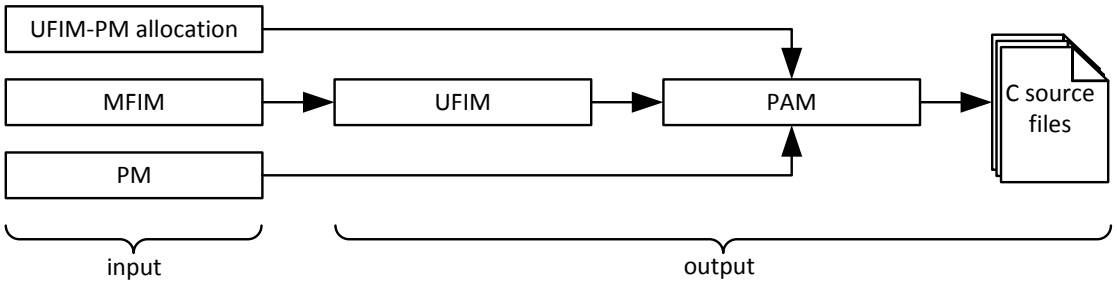


Figure 5.6: The Model Transformation Tool

for scheduling PDSes according to our description in Section 4.3. Table 5.1 gives a brief lines of source code overview that we obtained by employing the tool cloc[2].

Files	Comments	Code
72	2 030	6 955

Table 5.1: Source Code Overview

The generated C configuration output is a set of C source and header files that include:

- **Architectural Component Configuration:** Some architectural components (e.g., RMA, DU, gateways) need to utilize information about general component parameters regarding the TTNoC Interconnect, gatewayed channels and monitored channels or services. The tool produces this information in the C module `noccfg`, `ttegw_cfg`, `ttegw_mapping` and `servicecfg`. Further, for the RMAs the tool creates a PDS repository. RMAs need to know about all possible PDSes that might need to be scheduled on the respective TTNoC Interconnect. Our tool produces this information per component that contains an RMA in the C module `pdscfg`.
- **User Component Configuration:** The behavioral description of components operates on services. Hence, the tool structures the message ports according to the service they belong to in the C module `servicecfg`. The tool also generates a C module per component called `portcfg` where the individual port configuration is contained. This port configuration includes the port type (state or event), maximum message rate, size, direction and a bitmask to indicate at which operational modes the port is active.

5.6 Validation of Requirements

All requirements that we defined in Section 5.1 are met. We briefly outline the mechanics of our development approach that are responsible for fulfilling the stated requirements in Table 5.2.

[2]http://cloc.sourceforge.net

Requirement	Rationale/Notes	Result
R0	The development approach supports the integration of multiple distributed applications. See R3 for additional rationale/notes.	✓
R1	Right after the transformation from the AAM to a PIM that is TTA specific, services must be augmented with information to describe the LIF of the component they are mapped to. The sum of the service behavioral descriptions that end up in a single component must adhere to this component LIF. Hence, the development approach supports the creation of time-triggered applications.	✓
R2	The development approach supports the usage of tools to derive UFIM channels and instantiate them in the PAM. Tools can schedule these instantiated UFIM channels enabling their direct deployment to TTSoC chips as TTNoC Interconnect configuration. UFIM channels can also remain unscheduled to allow for dynamic runtime reconfiguration scheduled by the RMA.	✓
R3	Suppliers can deliver their IP in form of parameterizable services or components at the MFIM level to allow the engineer more design flexibility or they can deliver their IP as parameterizable physical COTS components. Even legacy components can be integrated by means of wrapper components that ensure interoperability with other components.	✓
R4	There are macros in the MFIM available that implement non-functional requirements with architectural means. Tools automatically expand these macros in the MFIM-to-UFIM transformation and instantiate available architectural plugins as required.	✓
R5	Arbitrary assertions can be defined on the level of services and single message ports. The DU can check these assertions. Hence the development approach supports the development of applications that can be checked during runtime.	✓
R6	Services can define operational modes. According to the currently active mode they can exhibit different behavior and active/inactive message ports. Thus, the development approach supports the development of applications that can adjust their resource requirements.	✓

Table 5.2: Validation of Requirements

5.7 Discussion

We presented our development approach which takes advantage of our proposed improvements concerning the TTSoC Architecture in the previous chapter.

In our approach we introduced the concept of service as the central unit of design in the AAM. We regard a service as a container for a (partial) component LIF, functionality and non-functional properties. In principle we could already introduce the recursive component concept in the AAM, but we deem the concept of services more appropriate: Firstly, at the AAM we are faced with a set of application requirements and should not concern ourselves with system structure details yet. Secondly, with services the engineer is already able to express all fundamental interaction relations and non-functional dependency relations among services by building the service dependency tree during decomposition. Thirdly, we believe a functional decomposition of services similar to the object oriented programming paradigm gives the engineer better conceptual support than dealing with the decomposition of components directly. Finally, the concept of services remains useful even after the design flow of our development approach enters TTA specific models where we have the concept of components available. For example, at the MFIM where components are the central unit of design we still do not need to deal with any channels: all interaction relations are expressed by each component's set of requested services and provided services. The concept of channels is first introduced in the UFIM.

Naming in distributed systems is a very important aspect that we did only barely handle in this thesis. However, the naming scheme proposed by Salloum in his dissertation [13] still applies. Our approach does not require the engineer to deal with naming issues directly which is also because of to the service-oriented MFIM. Only components and services need to be addressable and for both elements we required unique names. Channel and component names at the UFIM are all based on these names and tools can take care of automatically establishing a naming scheme of all system descriptions below the MFIM.

We also need to point out that the development approach is still slightly ahead of existing implementations of the TTSoC Architecture. For example, there remains much future work to implement feasible RMAs and DUs (incl. their cooperation) as well as gateways that can incorporate the derived behavior from the MFIM-to-UFIM transformation. Notwithstanding, we believe we solved most of the problems regarding application development for the TTA and specifically for the TTSoC Architecture.

CHAPTER 6

Case Study: Mixed Criticality

In the following case study, we show the viability of our development approach. The case-study itself is based on a demonstrator that we implemented for the projects TT-SoC and GENESYS. Originally the purpose of the demonstrator was to carry out integration tests of all the developed architectural elements and show some of the benefits of the TTSoC Architecture. However, in this case study we focus on previously unpublished results with respect to our work described in Chapters 4 and 5. Specifically, we want to exercise the development approach and employ our advancements concerning the TTSoC Architecture.

