Diplomarbeit

Design and Implementation of an AUTOSAR Abstraction Layer for a FlexRay Network Co-Simulation

Stefan Krug

Institut für Technische Informatik
Technische Universität Graz
Vorstand: O. Univ.-Prof. Dipl.-Ing. Dr. techn. Reinhold Weiß

Begutachter: Ass.-Prof. Dipl.-Ing. Dr. techn. Christian Steger
Betreuer: Ass.-Prof. Dipl.-Ing. Dr. techn. Christian Steger,
Dipl.-Ing. Michael Karner

Graz, im Dezember 2009

Kurzfassung

Die Elektronik in modernen Fahrzeugen bildet ein sehr komplexes verteiltes System. Dabei ist nicht nur die Hardware, sondern auch die Software von essentieller Wichtigkeit um eine korrekte Funktionalität des Systems zu gewährleisten. Um die Portabilität der Software zwischen Hardware verschiedener Hersteller zu ermöglichen beziehungsweise zu vereinfachen, wird immer häufiger eine AUTOSAR Middleware für die Abstraktion der hardwarespezifischen Details zum Einsatz gebracht.

Die komplexen Netzwerke aus Sensoren, Aktuatoren und Steuergeräten müssen speziell in sicherheitskritischen Bereichen umfassenden Tests unterzogen werden. Es ist dabei erstrebenswert, Fehler sowohl der Hardware als auch der Software bereits in einem frühen Stadium des Entwicklungsprozesses — möglichst noch in einer Simulation — zu finden um die Kosten gering zu halten.

Um das Zusammenwirken zwischen der Kommunikationshardware und der verteilten Software zu untersuchen wurde in dieser Diplomarbeit eine Möglichkeit geschaffen um Softwarekomponenten und deren Kommunikation zu simulieren. Diese Arbeit wurde im Rahmen des TEODACS Projekts durchgeführt und erweitert eine existierende Cosimulationsumgebung ausgehend von einem SystemC Modell eines FlexRay Communication Controllers um ein AUTOSAR nahes Software Komponenten Interface. Die für AUTOSAR typische Trennung zwischen Hardware (SystemC Modell) und Software ermöglicht es, Softwarekomponenten zu entwickeln und zu simulieren, ohne sich mit Details der Hardware auseinandersetzen zu müssen. Durch die Einhaltung der in AUTOSAR spezifizierten Interfaces wird eine Wiederverwendung von Code zwischen Simulationsumgebung und Hardware-Prototypen vereinfacht.

Abstract

The electronic components of modern vehicles are forming complex distributed networks. Not only the hardware, also the software is of essential importance for ensuring fault-free operation of the system. To facilitate the portability of software among hardware components of different vendors, AUTOSAR is becoming the de-facto middleware layer.

The complex networks of sensor, actuator and electronic control units need to undergo demanding tests. Especially for safety-critical systems the test methods must impose severe demands on the devices under test. It is desirable to detect faults of the hardware as well as the software in an early stage of the development process. To minimise costs, at best all faults are detected already in a simulation.

To analyse the cooperation between communication hardware and a distributed software system, in this thesis a concept for distributed software simulation in a co-simulation environment is developed. In cooperation with the Virtual Vehicle Competence Center in the context of the TEODACS project, based on a SystemC model of a FlexRay communication controller, a possibility was established to develop and simulate software components with an AUTOSAR like interface. The AUTOSAR-typical separation of hardware and software permits the development of applications without needing to take care of underlying hardware details. The application behaviour and communication is examined and validated in a co-simulation environment. By complying with the AUTOSAR specification a reuse of code between the co-simulation environment and a prototypical hardware platform is simplified.

Statutory Declaration

Acknowledgement

First of all I would like to thank my advisor at the Institute for Technical Informatics, Michael Karner, whose encouragement, guidance and support from the initial to the final level made this thesis possible. His input on the subject as well as on scientific article writing was of great value. I would also like to thank my supervisor Christian Steger. He provided valuable hints concerning contents and formal structure of the thesis. Furthermore the TEODACS project members at the Virtual Vehicle Competence Center deserve special thanks for their support. Especially the enthusiasm of Eric Armengaud was an excellent motivation.

I would like to show my gratitude to my family, and especially my parents, for their absolute confidence in me. They always kept me away from family responsibilities and encouraged me to concentrate on my study.

Finally, I would like to thank everybody who was important to the successful realisation of thesis, as well as expressing my apology that I could not mention personally one by one.

Contents

Chapter 1

Introduction

The influence of software engineering on the automotive industry grows steadily. An increasing number of functionalities with increasing complexity are deployed in vehicles for the sake of higher security and enhanced comfort for the driver. Therefore, methodologies are developed to improve the efficiency and reduce the error-proneness of the developed software. Also this thesis is settled in the domain of automotive software engineering and deals with the simulation of automotive embedded software.

The first section of this introduction illustrates the problems and challenges on the electronic systems used in vehicles and gives a survey of how the automotive industry strives to overcome them. The second section will express the motivation for this thesis. An overview of the goals is given in section three. Finally the last section contains details on the structure of this document.

1.1 Automotive Electronics

Whereas previously cars were mechanical miracles — the only electrical components were lightening and ignition — nowadays cars include the most modern electronic systems with varying fields of operations like engine control, climate control or power saving technologies. Even before we enter a car, microcontrollers are already at work, checking whether we are allowed to enter. So it is not hard to comprehend the complexity of automotive electronics. This section gives an insight into the employed technologies for communication networks and illustrates the problems of software engineering in the automotive domain.

1.1.1 Bus Systems in Vehicles

Along with the ever growing number of processing units in modern vehicles comes a higher demand on the communication networks. Furthermore the range of electronically controlled application has increased drastically and nowadays includes multimedia application as well as safety critical x-by-wire systems. Those safety critical systems also introduce demands on dependability of the communication networks. As additional complication, the environment for the electronic systems in cars is rather hostile (e.g., electromagnetic fields, high temperature gradients and dirt) [49].

Traditionally event triggered bus systems were used in car networks. The problem with event triggered systems is, that the load on the network is hard to estimate, which can lead to congestions and failures on the network. This makes them unsuitable for safety critical systems [19].

In order to overcome the problems mentioned above, time triggered bus systems are used. A fixed schedule is used for the communication, each node on the bus is granted bus access for an a priori known time interval. Because the schedule has to be configured at design time, knowledge of the communication load is already needed at design time.

The following list gives a short overview of the most important bus systems (all of them are serial buses) used in the automotive domain and their characteristics and advantages and disadvantages:

- CAN: The Controller Area Network (CAN) is an example of an event triggered bus system. It is the de-facto standard for automotive communication networks. The low capacity of CAN limits its use for high traffic applications [1]. For enhanced dependability also a time triggered extension to CAN exists [25].
- LIN: The Local Interconnect Network (LIN) is a very cheap and slow system. Often LIN is used as sub-bus to connect multiple sensors or actuators to a CAN bus system. A single master controls the bus and polls the connected slaves. To further reduce costs, LIN can use the chassis as return path resulting in a single required wire for communication [4].
- byteflight: The byteflight bus system[1] uses a time triggered approach. It is an optical bus system and combines the determinism of a time triggered system and the flexibility of a event triggered system by providing different message classes.
- FlexRay: The FlexRay protocol is a pure time triggered protocol. It is a successor of byteflight and offers a deterministic communication system with relatively high data rates [2]. Further details of the FlexRay protocol are explained in section 2.1.
- MOST: The Media Oriented Systems Transport (MOST) bus was originally designed for multimedia applications in the automotive environment. It uses optical fiber as transportation medium and allows high data rates for peripherals like car radios, CD and DVD players and GPS navigation systems [14].

1.1.2 Challenges of Automotive Software Development

The amount of Software in cars grows rapidly. This is caused by the demand for innovations and the availability of hardware which is getting cheaper and more powerful from generation to generation.

With the increased volume and complexity of the embedded software, naturally also the error rate rises. It is not only a rare occurrence, that a single system contains up to 20.000 errors, which often have to be fixed after delivery with expensive software updates [46]. To reduce software development time and costs and to avoid errors, it is desirable, that the major part of the software can be reused. Currently, in many sub-domains the functionality from one car generation to the next is only changed and enhanced by 10% while more than 90% of the software is rewritten. The reason is a low level, hardware specific implementation, which makes it difficult to change, adopt, and port existing code [20].

In order to be able to reuse software in embedded systems, the applications must be as independent as possible from details of the underlying hardware. Individual software modules must have an optimal modularity. This means that one function (e.g., central door looking system, exterior light) might consist of different individual subcomponents. Building subcomponents improves the reusability, since a functional change can imply only changing a single subcomponent [28].

An in-depth analysis of challenges in the field of automotive software engineering is given by Pretschner et al. in [39]. They list demands on the automotive software and describe the architectural consequences.