The topic of our case study is mixed criticality. We develop a safety critical automotive application and a low critical multimedia application with several operational modes executing on the same platform. First, we design services to meet a given set of requirements at an abstraction level much higher than the TTSoC Architecture or even distributed systems. Step by step we follow our development approach design flow, i.e., we decompose services, then integrate them to components and finally allocate these components on our demonstration platform. At the end of this chapter, we deploy the allocated components and validate the results.

6.1 Overview and Requirements

Vehicles have numerous separately operating systems called Embedded Control Units (ECUs) that are responsible for various services: e.g., engine control, braking system, power steering. In recent years the number of these services increased, hence we currently find more than 100 ECUs (Audi A8 [16]) in a modern car. In this case study we concentrate only on four such services, namely an Anti-Lock Braking System (ABS), configurable steering that also restricts steering during highs speeds, an electronic gas pedal service and a multimedia display system. However, starting at the beginning we give for illustrative purposes a set of (toy) requirements:

R1 The vehicle shall remain steerable during (emergency) braking.

R2 The driver shall be able to steer the vehicle according to a configurable steering translation.

R3 During high speeds the driver shall be able to apply at most 1/3 of the possible car steering angle.

R4 Car acceleration shall respond to an electronic gas pedal.

R5 Requirements R1, R2, R3 and R4 shall be certified up to safety-criticality levels.

R6 A video playback system shall present a video streamed to two LCD screens. Multiple operation modes shall be supported: a negative mode that processes the video stream such that its negative is displayed and a normal mode that does not modify the video stream.

R7 All services shall offer an appropriate set of service degradation modes that can be activated in the event of resource shortage before total service shutdown.

6.2 Towards the service-oriented MFIM Representation

From the list of requirements we conclude that the system under development needs to provide four services: an ABS service, a steer-by-wire service, a gas-by-wire service and a multimedia service. For each service we define the properties behavior, operational modes and non-functional annotations:

S1 **ABS Service:** ABS, invented in 1929 by Gabriel Voisin [1], helps in keeping a car steerable during (emergency) braking. The wheels are prevented from locking, thus the vehicle does not start to skid uncontrollably. A typical ABS (for e.g. motor-cycles, cars, airplanes, ...) consists of an ECU, two or more wheel speed sensors (one sensor for each wheel) and two or more actuators responsible for modifying the braking force applied on each wheel. During a braking process, the speed of the wheels is repeatedly checked and if one or more wheels rotate considerably slower than the other wheels or the speed indication from the transmission, locking is prevented by reducing the braking force on those wheels that start to lock.

This service has only one operational mode. Any degradation would invalidate safety margins.

This service is safety-critical.

S2 **Steer-by-Wire Service:** The steer-by-wire service steers the vehicle according to user input from a driving wheel and a customizable steering ratio[2]. This customizable steering ratio determines a simple linear function that controls how driving wheel turning relates to the actual vehicle steering. In case the car drives faster than a configurable limit only one third of the normal steering angle range is available to the driver.

This service has only one operational mode. Any degradation would invalidate safety margins.

[1] http://en.wikipedia.org/w/index.php?title=Anti-lock_braking_system&oldid=583881021

[2] While x-by-wire is very popular in the avionic domain, in the automotive domain costs for x-by-wire are still an issue and even the most modern cars still retain mechanical linkage.

This service is safety-critical.

S3 **Gas-by-Wire Service:** The service controls car acceleration according to user input from a gas pedal.

This service has only one operational mode. Any degradation would invalidate safety margins.

This service is safety-critical.

S4 **Multimedia Service:** This service shows a rotating 3d cube on two screens. There are 2 degradation modes, a normal mode and a negative mode. The first degradation mode divides the video resolution by two and the second degradation mode divides the video resolution by four. The image is in both degradation modes upscaled to the original resolution. The negative mode just applies the binary operation 'not' to each pixel. All available service modes are cycled through. A switch to the next mode occurs every 4 seconds.

This service has the operational modes: normal, negative, degradation mode 1, degradation mode 2

There are no non-functional annotations.

Clearly, S1 is traceable to the requirements R1 and R5, while S2 to R2 and R5, S3 to R4 and R5 and S4 to R6. All services are traceable to R7.

The next step is to decompose the four services and build the service dependency tree. Figure 6.1 shows our decomposition. We consider the service 'engine control' as a black box (e.g., supplied by an OEM) that we do not decompose further. The tree has four roots and we already performed a subtree duplicate removal optimization: it turned out that both the ABS and the steer-by-wire service require the car velocity service. The set of services should be described similarly to the four non-decomposed services we described earlier. The service 'car velocity' provides the velocity of the car measured at different points: at each of the four wheels (our car has four wheels) and directly at the transmission. For the sake of brevity we do not explicitly give a behavioral description of the other decomposed services and assume that the service name is descriptive enough. The set of operational service modes is inherited in the direction of the dependency relation. All non-functional annotations are also inherited in the direction of the dependency relation. We do not introduce any additional operational service modes or non-functional annotations for any of the decomposed services. When checking the service dependency tree, we can easily validate:

- Traceability from each service to at least one requirement remains intact by reversing the edge direction of the dependency tree and enumerate all reachable roots.
- No safety critical service depends on a non-safety critical service.

In our development approach design flow we have arrived at the AAM. Next, we need to refine the services and subsequently let them be provided and requested by components. The refinement process comprises formalizing the behavioral description, defining messages and service ports, classifying the service as multiclient or multicast, and defining the local I/O interface and