1.1.3 Operating Systems in Automotive Embedded Systems

Schröder-Preikschat analyses the use of operating systems in the automotive domain in [44]. The electronic components of modern vehicles form highly distributed systems with up to 80 processing devices

[1] http://www.byteflight.com

and a completely heterogeneous structure. The electronic system includes processors with 8-, 16- and 32-bit technology which are developed by different suppliers, resulting in a different instruction set and input/output behaviour. Concerning the interconnection networks, there is also a great variety of employed technologies (see subsection 1.1.1).

There must be differentiated between two fields of application for automotive operating systems. On the one hand, there is the multimedia domain (e.g., audio, video, telephony) with relatively high demands on the processing power. Here, no hard real-time constraints are encountered, which makes it possible to employ also general purpose operating systems. Preferred are QNX[2] and VxWorks[3] but, rarely, also variants of Linux and WindowsCE are used. On the other hand, there are the classic control systems with hard real-time constraints. In this domain, mainly OSEK (Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen) compatible operating systems are used. OSEK [36] defines an European operating system standard. Also the constraints on interrupt handling depend on the problem domain. Whereas it is tolerable for multimedia applications, to miss interrupts, for control systems, usually no loss of interrupts is allowed, which prohibits the usage of some operating systems on some problem domains.

1.2 Motivation

As already mentioned before, in modern vehicles, a very large number of Electronic Control Units (ECUs) form a greatly distributed system. The communication between the components is essential to ensure the correct functionality of the vehicle. Especially safety critical applications require predictable and dependable operation of the communication devices. At the Virtual Vehicle Competence Center (ViF) a project for gaining expertise to cope with those problems is ongoing. The main goal of the TEODACS project[4] (Test, Evaluation and Optimization of Dependable Automotive Communication Systems) is to examine the modern FlexRay communication system. In addition to a realistic replica of a FlexRay network in the laboratory (FlexRayXpert.Lab), the entire distributed system is simulated within a co-simulation environment (FlexRayXpert.Sim) [16].

As layer between the hardware and the application, AUTOSAR[5] (AUTomotive Open System ARchitecture) is becoming the de-facto standard middleware. In the context of the TEODACS project, also the behaviour of applications should be examined. To enable an efficient integration of application code into the co-simulation environment, an AUTOSAR layer is integrated into the co-simulation environment and, as middleware, introduces the necessary abstraction of the hardware and eases the application development by providing standardised interfaces.

1.3 Goal

The goal of this work is to design and implement an abstraction layer for the aforementioned co-simulation environment which allows the easy integration of application code. The interfaces between the developed applications and the co-simulation environment should comply to the AUTOSAR standard. This provides a high amount of flexibility and simplifies the reuse of code between the laboratory setup and the co-simulation environment. Therefore, a concept for a realisation of the Virtual Function Bus (VFB) specified by AUTOSAR should be developed which allows the communication between application software components across ECU boundaries. AUTOSAR describes a sender-receiver and a client-server communication model. The abstraction layer should provide both mechanisms to the application developer.

[2]http://www.qnx.com/
[3]http://vxworks.com/
[4]http://www.teodacs.com
[5]http://www.autosar.org

AUTOSAR requires the application to define ports, data elements and operations for specifying the needed communication elements of the application. Therefore a possibility to specify this configuration in the context of the VFB abstraction layer must be provided. The configuration specification should be accomplished as easy as possible.

In connection with the TEODACS project, numerous models representing the communication infrastructure (e.g., cable, communication controller) were already implemented and tools for the configuration of those models do exist. The implementation has to be done in a way that an efficient integration with the existing environment and a reuse of the existing tools is possible.

1.4 Structure of this Document

Chapter 2 gives an overview of related work. First it introduces the AUTOSAR and FlexRay specifications which are of high importance for the understanding of this thesis. Afterwards it gives an insight into the state of the art of software simulation on different abstraction levels and finally presents the existing co-simulation environment. **Chapter 3** illustrates the concept of the implementation performed in context of this thesis. It provides use cases and presents requirements and reasoning for made design decisions. A detailed description of the implementation can be found in **chapter 4**. This chapter does also provide usage notes for the presented implementation. In **chapter 5** applications are presented which demonstrate the functionality of the system and finally **chapter 6** concludes this thesis and gives ideas for further enhancements and research.

Chapter 2

State of the Art of Automotive Embedded Software and Software Simulation

This chapter is dedicated to related work. The first section introduces the FlexRay protocol specification to provide an overview of supported features and required parameters. The next section deals with the AUTOSAR specification. It illustrates the layered software concept and gives details about communication system relevant parts of the specification. Afterwards, the state of the art of software simulation on different abstraction levels is discussed. Starting with high detailed instruction set up to abstract functional simulation, many aspects of software simulation are covered. Also possibilities for the simulation of AUTOSAR software are analysed and an open source AUTOSAR implementation is shortly evaluated. The next subsection concerns delay and timing analysis on various communication systems. The last part of this chapter gives an overview of the co-simulation environment of the TEODACS project and introduces used tools and included simulation models.

2.1 Overview of the FlexRay Protocol

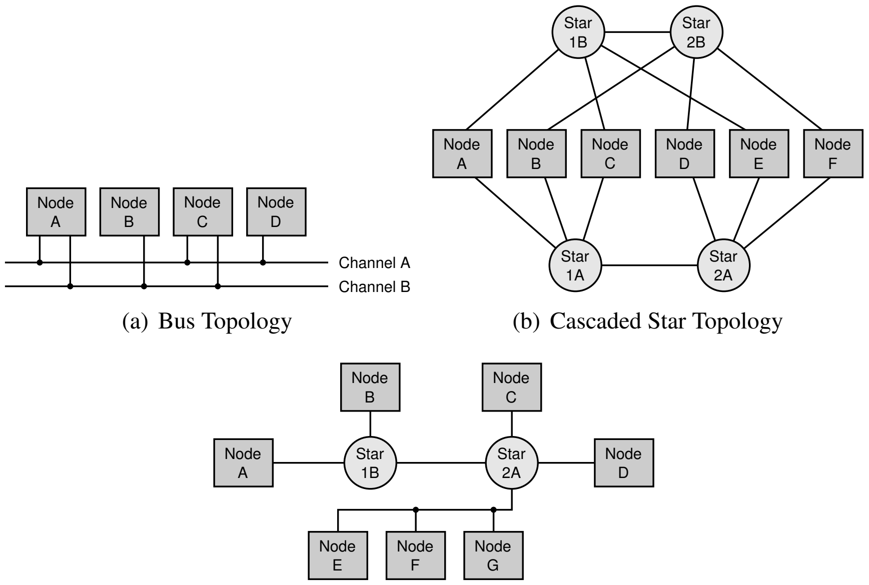
The FlexRay communication system was designed to support the increasing demand on communication of in-car networks. It guarantees a dependable and fault-tolerant communication between the high number of ECUs which are present in modern vehicles.

The FlexRay protocol related parts of this thesis are based on the *FlexRay Communications System - Protocol Specification, v2.1 Revision A* [2]. It is downloadable upon free registration from the homepage of the FlexRay consortium[1]. In addition to the protocol specification also extensive specifications for the electrical physical layer and for conformance tests for all aspects of the protocol exist and are also available for download.

The most important parts of the FlexRay protocol specification which may be required for understanding of this work are summarised in this section. At first the supported bus topologies are shown. In the following subsection, the media access scheme which is used by FlexRay is explained. The next subsection briefly illustrates the FlexRay frame structure. Finally some information about the interface between the FlexRay controller and the host processor is given.

Regarding speed, the FlexRay system is targeted to support a data rate of 10 Mbit/sec available on two channels, giving a gross data rate of up to 20 Mbit/sec. Concerning network topology, FlexRay provides a great amount of flexibility. Star and bus topologies as well as various hybrid configurations with one or two channels are possible. Figure 2.1 shows examples of different configurations.

[1]http://www.flexray.com



(a) Bus Topology (b) Cascaded Star Topology

(c) Hybrid Topology

Figure 2.1: Different network topologies supported by FlexRay. [2]

2.1.1 Media Access Control

The media access scheme of the FlexRay protocol is based on a recurring communication cycle. Within this cycle, the time is split up into segments. The scheme is depicted in figure 2.2.

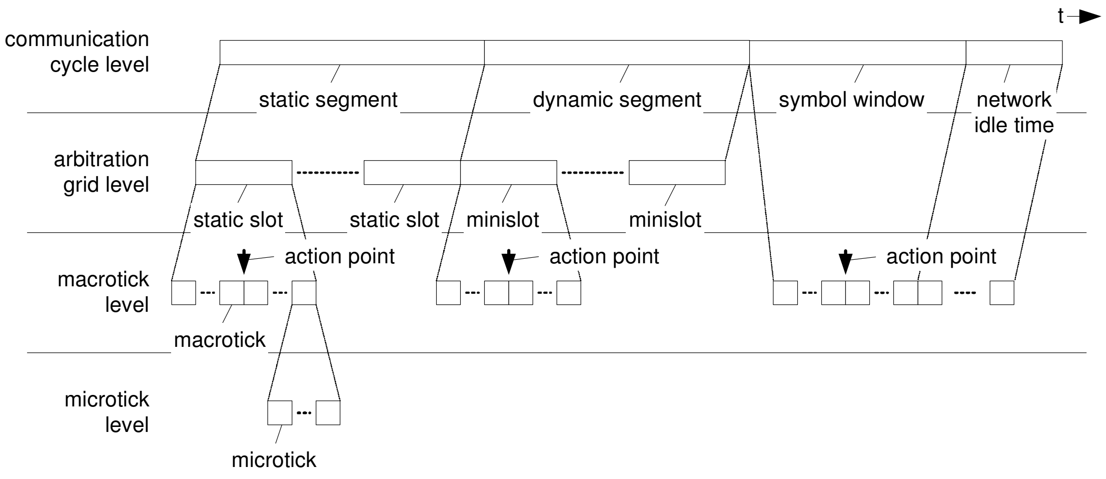


Figure 2.2: FlexRay media access control scheme. [2]

At the beginning of each cycle the static segment is transmitted. This segment is divided by Time Division Multiple Access (TDMA) scheme into a fixed number of statically configured slots which are statically assigned to communication nodes.