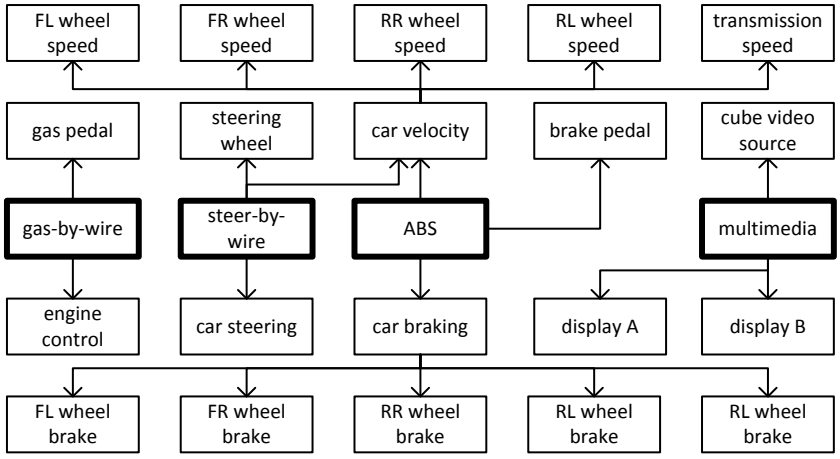


Figure 6.1: The Service Dependency Tree

assertions. We provide the formalized behavioral description as a combination of C source code, compiler and executing processor in our implementation. We give the remaining parameters:

SERVICE: **car velocity**

message ports:
- FL wheel speed (32-bit integer, signed, velocity m/s, period $2^{-6}s$, outgoing, periodic)
- FR wheel speed (same as FL wheel speed port)
- RR wheel speed (same as FL wheel speed port)
- RL wheel speed (same as FL wheel speed port)
- transmission speed (same as FL wheel speed port)

multiclient: no

local I/O if: no

SERVICE: **car braking**

message ports:
- FL wheel braking force (16-bit integer, signed, -2^{15} .. no braking; $2^{15}-1$ .. full braking, period 2^{-6}, incoming, periodic)
- FR wheel braking force (same as FL wheel braking force port)
- RR wheel braking force (same as FL wheel braking force port)
- RL wheel braking force (same as FL wheel braking force port)

multiclient: no

local I/O if: no

SERVICE: **brake pedal** and **gas pedal**

message ports:
- $*$ pedal (16-bit integer, signed, -2^{15} .. fully released; $2^{15}-1$ fully pressed, period 2^{-6}, outgoing, periodic)

multiclient: no

local I/O if: pedal sensor

SERVICE: **steering wheel**

message ports:
- $*$ pedal (16-bit integer, signed, -2^{15} .. fully turned left; $2^{15}-1$ fully turned right, period 2^{-6}, outgoing, periodic)

multiclient: no

local I/O if: wheel sensor

SERVICE: **car steering**

message ports:
- $*$ steering angle (16-bit integer, signed, -2^{15} .. fully turn left; $2^{15}-1$ fully turn right, period 2^{-6}, incoming, periodic)

multiclient: no

local I/O if: car steering actuator

SERVICE: **engine control**

message ports:
- $*$ gas (16-bit integer, signed, -2^{15} .. full acceleration backwards; $2^{15}-1$ full acceleration forwards, period 2^{-6}, incoming, periodic)

multiclient: no

local I/O if: car engine control

SERVICE: **cube video source**

message ports:
- $*$ frame_full (14416 byte frame data for 320x240 24bit RGB, period 2^{-6}, outgoing, sporadic, active in normal and negative modes)
- $*$ frame_half (7216 byte frame data for 160x120 24 bit RGB, period 2^{-6}, outgoing, sporadic, active in first degradation mode)
- $*$ frame_quarter (3616 byte frame data for 80x60 24 bit RGB, period 2^{-6}, outgoing, sporadic, active in second degradation mode)

Additional notes: We structured the video data into single frames. Each frame is represented as a (raw) bitmap, where each pixel is described by its color. The color is encoded in 8 bit sized words and the highest resolution we want to support is 320x240 pixels. We target a maximum Frames per Second (FPS) value of around 11 FPS, so the video data bandwidth we need to sustain totals 901 Kbyte/sec All ports operate in π_8 (64 Hz). The 'frame_full' port transports 14416 byte sized parts

of a whole frame in frame packets. A frame packet is made up of frame data and one header word (4 byte) that describes the position of the packet data in the current frame. The ports related to degradation modes respectively transport smaller sized packets.

multiclient: no

local I/O if: none / cube video defined in behavioral description

For the sake of brevity, we skip some services. The services 'car velocity' and 'braking' depend on services that provide the respective speed sensor values or provide control of the respective wheel brake via their respective local I/O interfaces. The services 'display A' and 'display B' show the received video data on a screen attached to their local I/O interfaces and have otherwise the same properties as the 'cube video source' service (with reversed port directions). The services located at the roots of the dependency tree have no message ports, and also no local I/O interface. At this point we also could add assertions to ports or services. The current implementation of the TTSoC Architecture only offers a partial DU [3], hence we refrain from defining assertions here.

For our purposes the services are sufficiently refined. We proceed in the development approach by mapping the services to components. One method is to start at the leaves of the service decomposition tree and try to place sparse subtrees in their own component. Still, this is a creative process where the engineer needs to use his/her skill, non-functional annotations of services and consideration towards the final platform. In our case, we structure the services into the following components. We also use the component structure to give a few remarks on the actual service implementations.

- C automotive
  - C vehicle sensors and actuators
    - S car velocity, collapsed services: all wheel and transmission speed services
    - S car braking, collapsed services: all wheel brake services
    - S car steering
    - S engine control

    We consider all these services as if they had been supplied by an OEM and were part of the platform.
  - C user I/O
    - S steering wheel
    - S brake pedal
    - S gas-by-wire, collapsed service: gas-pedal

    These services only forward data obtained from the environment.
  - C ABS controller

[3] Currently, the DU is limited to count synchronization errors, and count intermediate- and final-checks (on message checksums) failures.

S ABS
We use a preexisting implementation of ABS employed in one of the robot drivers in a racing car simulation (see Section 6.3).

C steer controller

S steer-by-wire
This service applies a deployment-time configurable constant factor to the steering wheel input that is obtained by the steering wheel service. In case the car velocity service reports a velocity over a given over-speed limit, a limiter ensures that the steering angle is at most a third of the possible range. The final result of the steering angle is deployed to the vehicle via the car steering service.

C multimedia

C cube

S multimedia, collapsed service: cube video source
This service uses a simple graphics library (included in one of Altera's demo designs[4]) to generate the video of a spinning three dimensional cube. There are two virtual framebuffers: one active and one inactive. The active one is repeatedly streamed by using the services 'display a' and 'display b'. We render the cube in accordance with the currently active application mode into the inactive virtual framebuffer. Each time the computation of the frame is finished and after the active framebuffer has been completely sent at least once, the two framebuffers are swapped.

C display a

S display a
The service – depending on the operational mode – scales received frame data to 320x240 and shows the scaled image on a screen attached to the local I/O interface.

C display b

S display b
Similar to display a.

Having refined the services and mapped them to components, we have reached a refinement stage that we can represent formally in our service-oriented MFIM. For our case study we define this MFIM in a set of XML files that we can use directly as input for our tool (see Section 5.5).

6.3 The PAM

The PAM describes the system under development with respect to the hardware/physical platform. Figure 6.2 gives an overview of the platform that we developed for our case study.

We give a short description of each platform component[5]:

[4] http://www.altera.com/support/refdesigns/sys-sol/computing/ref-niosii-vga.html

[5] For more in-depth details we refer to our three deliverables concerning work package 8 of the TT-SoC project.

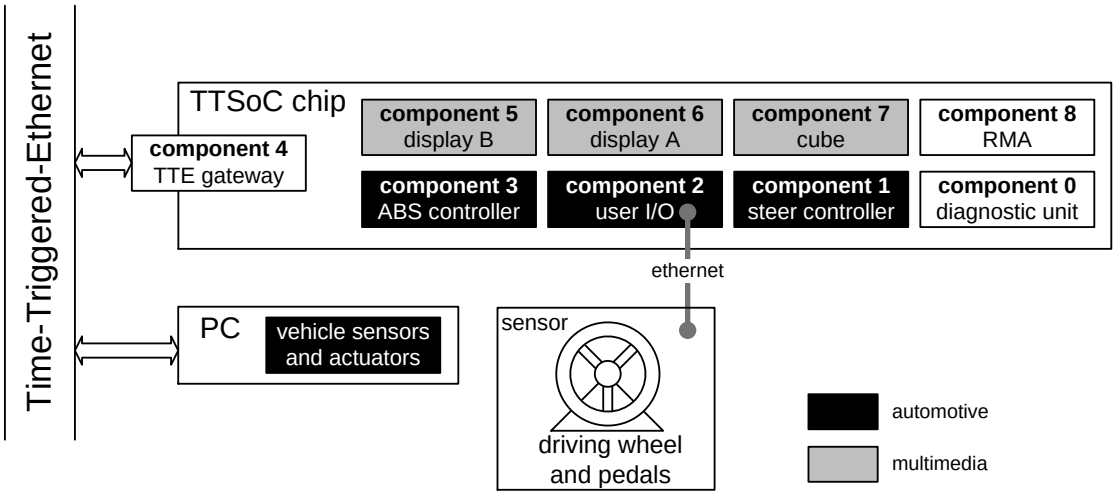


Figure 6.2: System Model

- **TTSoC Chip:** This is a carrier board designed by TTTech[6]. It contains one user configurable Altera EP2C70 FPGA that we use to host the TTNoC Interconnect. The interconnect consists of a 2×3 mesh network with 4-port fragment switches and ten interconnect endpoints. The interconnect parameters for transporting PDSes are: $N = 16$, $\delta = 1$, $\mathcal{M} = 11$, $\mathcal{N} = 32$. The flit size is 32-bit and we use header-payload routing. Further the board contains nine extension slots whereas each extension slot is connected to the TTNoC Interconnect via a serialized UNI. An extension slot can house an extension board. Each of the extension boards we have available contains RAM, flash memory and an Altera EP2C35 FPGA that can be configured with custom logic. In our case study we exclusively configure this FPGA with soft-core CPU systems. These CPU systems realize host components for virtual components. Further the extension board can be equipped with either a multimedia board or an I/O board. Essentially, the multimedia board features a 320x240 LCD screen and the I/O board Ethernet and serial interfaces. All these additional periphery is directly connected to the FPGA and can be used to realize local I/O interfaces.

- **Sensor:** The sensor consists of an USB-driving wheel connected to the single-board computer Soekris net48015[7] that operates under Linux 2.6. The USB-driving wheel is supported by the Linux Input Drivers project[8], so we can sample joystick input data from the input subsystem. Further, we have implemented an UDP IP streamer that opens the joystick input device, parses the joystick events for driving-wheel, pedals and button events and streams them to a configurable target UDP IP address. We use a bootable 512MB compact flash card where all required data and configuration are stored on, hence the Soekris single-board computer starts and operates autonomously after power-up.

[6] http://www.tttech.com
[7] ttp://www.soekris.com
[8] http://atrey.karlin.mff.cuni.cz/ vojtech/input/

- **User I/O:** The user I/O platform component consists of a LEON3 user host equipped with an I/O extension board. The host's application computer runs the Snapgear Linux 2.6 distribution. We implemented an Amba AHB-to-TTNoC Interconnect hardware bridge in VHDL (contains the dual clocked port memory and handles the UNI) and wrote a TTNoC Interconnect Linux driver for it. Our Linux user-space implementation of the VCRE uses the driver and is able to host virtual components. The single Ethernet interface of the host is set up to receive IP packets from the sensor (driving wheel). We have implemented an UDP IP receiver (counterpart of the sensor's streamer) that maintains a shared memory with the joystick state data. We regard the shared memory as Local I/O interface.

- **ABS Controller:** This platform component consist of a Nios II user host and runs our TTOSVC to provide the VCRE and subsequently host virtual components. There is no local I/O interface.

- **Steer Controller:** Similar to the ABS controller platform component, the steer controller consists of a Nios II CPU system and runs the TTOSVC. There is no local I/O interface.

- **RMA:** This platform component realizes an architectural element of the TTSoC Architecture. We used a Nios II CPU stem that runs our TTOSVC. We developed a system task that employs our topology invariant scheduling solution from Section 4.3 and makes use of configuration data (e.g., PDS repository) that we generate with our model transformation tool (cf. Section 5.5). The RMA system task also processes operational mode change requests and carries out reconfigurations of the TTNoC Interconnect.

- **TTE gateway:** This platform component is an architectural element of the TTSoC Architecture. We used a Nios II CPU system and extended it with a preexisting TTE IP core. We implemented a TTE driver for our TTOSVC and a gateway system task. The gateway system task takes care of setting up ports on either side of the gateway (TTNoC Interconnect or TTE) and forwards messages between them. To this end, the gateway task uses configuration data that we generate by our model transformation tool (cf. Section 5.5).