Following the static segment, the dynamic segment is the next part of the FlexRay cycle. It consists of slots too, but the slots in the dynamic segment are typically of smaller size than slots of the static segment. Those slots are called minislots. In contrast to the static segment, a message does not have a fixed size which is bound to the size of the slot. A message may occupy more than one minislot. The minislots are distributed according to requirements and priorities of the messages among the nodes. No guarantee is given that a node is able to send data in every cycle. Therefore the dynamic segment is usable for uncritical data only.

Optionally, after the dynamic segment a symbol window is defined. This time of the cycle is reserved for sending unique bit-patterns over the bus which have special purposes like waking up other nodes or signalling the startup of a communication cluster.

The remaining time of the cycle is called network idle time and is unused in terms of communication. The nodes may synchronise its internal clocks and perform other implementation specific communication related tasks during the network idle time.

2.1.2 Frame Format

All transmitted data has, according to the FlexRay protocol specification, to be encapsulated in frames. The structure of the frame format is depicted in figure 2.3.

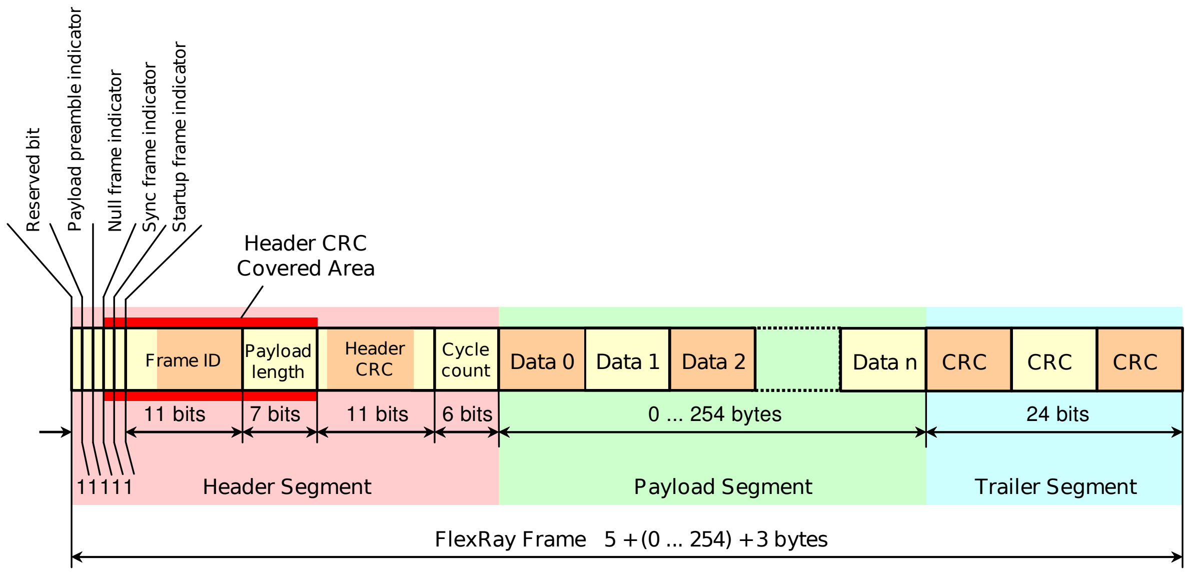


Figure 2.3: The frame format defined by the FlexRay protocol specification. [2]

A frame contains in addition to the payload segment a header and a trailer segment. The header segment contains information about the state of the FlexRay schedule (e.g., frame ID, cycle count) and some flags which indicate special purposes of the frame (e.g., sync frame indicator, startup frame indicator). A Cyclic Redundancy Check Code (CRC) is calculated over parts of the header. Following the payload, the trailer segment contains a checksum calculated over the whole frame. The checksums allow the receiver to check if errors occured during transmission of the frame data.

2.1.3 Controller Host Interface

The FlexRay protocol also defines an interface for the interaction between a host processor and the FlexRay protocol engine. This interface is called Controller Host Interface (CHI) and pictured in figure 2.4. It is divided in two major blocks, called protocol data interface and message data interface.

The protocol data interface manages all data exchange relevant for the protocol operation. Examples are protocol configuration data (e.g., the duration of a static slot in the static segment, the length of a communication cycle) and protocol status data (e.g., the current sate of the controller, the current point in the schedule (cycle, slot)).

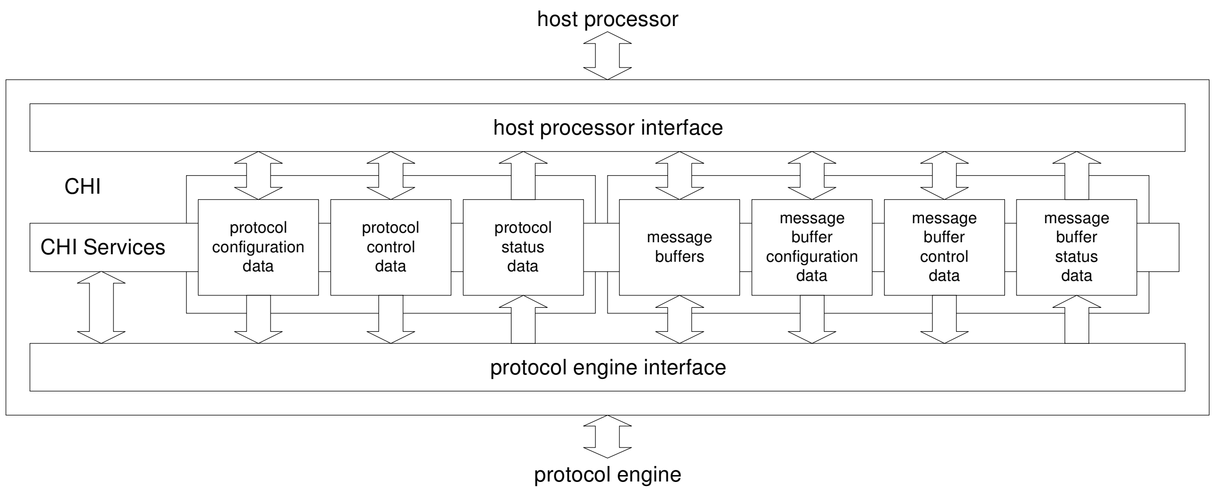

host processor
host processor interface
CHI
CHI Services
protocol configuration data
protocol control data
protocol status data
message buffers
message buffer configuration data
message buffer control data
message buffer status data
protocol engine interface
protocol engine

Figure 2.4: Block diagram of the Controller Host Interface (CHI). [2]

The message data interface manages all data exchange relevant for the exchange of messages. It handles for example the assignment of transmission buffers and the retrieval of frame status and message data of received frames.

Although the requirements on the CHI are specified in the FlexRay protocol specification, the details of the implementation depend in large parts on the manufacturer. Additional information of the CHI of the controller model used for this work can be found in section 4.3.2.

2.1.4 Startup of a FlexRay Cluster

The startup of a FlexRay cluster is initiated by so-called coldstart nodes. The coldstart node which actively starts the cluster is called the leading coldstart node. It sends a collision avoidance symbol and starts transmitting frames with set startup bit in the keyslot assigned to the node. After four cycles it is joined by the other coldstart nodes of the cluster, called following coldstart nodes, which now start transmitting startup frames in their keyslots themselves. After all coldstart nodes are running, the remaining nodes can integrate on the now already established schedule. An example of a fault-free startup is depicted in figure 2.5. If no problems occur, the startup of the network requires seven communication cycles. All nodes of the cluster can leave startup at the end of cycle seven, just before entering cycle eight. For a typical FlexRay cycle duration of 5 ms this means the startup is complete after about 40 ms.

2.2 Introduction to AUTOSAR

AUTOSAR (AUTomotive Open System ARchitecture) is a partnership of automotive manufacturers and suppliers working together to develop and establish an open industry standard for automotive electric/-electronic architectures. This is necessary to cope with the increasing demands on in-car electronics. A common interface with a proper hardware abstraction allows reusing existing code among ECUs of different manufacturers. Along with the reduction of costs due to better reusability also come increased quality and additional flexibility [13].