- **DU:** We implemented a rudimentary DU, again as a Nios II CPU system running our TTOSVC. We implemented a DU system task that collects diagnosis disseminations of the TTNoC Interconnect concerning synchronization protocol failures (i.e., in case a component does not follow the TTNoC Interconnect interaction protocol) and message checksum failures.

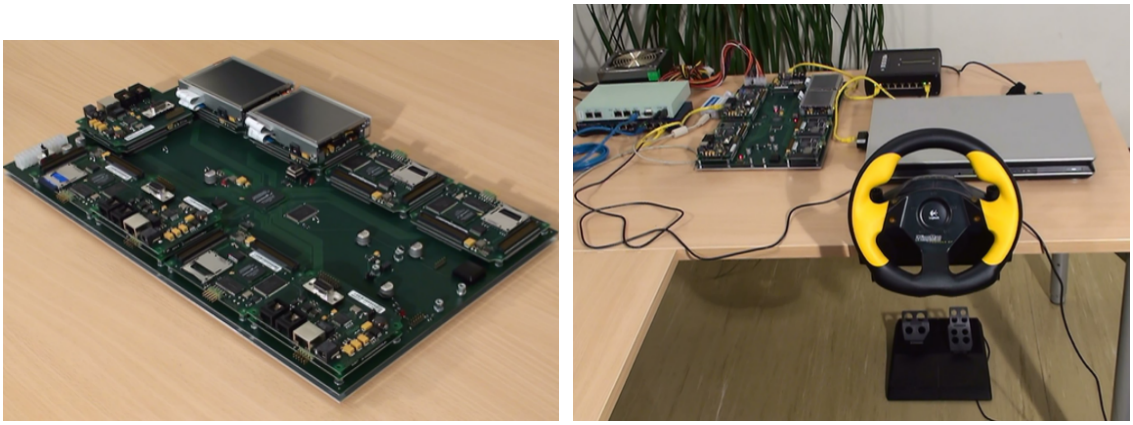
- **PC and Vehicle Sensors and Actuators:** We have set up the racing car simulation TORCS[9] on a notebook with Debian 2.6, RTAI extensions[10] and a TU Vienna developed TTE-PCMCIA [47] network interface. We implemented an RTAI application kernel module which serves as a gateway between simulation and the off-component TTE network. It writes actuator set values received via TTE into a structured shared memory, respectively reads sensor data from the shared memory and sends it over TTE. During this operation

[9]TORCS: The Open Racing Car Simulator, http://torcs.sf.net/
[10]RTAI: Real-Time Application Interface: http://www.rtai.org/

all values are appropriately scaled, i.e., we take care about property mismatch. TORCS offers interfaces to extend the racing simulation with robot drivers. We have implemented such a robot driver that communicates with the RTAI application kernel module by means of the structured shared memory. For practical reasons, the robot driver also implements auto-transmission.

- **Cube:** This platform component consists of a Nios II CPU system that also runs our TTOSVC to host virtual components.
- **Display A and Display B:** The display platform components consist each of a LEON3 CPU system and make use of the multimedia extension board (320x240 color LCD screens) and a Amba AHB LCD controller hardware that we designed in VHDL. gain, we use Snapgear Linux 2.6 and our TTNoC Interconnect driver. For accessing the LCD controller hardware, we implemented a simple framebuffer device driver. The VCRE has access to the framebuffer device and provides it as a local I/O interface to its virtual components.

Figure 6.2 also shows the allocation from MFIM components to platform components. We described this platform and the allocation also in a set of XML files.
Figure 6.3 shows the hardware setup of the patform.



(a) A TTSoC Prototype ”Chip” (b) Platform

Figure 6.3: Hardware Setup

6.4 Compilation and Deployment

We used our model transformation tool (see Section 5.5) on the MFIM, the PM and the allocation description to produce the platform configuration data. Together with the C implementation of each of the virtual components and various required libraries we compiled ELF binaries for all host platform components of the TTSoC Chip platform. To this end, we employed a Unix C-make based build system. The deployment process itself involved starting the TORCS racing

car simulation on the PC and the upload of each of this nine binaries to the respective host platform component via the debug interface (i.e., via Altera USB-Blaster JTAG). After resetting the trusted subsytem (which is also in control of the resets of all attached components), the distributed system and all its developed services became life.

6.5 Validation and Results

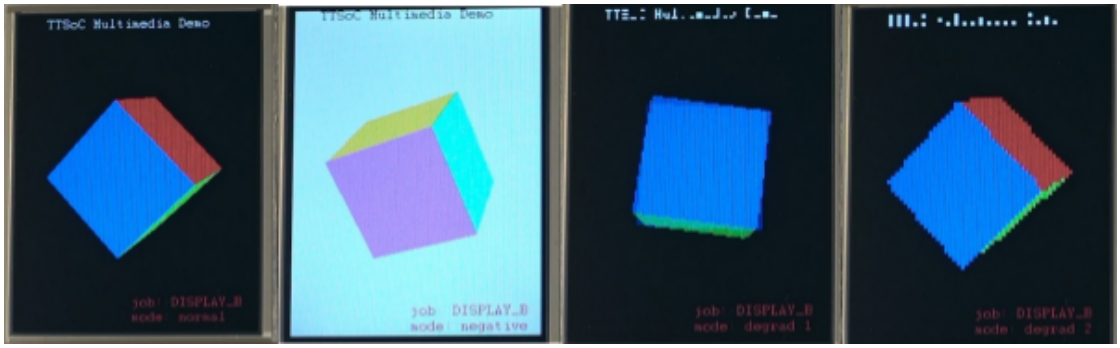
The DU reported no errors during runtime which means that all (virtual) components sent and received messages correctly. We used the tool Wireshark[11] with a TTE plugin to validate the message transport on the TTE network and found no discrepancies to the generated platform configuration data. Visual inspection of the multimedia distributed application showed that all 4 operational modes where cycled through, each mode lasted exactly 4 seconds before a reconfiguration switched to the next mode. Both displays showed the same video. Concerning the safety critical distributed applications the car behaved as intended. Although some of the races provided in the TORCS racing simulation were challenging, we could win quite a few. The winning rate was similar to operating the racing car simulation with directly attached controls. Also ABS and steering functionality worked as designed.

Figure 6.4a shows the screen output of the 'display b' platform components for each of the multimedia operational modes: normal (full resolution), negative (full resolution), degraded 1 (half resolution), degraded 2 (quarter resolution). Note that the streamed video is overlayed with red text in the lower right screen which informs about the name of the virtual component and the active operational mode. However the white text in the upper center of the screen is part of the streamed video. During degradation modes this text becomes unreadable, while the cube remains recognizable. Figure 6.4b illustrates that both display platform components show the same video feed. Finally, Figure 6.4c gives some impressions about the safety critical distributed application.

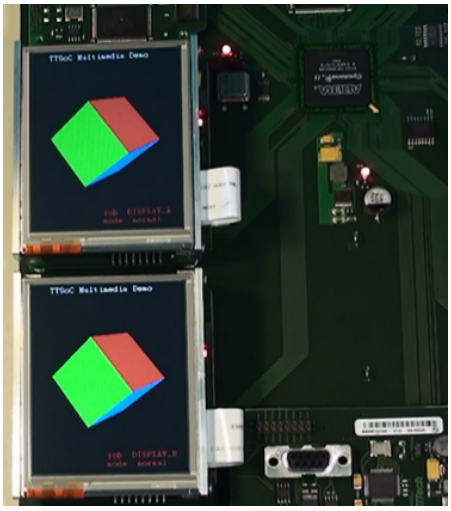
We do not state implementation source code metrics here, because partly the code was generated from models and partly the code was added to existing libraries or applications. Still, we found one metric interesting: the size of the vendor tools required to build the system at hand is about 13 GB. The source code repository has a size of about 2 GB (including the TORCS racing simulation and other libraries/applications).

Thanks to our development approach all platform components and their implementation can be traced back to one or more requirements. Also as we have already observed at the service dependency tree, the two criticality partitions (i.e., the automotive and the multimedia components) remain apart. Hence, the validation of the developed system is trivial. Naturally, while validation is indeed trivial, certification is surely not. However, for this case study we assume that the TTSoC Architecture sufficiently isolates all unrelated distributed applications (We give more detailed support for this claim in [35]). Further, certification is not in the scope of this thesis and we assume that the automotive distributed applications and the TTSoC Architecture elements conform to our non-functional criticality requirement.

[11] http://www.wireshark.org



(a) Multimedia Distributed Application Modes



(b) Multimedia Distributed Application

(c) Automotive Distributed Application

Figure 6.4: Impressions during Platform Runtime

6.6 Concluding Remarks

In this chapter we successfully illustrated our development approach on a case study where all the introduced concepts apparently make sense. They work towards manageable integration of distributed applications. We started with a small set of requirements and arrived at a highly complex distributed system involving lots of heterogeneous subsystems. Viewed at higher abstraction levels, e.g., the service-oriented MFIM, all this complexity melts away. There, we easily could extend the system with the confidence to not influence any unrelated distributed applications.

CHAPTER 7

Conclusion

The main contributions of this thesis are:

- the advancement of the concepts and current implementation of the TTSoC Architecture towards application development, and
- the creation of an application development approach for the TTSoC Architecture.

The chapters 2 and 3 give a concise overview of the used concepts used in this thesis and the TTSoC Architecture.

In Chapter 4, we applied the recursive component concept to the TTSoC Architecture which increased our abilities to manage chip complexity in three ways: *(1)* recursion towards lower abstraction levels, where components can host virtual components and provide a Virtual Interconnect including an 'off-virtual' gateway, *(2)* recursion towards higher abstraction levels, where a component encapsulates a set of components by acting as a gateway for them, *(3)* linking components at the same abstraction level, where a set of components interacts with another set of components through gateways. Immediate benefits of this proposed recursive component model are an elegant description of software executing as virtual components on host components and the abstraction of a set of components behind a gateway component. We believe that these benefits lead to a finer control of chip resources with respect to NoC bandwidth or power scaling as well as possibly improved on-chip TMR. Components that host virtual components give the engineer greater flexibility in regard to component behavior deployment and facilitates the design of a generic TTSoC Architecture chip as a cost-effective Commercial Off the Shelf (COTS) device.

We implemented the Time-Triggered Operating System for Virtual Components (TTOSVC) including hardware support for the soft-core CPUs Altera Nios II and Gaisler LEON3. On top of our own and on top of the Linux operating system we implemented the Virtual Component Runtime Environment (VCRE) that allows for the execution of virtual components implemented in the programming language C. We employ the TTOSVC also in our implementations of the architectural elements RMA, DU and the TTE gateway.

For the TTNoC Interconnect we developed a topology invariant scheduling algorithm that routes ECCs implemented as PDSes through the NoC resource. While the worst-case runtime complexity of the algorithm is exponential in the number of PDS bursts, it is able to find valid schedules and can be used in small systems for dynamic resource management or as a generic offline scheduler for static schedules.

In Chapter 5 we proposed a model-based application development approach for the TTSoC Architecture that is easily generalizable to a development approach for any TTA. Central to our approach is the recursive component concept and the concept of services that we borrowed from the SoA. The development approach allows for top down development starting with a set of requirements at the highest abstraction level. We express the combination of behavioral and non-functional properties as a service which is intended to meet one or more requirements. Through several refinement steps where we decompose these services and obtain a service dependency tree we finally arrive at a service-oriented Macro Fully-specified Interface Model (MFIM). This MFIM is a formal description of the system where services are provided and requested by components. Our tool takes this model, a model of the platform and allocation information as input for an automatized transformation to the PAM. The PAM is the lowest abstraction level that we consider in our approach and describes the system by using platform components. However, for supplied components the development approach also allows for a bottom up technique: the engineer only needs to have a description of the supplied component available in the MFIM and is restricted to the allocation of the supplied components to their respective platform components. In the case of legacy components that cannot be adjusted for direct integration, the engineer needs to design a wrapper component that resolves interoperability issues.

In our case study (cf. Chapter 6) we demonstrated the viability of the development approach by following our proposed design flow step by step. We started with a set of requirements and concluded the case study with a platform that shows all the features we added during refinement. Specifically, the platform meets the original requirements. The case study also proved useful to validate the architectural elements RMA, gateway and DU that we developed to support our design flow.

Answers to Research Questions

Having arrived at the end of this master's thesis, we like to give answers to our research questions from Chapter 1:

- *How would we design and implement distributed applications for the TTSoC Architecture?* Our proposed model-based development approach which is based on scientifically sound methods and backed by our case study allows for designing applications for the TTSoC Architecture.