This section gives an introduction to AUTOSAR. It explains the layered software architecture concepts and explains in a short summary the tasks and features of the different layers. Especially emphasised are the parts of the specification which are related to FlexRay communication and communication in general and which are relevant for the understanding of this thesis.

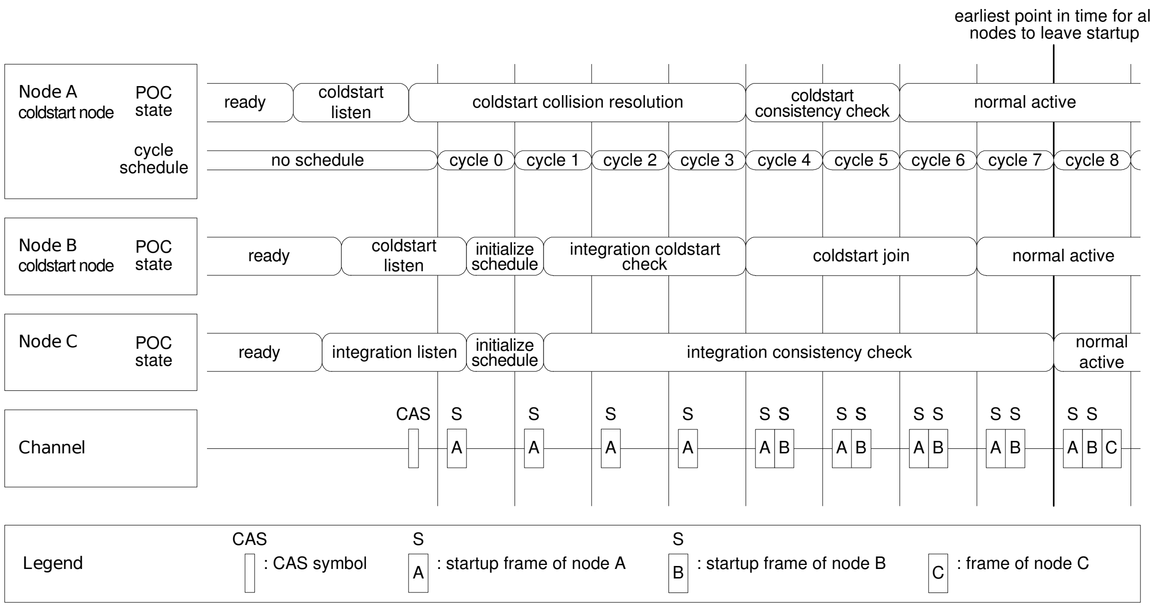


Figure 2.5: Schema of the system startup used by the FlexRay protocol. [2]

2.2.1 Layered Software Concepts of AUTOSAR

The concept of AUTOSAR follows a layered design, the whole system is split in three layers. The layered architecture, which is depicted in figure 2.6, ensures a decoupling of the hardware and the software by providing increasing degrees of abstraction. The Basic Software Layer is highly dependent on the underlying hardware, whereas the Run-Time Environment (RTE) solely uses modules of the Basic Software and, therefore, is quite independent from details of the hardware. The RTE provides standardised services which are identical on every type of hardware to the Application Layer. The application uses services provided by the RTE and can therefore be easily ported to different ECUs.

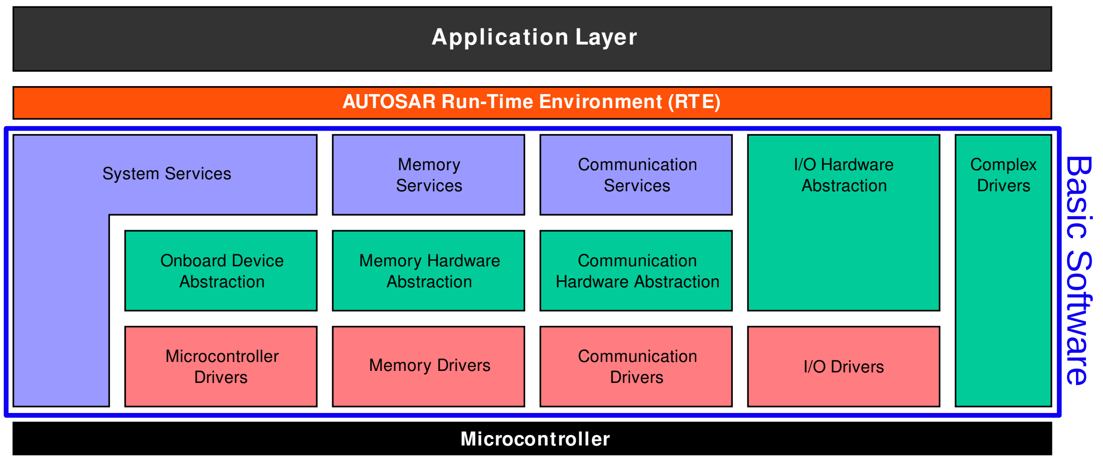


Figure 2.6: Overview of the AUTOSAR abstraction layers. [5]

The following subsections describe the three AUTOSAR abstraction layers in greater detail, starting at the Application Layer, continuing with the RTE and finally moving on to the modules of the Basic Software Layer which are relevant to the communication.

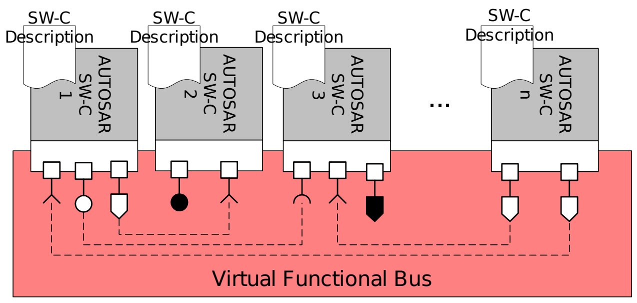
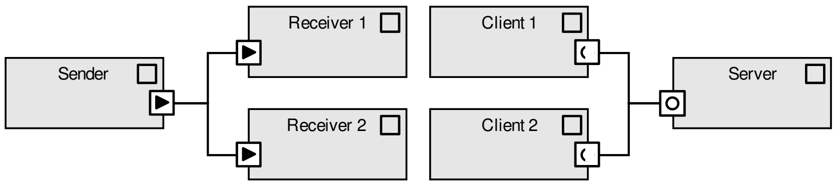


Figure 2.7: Abstract communication via the VFB. [13]

2.2.2 Application Layer

From the point of view of an application developer, AUTOSAR consists of a set of software components which can communicate via a VFB. Those software components represent parts of an application and have a standardised AUTOSAR interface. In addition to the implementation, a software component description is required for each software component. In this description the required interfaces, needed resources and other aspects essential for the integration of the software component are listed. Figure 2.7 depicts the VFB and the virtual communication channels between some software components [12].

The software components get mapped on different ECUs later. In this design step, the VFB communication is also affected. Two software components may get mapped to the same ECU. In this case, the communication between those two components is performed locally via shared memory. Otherwise, if the software components are mapped to distinct ECUs, the communication is handled by network-specific communication mechanisms (e.g., FlexRay frames, CAN). This whole mechanism is handled by the AUTOSAR Basic Software and is transparent to the developer of the software component.



(a) Sender-receiver communication model. (b) Client-server communication model.

Figure 2.8: VFB communication modes. [12]

The communication on the VFB can follow two different models. In figure 2.8 the sender-receiver and client-server communication model are depicted. The differences are explained in the following sections.

2.2.2.1 Sender-Receiver Communication Model

The sender-receiver communication model enables a sender software component to distribute information to one or several receivers. The sender neither has information on the identity of the receivers it sends data to, nor on its number. Therefore, the sender-receiver model allows for strong decoupling of the communication partners.

Two paradigms for data transfer are possible:

- last-is-best: Every time a new value is received by a receiver port, the old value is discarded and the new value is used. Using this communication model, it is only possible to have a $1 : n$ relationship

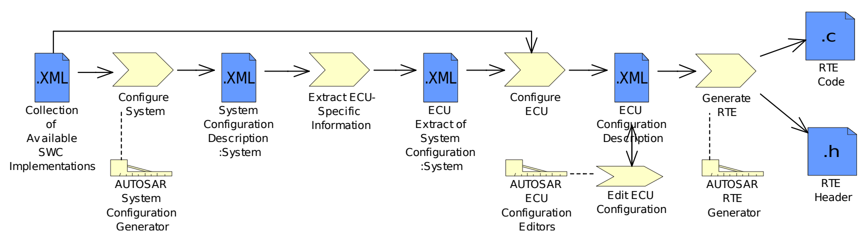

.XML
Collection of Available SWC Implementations
Configure System
AUTOSAR System Configuration Generator
.XML
System Configuration Description :System
Extract ECU-Specific Information
.XML
ECU Extract of System Configuration :System
Configure ECU
AUTOSAR ECU Configuration Editors
.XML
ECU Configuration Description
Edit ECU Configuration
Generate RTE
AUTOSAR RTE Generator
.c
RTE Code
.h
RTE Header

Figure 2.9: Data flow and dependencies of the RTE generation phase. [11]

between senders and receivers (with $n \geq 0$).