- *What application properties must be specified and what properties can be automatically derived at which level of abstraction?*

In our approach, the engineer's manual efforts are twofold: first he/she needs to specify services that meet requirements and refine those services until he/she reaches the MFIM. Second, the engineer needs to describe the UFIM-PM allocation (i.e., the deployment platform and allocate UFIM elements on it). Tools can help to automatize two model transformations: the MFIM-to-UFIM and the UFIM-to-PAM.

In regard to parameters, we first concentrate at the highest abstraction level on behavior and non-functional annotations and then derive a conforming LIF specification during the refinement process. This LIF specification is completely defined with respect to message semantics and syntax, but only sufficiently constrained in the time domain. Any lower abstraction level may further refine timing constraints, but must not violate constraints given at higher abstraction levels. Some non-functional properties are resolved during the automatized MFIM-to-UFIM transformation by instantiating architectural plugins, additional components and ECCs. The remaining non-functional properties (e.g., concerning allocation constraints) are resolved during the UFIM-to-PAM transformation.

Future Work

As promising as the initial evaluation of our development approach looks, there remains much to be investigated. For example, our formalization of non-functional properties is only rudimentary. The set of available architectural plugins and the set of available macros determine what kind of non-functional annotations can be handled by the architectural services. Also we consider the validation of the refinement steps during service decomposition crucial: We must ensure that a decomposition still represents the decomposed services, e.g., by formal methods. Next, there are dependencies between the development approach and the architectural elements RMA, gateways and DU. While we made an effort to implement part of their intended services, many details about dynamic reconfiguration, diagnosis and their interplay are still only theoretical considerations for the TTSoC Architecture implementation at hand. However, we consider the development approach and our advancements of the TTSoC Architecture concerning concepts and implementation as an important enabling contribution for this future work.

Acronyms

AAM Abstract Application Model

ABS Anti-Lock Braking System

ATL Atlas Transformation Language

API Application Programming Interface

CAN Controller Area Network

CI Control Interface

CIA Confidentiality, Integrity and Availability

CIS CAN Interface Subsystem

CSMA/CA Carrier Sense Multiple Access/Collision Avoidance

CNI Communication Network Interface

COTS Commercial Off the Shelf

CPU Central Processing Unit

DAS Distributed Application Subsystem

DU Diagnostic Unit

ECC Encapsulated Communication Channel

ECU Embedded Control Unit

EMOF Essential Meta-Object Facility

ESB Enterprise Service Bus

ETCS Electronic Throttle Control System

FCU Fault Containment Unit

FIM Fully-Specified Interface Model

FPS Frames per Second

fUML Foundational Subset for Executable UML

GW Gateway

HAL Hardware Abstraction Layer

IP Intellectual Property

LIF Linking Interface

MFIM Macro Fully-specified Interface Model

MDA Model-Driven Architecture

MDD Model-Driven Development

MDE Model-Driven Engineering

MOF Meta-Object Facility

MPSoC Multi Processor System-on-Chip

NI Network Interface

NoC Network-on-Chip

NTP Network Time Protocol

OCP Open Core Protocol

OEM Original Equipment Manufacturer

OMG Object Management Group

PAM Platform Allocation Model

PDS Pulsed Data Stream

PI Port Interface

PIM Platform Independent Model

PM Platform Model

PSM Platform Specific Model

QVT Query View Transformation

RMA Resource Management Authority

SCADE Safety-Critical Application Development Environment

SEU Single Event Upset

SoA Service oriented Architecture

SoS System of Systems

TDMA Time-Division-Multiple-Access

TMR Triple Modular Redundancy

TNA Trusted Network Authority

TTA Time-Triggered Architecture

TTE Time-Triggered Ethernet

TTOSVC Time-Triggered Operating System for Virtual Components

TISS Trusted Interface Subsystem

TTNoC Time-Triggered Network-on-Chip

TTSoC Time-Triggered System-on-Chip

UFIM Unified Fully-specified Interface Model

UNI Uniform Network Interface

UML Unified Modeling Language

VCPI Virtual Component Programming Interface

VCRE Virtual Component Runtime Environment

WCET Worst Case Execution Time

XMI XML Metadata Interchange

xUML Executable UML

Bibliography

[1] Parosh Aziz Abdulla, Johann Deneux, Gunnar Stålmarck, Herman Ågren, and Ove Åkerlund. Designing safe, reliable systems using scade. In *Leveraging Applications of Formal Methods*, pages 115–129. Springer, 2006.

[2] Andreas Abel, Florian Benz, Johannes Doerfert, Barbara Dörr, Sebastian Hahn, Florian Haupenthal, Michael Jacobs, Amir H Moin, Jan Reineke, Bernhard Schommer, et al. Impact of resource sharing on performance and performance prediction: A survey. In *CONCUR 2013–Concurrency Theory*, pages 25–43. Springer, 2013.

[3] European Aviation Safety Agency. *Certification Memorandum, Development Assurance of Airborne Electronic Hardware*, 2011. See `http://www.easa.europa.eu/certification/docs/certification-memorandum/EASACM-SWCEH-001DevelopmentAssuranceofAirborneElectronicHardware.pdf`.

[4] Ravindra K Ahuja, Thomas L Magnanti, and James B Orlin. Network flows: theory, algorithms, and applications. 1993.

[5] Terry Bahill and Jesse Daniels. Using objected-oriented and uml tools for hardware design: A case study. *Systems Engineering*, 6(1):28–48, 2003.

[6] Lubomir Bic and Alan C Shaw. *Operating systems principles*. Prentice Hall, 2003.

[7] Peter Bishop and Robin Bloomfield. A methodology for safety case development. In *Industrial Perspectives of Safety-critical Systems*, pages 194–203. Springer, 1998.

[8] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool, 2012.

[9] Christopher Brooks and Edward A. Lee. Ptolemy ii - heterogeneous concurrent modeling and design in java, February 2010. `http://chess.eecs.berkeley.edu/pubs/655.html`.

[10] Frank P Coyle and Mitchell A Thornton. From uml to hdl: a model driven architectural approach to hardware-software co-design. In *Information systems: new generations conference (ISNG)*, volume 1, pages 88–93, 2005.

[11] Edsger W Dijkstra. Co-operating sequential processes. f. *Programming Languages. Academic Press, New York*, 1968.

[12] Johan Eker, Jorn W Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Sonia Sachs, Yuhong Xiong, and Stephen Neuendorffer. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[13] Christian El Salloum. *Interface design in the time-triggered system-on-chip architecture*. PhD thesis, Vienna University of Technology, 2007.

[14] Christian El-Salloum, Roman Obermaisser, Bernhard Huber, and H. Kopetz. A time-triggered system-on-a-chip architecture with integrated support for diagnosis. In *Workshop on Diagnostic Services in Network-on-Chips*, 2007.

[15] Gerhard Alois Engleder. A resource management scheme for the TT-SoC architecture. Master's thesis, Vienna University of Technology, 2007.

[16] Stephan Esch, Jürgen Meyer, and Günter Linn. Partial networking deactivation of inactive ecus. *ATZautotechnology*, 12(1):52–57, 2012.

[17] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding code mobility. *Software Engineering, IEEE Transactions on*, 24(5):342–361, 1998.

[18] Philipp Gutwenger. Model-driven design for the time-triggered soc architecture, 2009.

[19] Thomas A Henzinger and Joseph Sifakis. The discipline of embedded systems design. *Computer*, 40(10):32–40, 2007.

[20] B. Huber, R. Obermaisser, and P. Peti. Mda-based development in the decos integrated architecture - modeling the hardware platform. In *Object and Component-Oriented Real-Time Distributed Computing, 2006. ISORC 2006. Ninth IEEE International Symposium on*, pages 10 pp.–, 2006.

[21] Bernhard Huber. *Resource management in an integrated time-triggered architecture*. PhD thesis, Vienna University of Technology, 2007.

[22] John Hutchinson, Gerald Kotonya, Ian Sommerville, and Steve Hall. A service model for component-based development. In *Euromicro Conference, 2004. Proceedings. 30th*, pages 162–169. IEEE, 2004.

[23] Nicolai M. Josuttis. *SOA in practice: The Art of Distributed System Design*. O'REILLY, 2007.

[24] Roland Kammerer, Roman Obermaisser, and Bernhard Frömel. A Router for the Containment of Timing and Value Failures in CAN. In *EURASIP Journal on Embedded Systems*, 2012.

[25] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer international series in engineering and computer science: Real-time systems. Springer, 2011.

[26] Hermann Kopetz. Pulsed data streams. In *DIPES*, pages 105–114, 2006.

[27] Hermann Kopetz, Astrit Ademaj, Petr Grillinger, and Klaus Steinhammer. The time-triggered ethernet (tte) design. In *Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on*, pages 22–33. IEEE, 2005.

[28] Hermann Kopetz and Günther Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

[29] E.A. Lee. Cyber physical systems: Design challenges. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 363–369, 2008.

[30] Edward A. Lee. Computing for embedded systems. volume 3, pages 1830–1837. IEEE; 1999, 2001.

[31] Ekaterina Moreva, Giorgio Brida, Marco Gramegna, Vittorio Giovannetti, Lorenzo Maccone, and Marco Genovese. Time from quantum entanglement: an experimental illustration. 2013. `http://arxiv.org/abs/1310.4691`.

[32] Roman Obermaisser, Christian El Salloum, Bernhard Huber, and Hermann Kopetz. The time-triggered system-on-a-chip architecture. In *Industrial Electronics, 2008. ISIE 2008. IEEE International Symposium on*, pages 1941–1947. IEEE, 2008.

[33] Roman Obermaisser, Christian El Salloum, Bernhard Huber, and Hermann Kopetz. From a federated to an integrated automotive architecture. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(7):956–965, 2009.

[34] Roman Obermaisser, Christian El Salloum, Bernhard Huber, and Hermann Kopetz. Fundamental design principles for embedded systems: The architectural style of the cross-domain architecture genesys. In *Object/Component/Service-Oriented Real-Time Distributed Computing, 2009. ISORC'09. IEEE International Symposium on*, pages 3–11. IEEE, 2009.

[35] Roman Obermaisser, B Frömel, Christian El Salloum, and Bernhard Huber. Integrating safety and multimedia subsystems on a time-triggered system-on-a-chip. In *Industrial Informatics, 2008. INDIN 2008. 6th IEEE International Conference on*, pages 270–275. IEEE, 2008.

[36] Inc. Object Management Group. *Semantics of a Foundational Subset for Executable UML Models (fUML), v1.1*, 2013. `http://www.omg.org/spec/FUML/1.1/PDF/`.

[37] Christian Peter Paukovits. *The Time-Triggered System-on-Chip Architecture*. PhD thesis, Vienna University of Technology, 2008.

[38] RM Pirsig. Zen and the art of motorcycle maintenance. *New York: Bantam*, 1974.

[39] Christian El Salloum, Martin Elshuber, Oliver Hoftberger, Haris Isakovic, and Armin Wasicek. The across mpsoc–a new generation of multi-core processors designed for safety-critical embedded systems. In *Digital System Design (DSD), 2012 15th Euromicro Conference on*, pages 105–113. IEEE, 2012.

[40] Andreas Schranzhofer, Jian-Jia Chen, and Lothar Thiele. Timing predictability for resource sharing multicore systems-challenges and open problems. *eraerts*, page 52, 2011.

[41] Roman Seiger. An extensible interface subsystem for a novel time-triggered system-on-a-chip architecture . Master's thesis, Vienna University of Technology, 2008.

[42] Suresh Siddha, Venkatesh Pallipadi, and AVD Ven. Getting maximum mileage out of tickless. In *Linux Symposium*, volume 2, pages 201–207. Citeseer, 2007.

[43] Herbert A Simon. The architecture of complexity. *Proceedings of the American philosophical society*, 106(6):467–482, 1962.

[44] Herbert Stachowiak. Allgemeine modelltheorie. 1973.

[45] William Stallings. *Operating Systems: Internals and Design Principles, 6/E*. Pearson Education India, 2009.

[46] OASIS Standard. *Reference Model for Service Oriented Architecture 1.0*, 2006.

[47] Klaus Steinhammer. *Design of an FPGA-Based Time-Triggered Ethernet System*. PhD thesis, PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 2006.

[48] Carl A Sunshine. Source routing in computer networks. *ACM SIGCOMM Computer Communication Review*, 7(1):29–33, 1977.

[49] Bernhard Weirich. A resource management scheme for the TT-SoC architecture . Master's thesis, Vienna University of Technology, 2008.