- queued: Consecutive values of the sender are stored in a queue and each read of the data consumes one value. The queue may either be on the sender or on the receiver side, making $1 : n$ as well as $n : 1$ relationships possible ($n \geq 0$). To avoid further increase of the complexity, $m : n$ relationships are not allowed.

For primitive data types, it is also possible to enable a filtering mechanism between the sender and the receiver. If a filter is installed, the new value only gets added to the queue or overwrites the old value, if it matches the filter constraints.

2.2.2.2 Client-Server Communication Model

The client-server model takes a different approach on the communication. The server provides a service which can be used by a client. The communication is initiated by the client, sending a request for performing a specific operation to the server. The parameters for the operation are specified by the client in the request. After finishing the operation, the server replies to the client. The response indicates whether the operation was successful and does contain return values for the client. The state of the server might change due to an operation call.

2.2.3 Run-Time Environment

The RTE provides the infrastructure services for the communication relations which are necessary for the application and which are defined in the software component description. Thus, it is basically a realisation of the VFB. To provide an optimal implementation for a specific ECU, the RTE is generated based on configuration specifications for all software components and the ECU. Figure 2.9 depicts the methodology and the necessary configuration information for the RTE generation step [11].

The RTE handles the communication of the software components by providing an Application Programming Interface (API) to the software components. Two forms of the RTE API are available: direct and indirect. The direct API typically consists of macros which follow a naming convention based on the name of the software component and the port, permitting the unambiguous identification of API calls.

The indirect API uses indirection through a port handle to invoke RTE API functions. The generation of the indirect API is optional and only necessary, if a software component uses this form of API calls. By providing port handles, it is possible to iterate over multiple ports of the same type within a single software component. This allows a different programming style that may be more convenient in some situations.

Not only the communication is taken care of by the RTE. It also handles the scheduling of the software components. Each software component defines in the software component description one or more runnable entities. A runnable entity is a piece of code with a single entry point and associated data. The execution of runnable entities is triggered by the RTE in response to the occurrence of RTEEvents.

RTEEvents are defined in the software component description and describe events on which the software component should react. The most basic type is a TimingEvent which periodically triggers the execution of a runnable entity. More complex types of events like the DataReceivedEvent or the DataSendCompleteEvent trigger the execution of runnable entities based on the state of the underlying communication mechanisms.

2.2.4 Basic Software Layer

The Basic Software Layer is responsible for the abstraction of the hardware. It consists of several parts (as can be seen in figure 2.6) which handle different aspects of basic ECU features. For the understanding of this master thesis, only the FlexRay specific modules of the communication abstraction are important. Therefore, this description of the Basic Software Layer is limited to those modules. Similar communication stacks are also specified for the CAN and LIN bus systems.

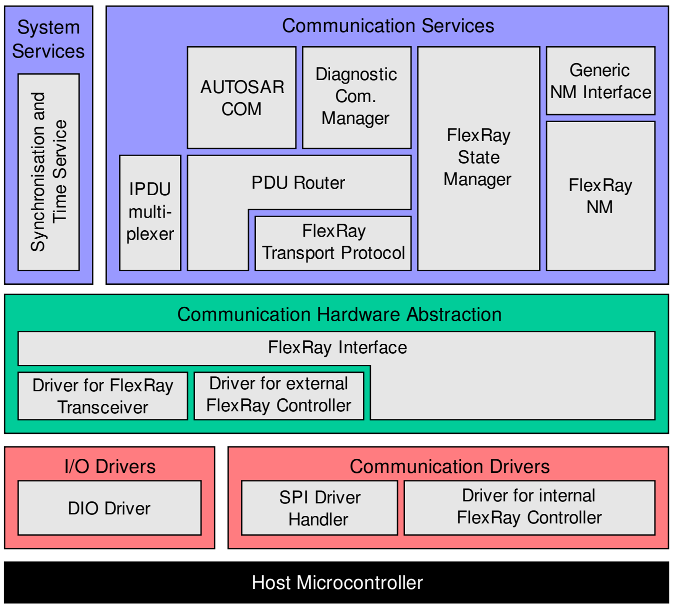


Figure 2.10: Basic Software modules of the FlexRay communication stack. [5]

In the fine grained view in figure 2.10 all modules of the FlexRay communication stack are depicted. The following subsections describe the important parts of the FlexRay communication subsystem.

2.2.4.1 FlexRay Transceiver Driver Module

The FlexRay Transceiver Driver configures the transceiver hardware. For this task, the basic Digital Input Output (DIO) driver module, which abstracts the access to the microcontroller's hardware pins, is employed. Basically the transceiver hardware can be set to four different operation modes (normal, standby, sleep and receive-only) by the Transceiver Driver. Additionally, the reception of a wakeup signal on the FlexRay bus is handled in first instance by the FlexRay Transceiver Driver. The FlexRay Transceiver Driver is specified in [10].

2.2.4.2 FlexRay Driver Module

The FlexRay Driver module is responsible for the hardware abstraction of the communication controller. Hardware details (e.g., registers, message buffers) are hidden from upper layers. Via a standardised

API Method Name	Description
Fr_ControllerInit	This method is responsible for the configuration of the communication controller hardware (e.g., cycle length, macrotick duration, buffer initialisation).
Fr_StartCommunication	A call to this API method starts the communication on the communication controller. The controller tries to integrate on the communication on the FlexRay bus.
Fr_HaltCommunication	This method instructs the communication controller to stop the communication at the end of the current FlexRay communication cycle.
Fr_TransmitTxLPdu	Upon a call of this method the data which is about to be transmitted is copied to the correct physical resource (e.g., message buffer) mapped to the given transmission parameters.
Fr_CheckTxLPduStatus	This method figures out whether data which has been scheduled before for transmission by a call to Fr_TransmitTxLPdu is still pending for transmission.
Fr_ReceiveRxLPdu	If data has arrived since the last call of this method, this method returns the data to the caller. Otherwise it is signalled, that no data has been received.

Table 2.1: Communication-related FlerxRay Driver module API methods.

interface, upper layers can invoke functions on the FlexRay Driver. According to the AUTOSAR specification, the implementation of the FlexRay Driver must be able to address up to four FlexRay communication controllers if the hardware is available and also the usage of different FlexRay Driver modules for different communication controller hardware must be possible. All details of the FlexRay Driver can be found in [8].

Table 2.1 lists some of the communication-related, important FlexRay Driver API methods.

2.2.4.3 FlexRay Interface Module

The purpose of the FlexRay Interface module is to provide an abstract interface to the upper layers which is, as far as data transmission is concerned, uniform for all bus systems supported by AUTOSAR. It accesses the communication hardware not directly, but uses the FlexRay Driver and FlexRay Transceiver Driver modules. This makes the bulk of the code of the FlexRay Interface independent of the communication controller and the transceiver. Knowledge of the underlying hardware topology is only needed when configuring the system. The FlexRay Interface module, however, has a tight relation to the FlexRay Driver module since many of the services offered to upper layers are actually carried out by the FlexRay Driver.

The communication controller hardware is not required to support asynchronous access to its transmit and receive buffers. So the calling of all functions accessing this buffers must be synchronized to the FlexRay schedule. This is achieved by defining a Job List for the FlexRay Interface at system configuration time which contains for each communication job a start time by means of FlexRay cycle and FlexRay macrotick offset and a list of communication operations. The execution of each job is then performed synchronously to the FlexRay schedule. For further details on the FlexRay Interface see [9].

2.2.4.4 FlexRay Transport Module

The FlexRay Transport module provides additional features to the upper layers. It enables the communication system to transmit data independently from the size constraints of FlexRay frames by using segmentation on the transmitter side and by collecting the data on the receiver side again. This module also provides mechanisms for controlling the data flow depending on buffer capacities of ECUs. Also a scheme for the acknowledgement and retry of data transmission is introduced in this module.

2.2.4.5 PDU Multiplexer

A Protocol Data Unit (PDU) consists of Protocol Control Information (PCI) and Service Data Unit (SDU). Multiplexing PDUs makes it possible to use the PCI of a PDU with more than one unique layout of SDU. Therefore, an additional selector field is part of the SDU which determines the used SDU layout. The PDU Multiplexer is responsible for combining multiple PDUs on the sender side and for interpreting the data based on the selector field on the receiver side.

2.2.4.6 PDU Router

The PDU Router is responsible for determining the communication mechanism which has to be used for a particular data element. For internal ECU communication, the communication is performed via shared memory, for external communication between ECUs, the PDU Router selects whether FlexRay, CAN or LIN must be used and calls the necessary methods of the transport layer of the respective communication stack. The PDU Router uses the PDU Multiplexer to combine multiple PDUs when sending data and to split the incoming data to its respective PDUs when receiving.

2.2.4.7 AUTOSAR COM

The main AUTOSAR communication module is called AUTOSAR COM. It provides a signal oriented data interface to the RTE. Only some important examples of the many functions it is responsible for are mentioned here. The AUTOSAR COM module handles the packing and unpacking of AUTOSAR signals to and from I-PDUs which are transmitted via lower level communication mechanisms. It is responsible for monitoring receive signals and notifying upper layouts about signal timeouts. Furthermore a filtering of incoming signals can be installed in the COM module. Another important functionality is the conversion of the byte order between communication system and processing platform.

2.3 Software Simulation

This section contains an overview of the state of the art of software simulation. For the simulation of the software part in hardware-software co-simulation projects, multiple approaches are possible. The palette reaches from very detailed instruction set simulation which emulates the underlying hardware to abstract functional simulation of the software which does no longer include any hardware details. The first subsections explain aspects of software simulation following the aforementioned path from very detailed to rather abstract approaches and also discuss the benefits and disadvantages of different methods. Afterwards, the scope lies on the simulation of AUTOSAR software. Similarities between SystemC and AUTOSAR are found and a approach to integrate SystemC within the AUTOSAR design process is evaluated. The next part shortly introduces an open source AUTOSAR implementation and the following part deals with the analysis of delays and timing constraint in different communication systems. Finally the existing co-simulation environment is presented. An overview of the used tools and their features and the included simulation models is given.

2.3.1 Instruction Set Simulation

The most accurate method for the simulation of software running on specific hardware is to use Instruction Set Simulators (ISSs). An ISS represents the target hardware at a very low abstraction level and includes not only a detailed description of the processor, but also of its registers and even may include models of additional peripherals. The simulation with ISSs is very time-consuming due to the high level of details.

Usually, instruction set simulation is done interpretively. Figure 2.11 depicts the control flow for this methods of simulation. At first, the software is compiled for the target architecture. The compiled code is then loaded into memory, but it is not executed. The control is given to the ISS.

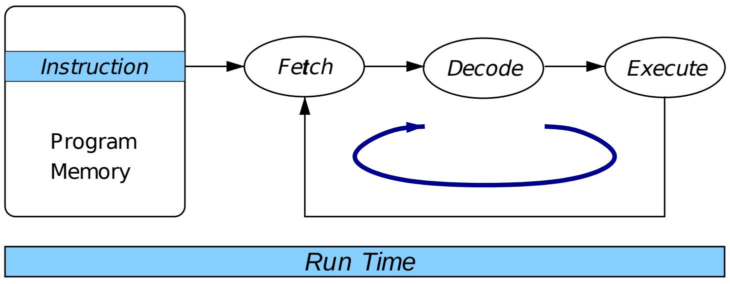


Figure 2.11: Interpretive simulation flow. [41]

The ISS fetches and parses the first instruction of the program. If the instruction is about to alter data, the ISS must ensure, that the memory is available. Afterwards, the pseudo registers of the simulated hardware must be loaded into temporary real registers, the real registers must be modified according to the instruction and stored back to the pseudo registers. Special care must also be taken about branches.

Taking into account all the necessary operations for the execution of a single instruction, it is clear, that this method of simulation contains a lot of overhead and is therefore extremely slow. On the other hand, using an ISS, the developer can view and alter registers and memory of the simulated processor and inspect in detail all performed operations.

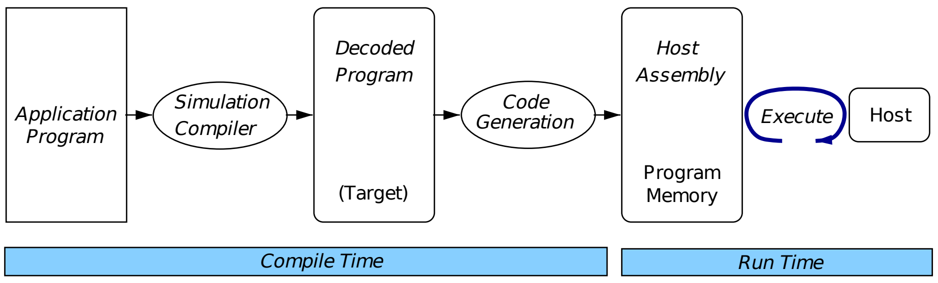


Figure 2.12: Compiled simulation flow. [41]

There exist approaches to speedup the simulation method by using a so called compiled simulation technique. The idea behind it is to shift workload from the simulation run-time to an additional step before the simulation [34]. Many of the steps which conventional instruction set simulators perform during run-time for every operation can be performed by a simulation compiler before running the simulation. This results in a significant reduction of run-time. The downside of compiled simulation is, that before simulation start, the whole schedule of the simulation has to be known in order to unroll the instructions. The method of compiled simulation is sketched in figure 2.12.

Also combinations of instructive simulation and compiled simulation are possible and have been presented, e.g., by Reshadi et al. in [41] and Braun et al. in [18].

In co-simulation environments, the hardware usually is described using description languages like VHSIC Hardware Description Language (VHDL), Verilog or SystemC. When combining these models with the software simulation, the synchronisation and timing between the two simulation components is critical. Formaggio et al. and Fummi et al. describe in [23] and [24] a method for combining an ISS with an abstract hardware model which is described with SystemC. They use the GNU Project Debugger (GDB)[2] which includes an ISS for the simulation of their software and extend the SystemC simulation kernel with methods which allow an interaction and communication with the ISS by using commands of the debugger. The extension stays completely transparent to the SystemC developer. A similar approach using an SimIT-ARM[3] ISS is described by Hau et al. in [29].

2.3.2 Abstract Functional Simulation

Not always all the details provided by the usage of an ISS are necessary. In this case, the software is executed natively on the host processor. This allows for faster simulation and increased performance. Especially SystemC is very suitable for abstract simulation of software, because it makes it possible to describe the hardware and the software components of a system in a single programming language, which makes it easy to combine and simulate both parts.

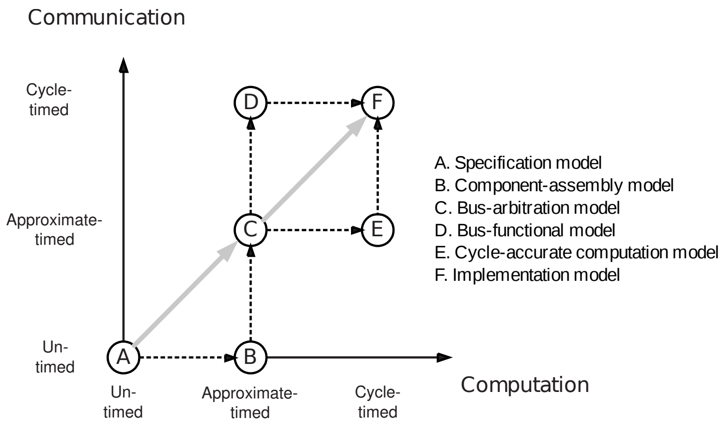


Figure 2.13: Simulation on different abstraction levels. [21]

When co-simulating hardware and software on a high abstraction level, obviously, the concepts of Transaction Level Modelling (TLM) are applicable. According to the TLM approach, the simulation of the system can be performed on different abstraction layers, depending on the simulation requirements. Cai et al. mention different abstraction levels in [21]. According to their classification there exist six abstraction layers:

- Specification model: On this layer only the functionality of the system is specified. There is no consideration of the underlying hardware. The communication between processes is accomplished with access on global variables. Therefore it is easy to port existing C or C++ code into this layer.
- Component-assembly model: This layer introduces the usage of different processing elements with memory which communicate over channels. The processing elements are approximately timed by using an estimation for the specific target hardware. Communication, however, is still untimed.

[2]http://www.gnu.org/software/gdb/

[3]http://sourceforge.net/projects/simit-arm/

- Bus-arbitration model: Similar to the component-assembly model also this layer uses processing elements, memory and channels. The channels are buses and a concept of bus arbitration and prioritisation is introduced. The communication on the bus does not follow a bus protocol.
- Bus-functional Model: Cycle-accurate timing of the buses is modelled and a bus protocol is defined on this layer.
- Cycle-accurate Computation Model: This model provides cycle-accurate timing information for the processing and an approximate timing for the communication. It can easily be generated from the bus-arbitration model by introducing accurate timing information in the processing elements.
- Implementation Model: This is the most detailed abstraction level. Cycle-accurate timing information for computation as well as for communication is included. Often this model is referenced to as register-transfer model.

Figure 2.13 depicts the different abstraction levels according to [21]. The arrows indicate a typical flow for the refinement of a system design.

There are more approaches on specifying the different layers of abstraction which are possible in transfer level modelling. A similar categorization using a different nomenclature is used by Donlin in [22] and by Tsikhanovich et al. in [45]. Because the terms of Donlin are used in later chapters, the following list gives an overview:

- Algorithmic (ALG): On this level only the functionality of the system is specified. There is no consideration of the underlying hardware. This layer is still too abstract for being part of the transfer modelling concept.
- Communicating Process (CP): The system is partitioned into parallel processes, which exchange complex high-level data structures over untimed point-to point communication links.
- Communicating Process with Time (CP+T): This level is functionally equivalent to the CP level, but here the timing of the processes is considered. The timing may be estimated or specified cycle-accurate. No timing of the communication is annotated yet.
- Programmer's View (PV): This level is much more architecture specific than the CP level. Concerning communication, bus-models are used and an arbitration of the bus-access is applied.
- Programmer's View with Time (PV+T): Functionally identical to the PV level, this level includes timing information.
- Cycle Accurate (CA): Models on this level contain a cycle-accurate timing specification. The communication is performed by bus-models which typically provide only bit-level interfaces.
- Register Transfer Level (RTL): On this level, models are described implementationally and architecturally accurate. This layer already lies below the transfer level modelling concept.

2.3.3 Simulation of Real-Time Systems

For the simulation of real-time systems, it is also necessary to consider the dynamic real-time behaviour of the embedded software. In the final implementation this behaviour is typically provided by a Real-Time Operating System (RTOS). At an early design phase, however, using a detailed, real RTOS implementation would negate the purpose of an abstract system model [26].

This section gives an overview of tools and techniques used for the simulation of RTOSs. The first subsection presents generic concepts for a RTOS model on a high abstraction level. The next subsection deals with the problem of interruptibility in the simulation of operating systems. Afterwards an approach on combining an abstract RTOS model with an ISS gaining additional speed for the operating system while retaining the accuracy of the ISS for the applications is presented. Finally a comparison of RTOS simulations on different abstraction levels illustrates the differences with respect to speed and accuracy.

2.3.3.1 Generic Concepts for an Abstract RTOS Model

Gerstlauer et al. describe in [26] an approach to specify a model of a real-time operating system at a high abstraction layer. They use SpecC for their implementation. Their RTOS model implements the original semantics of System Level Description Language (SLDL) primitives plus additional details of the RTOS behaviour on top of the SLDL core. Existing SLDL channels (e.g., semaphores) from the specification are reused by refining their internal synchronisation primitives to map to corresponding RTOS calls. The RTOS model provides four categories of services:

- Operating system management: This service type mainly deals with initialisation of the RTOS during system start.
- Task management: The task management is the most important function in the RTOS model. It includes various standard routines such as task creation, task termination and task suspension and activation. The RTOS model supports both periodic hard real time tasks with a critical deadline and non-periodic real time tasks with a fixed priority.
- Event handling: The event handing of the model is mapped directly to SpecC events. It uses the semantics of the underlying SLDL.
- Time modelling: Concerning the modelling of time within the RTOS model, the delay primitives of the underlying description language are replaced by special functions which allow for task preemption.

The cited paper contains a general overview of the RTOS model. Further implementation details can be found in a similar publication by Yu et al. [47].

2.3.3.2 Interruptibility of Tasks in Abstract RTOS Models

Posadas et al. describe in [38] the problems of interruptibility and synchronisation in an environment where multiple concurrent tasks are scheduled.

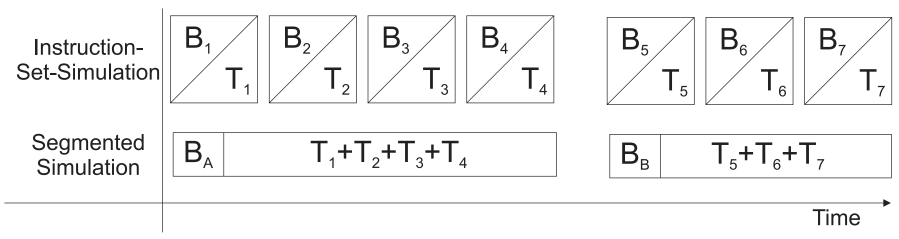


Figure 2.14: Time annotation as performed by an ISS versus code segment time annotation. [48]

In SystemC, to perform an exact timed simulation, a *wait* statement with the corresponding exact platform execution time would need to be placed next to each instruction. However, this produces very slow simulations because the SystemC scheduler runs once for each source code instruction executed. In order to improve the simulation efficiency, the time management has to be based on larger pieces of the source-code, called segments. During segment simulation, execution time can be estimated. The estimation can be performed automatically by a time manager by summing up the execution time of every operation at the time it is executed. In figure 2.14 the differences between the time annotation for each instruction (like used for instruction set simulation) and for code segments is shown.

To obtain a correct simulation, one segment has to be the fragment of code executed consecutively without being interrupted. With this definition, communication points are controlled. If only one task executes from the segment start time to the segment end, the communication order in that segment only depends on the order of communication instructions at the source code. Thus, during the segment execution, the timed model is correct because the communication instruction order is maintained.

When dealing with a set of concurrent tasks, the execution order of segments in each task has to be controlled. If simulation order is the same as in target platform execution, simulation would be correct. The result is that there is no difference between placing a *wait* statement for each instruction and placing only one *wait* at the end of the segment with the total amount of execution time.

The execution of the source code does not consume any simulated time. The time consumption is annotated with a *wait* statement after the code execution. This means hardware interruptions always arise during the waiting time. So the segment code has already been completely executed, although it should have been interrupted. This leads to incorrect system behaviour.

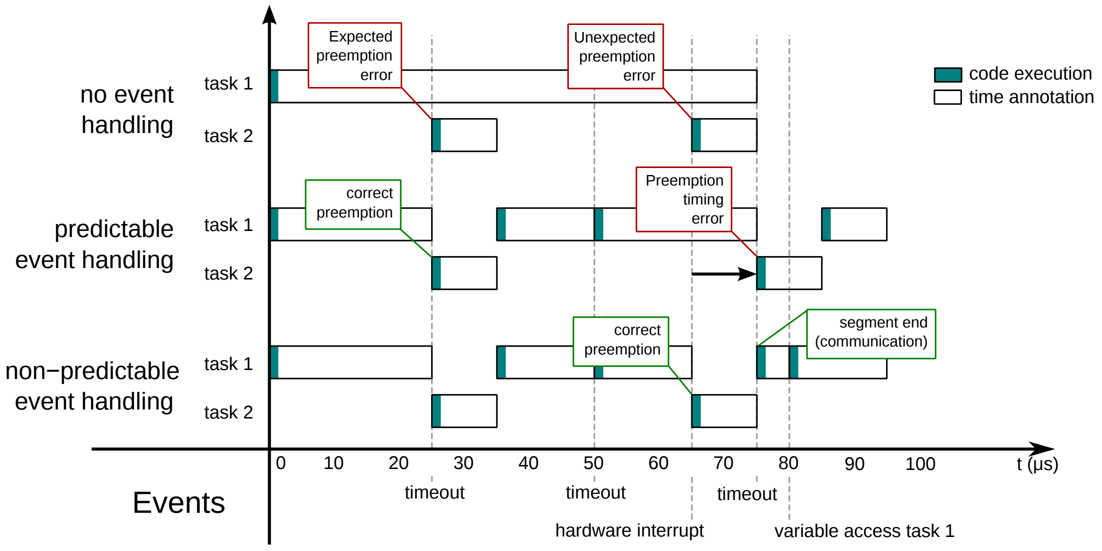


Figure 2.15: Simulation behaviour with occurrence of unpredictable interruptions. [38]

Concerning predictable interruptions this limitation can be circumvented by a workaround. At the beginning of each segment, the remaining time until the next interruption is calculated. When executing the source-code and summing up the execution time annotations, the time manager which sums up the execution time can stop the execution, when the time limit is reached.

For non-predictable interruptions, the solution is more complex. The annotated waiting time must be interruptible by an external event. In case of an interruption, the remaining time is calculated and annotated after the interrupt handling routine has completed. This solution does only solve the issue of processor assignment during this time, but if communication is done in the middle of a segment, data errors can happen. The solution for this problem is to use channels instead of global variables for communication. Each read and write access to the channel must end a segment. The actual operation is then performed at the beginning of a new segment, at the correct point in time.

The various methods of interruption handling and its influences on the correctness of the simulation are depicted in figure 2.15.

2.3.3.3 Combining Abstract Simulation and Instruction Set Simulation

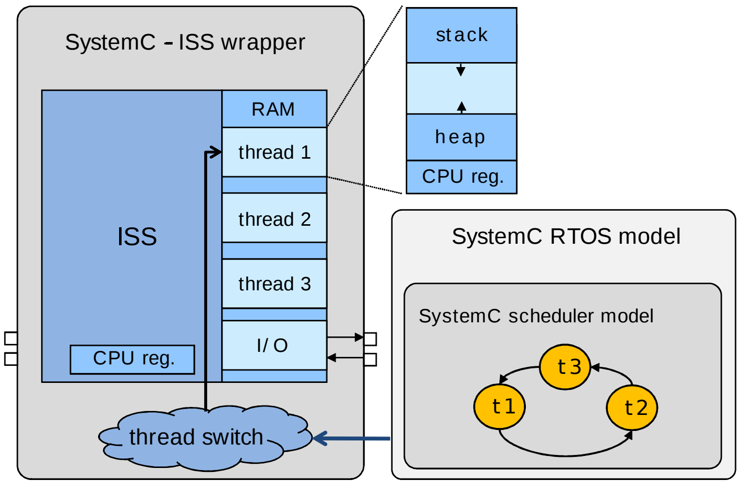


Figure 2.16: Structure of a modelling approach combining SystemC with an ISS. [33]

Krause et al. [33] extend the approach of combining an ISS with SystemC mentioned above. Since the simulation of a RTOS on an ISS can be very time consuming, they use SystemC not only to describe the hardware parts of the system but also model the operating system in SystemC. This reduces the overhead of simulating the RTOS on the ISS and increases simulation speed. For the application part of the system, the accuracy of an instruction set simulation is desired. So those parts are executed on the Simplescalar ISS[4] which is adapted and instantiated directly in SystemC. Figure 2.16 shows the structure of the modelling approach.

2.3.3.4 Comparison of RTOS Simulation on Different Abstraction Layers

Schirner et al. compare in [43] the performance and the accuracy of the simulation of an embedded application by gradually extending it with additional operating system features.

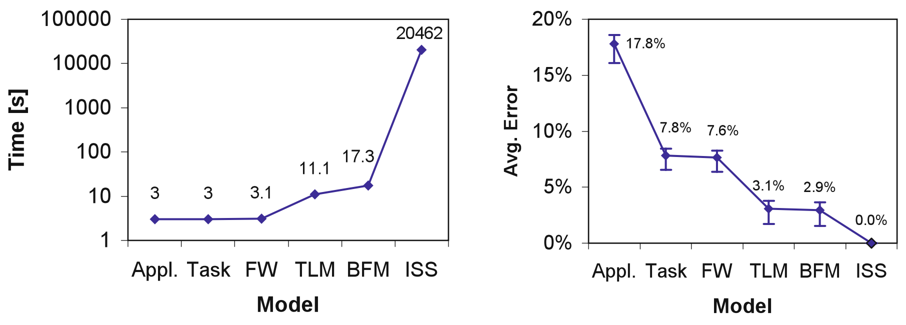


Figure 2.17: Simulation speed and accuracy at different abstraction layers. [43]

Figure 2.17 shows the results of their comparison. For better understanding of the used abbreviations, the following list gives an overview of the levels they used and also includes their abbreviations:

- Application (Appl.): At the highest level, the application is running natively on the simulation host.

[4]http://www.simplescalar.com

- Task (Task): At the Task level, an abstract RTOS model is introduced, and processes and communication channels are refined into tasks and inter process communication primitives running on top of the RTOS model.
- Firmware (FW): The Firmware model adds the features of interrupt handling and low level software drivers.
- Bus-Functional Model (BFM): A Bus-Functional model of the processor includes pin- and cycle-accurate models of the bus interfaces and the protocol state machines driving and sampling the external wires.

2.3.4 Concepts for the Simulation of AUTOSAR Software

Approaches on the simulation of AUTOSAR software were already made. This section contains an outline of the proposals. The first idea exploits the similarities of AUTOSAR and SystemC for modelling and a second one deals with the integration of a simulation model into an AUTOSAR toolchain.

2.3.4.1 SystemC within the AUTOSAR Design Process

AUTOSAR provides a methodology for the design of automotive software. However, AUTOSAR does not pay attention to the simulation and analysis of the developed implementations. SystemC on the contrary comes with a simulation kernel and allows the simulation of software and even the analysis of timing behaviour of the underlying hardware and communication paths.

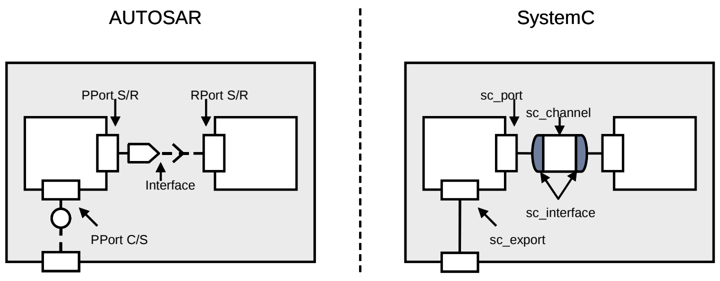


Figure 2.18: Port analogies between AUTOSAR and SystemC. [32]

Despite the different goals, there are many similarities with respect to the modelling structure between the concepts of SystemC and AUTOSAR [32]. Both have behavioural entities which can form hierarchies. The software components of AUTOSAR can be mapped to SystemC modules. Also regarding communication, there exist analogies. AUTOSAR as well as SystemC provide definitions of ports which are used for the exchange of data between processing elements or software components. They both support to export ports throughout the entire component hierarchy. Figure 2.18 illustrates the analogies of the port types between AUTOSAR and SystemC. Concerning scheduling, they both support entities which can wait for events and be scheduled via events.

A methodology for using of SystemC in the AUTOSAR design process is presented by Krause et al. in [32]. Figure 2.19 depicts the basic concept. The AUTOSAR software components are ported to SystemC, exploiting the analogies mentioned above. With SystemC the behaviour of the system is simulated and analysed and the insights influence the AUTOSAR configuration files.

Also, in the AUTOSAR methodology there exist different views, which can be compared to abstraction levels. The application on VFB view, independent of a particular infrastructure and mapping onto ECUs is similar to the SystemC CP (Communicating Process; see section 2.3.2) level. As a consequence, an AUTOSAR design can be transformed into an equivalent SystemC design at CP level.

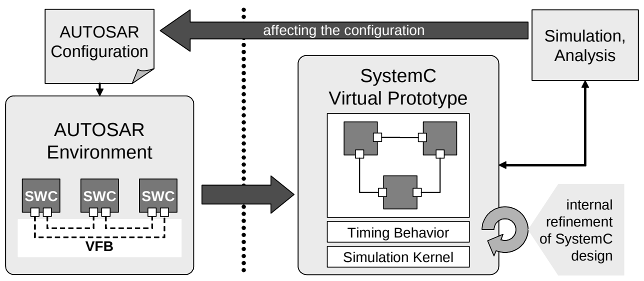
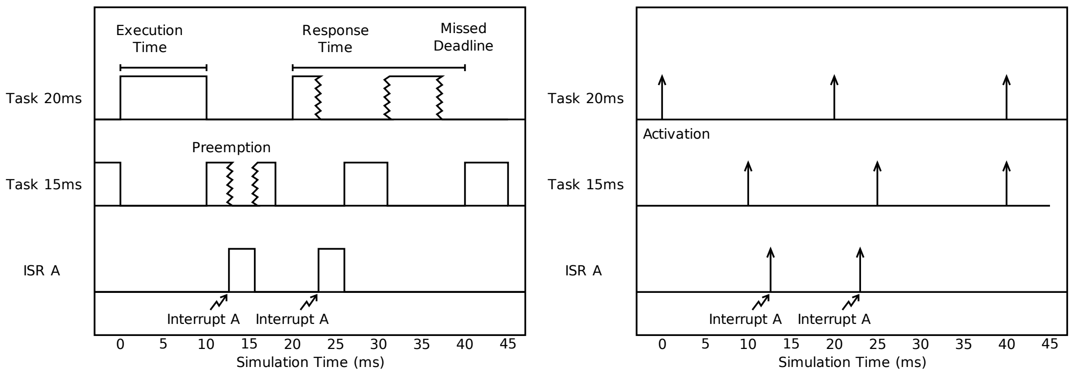


Figure 2.19: Methodology for using SystemC in the AUTOSAR design process. [32]

When the software components get mapped to ECUs, this relates to a SystemC model on PV level. From this type of model, which is still untimed, the SystemC model can be refined to PV+T or even CA if desired by annotating the timing information in the SystemC code.

2.3.4.2 Integration of RTOS Simulation into an AUTOSAR Toolchain

Becker et al. describe in [17] a method for integrating the simulation of a RTOS into an AUTOSAR toolchain. They implemented a RTOS with SystemC based on an earlier model by Gerstlauer et al. [26]. Concerning interruptibility of the time intervals, they also introduced concepts by Posadas et al. [38] which were already described above in section 2.3.3.2. By using a separate scheduling model for tasks and interrupt handlers their model gains additional precision [48].



(a) Timed simulation.

(b) Untimed simulation.

Figure 2.20: Difference between timed and untimed simulation. [17]

The tool SystemDesk from dSPACE is an Integrated Development Environment (IDE) for the development of AUTOSAR compatible automotive software which includes an offline simulator. The included simulator supports untimed simulation only. This means the execution times of the processes are not considered and they return immediately. The difference between timed and untimed analysis is depicted in figure 2.20. By taking a subset of the AUTOSAR features and mapping them on the RTOS model, Becker et al. were able to integrate their model into the simulator of SystemDesk. With this additional timing information, they were able to perform detailed analysis of the timings, performance and the influence of the operating system on the application.

2.3.5 Arctic Core - An Open Source AUTOSAR Implementation Project

Also the open source community has made efforts on the development of AUTOSAR software. Since June 2009, an open source implementation of the AUTOSAR Basic Software is in progress by a team supported by the Swedish company ARCCore[5]. The business model is to provide the AUTOSAR implementation free of charge and to offer commercial plug-ins to their Eclipse based IDE.

The development is still in an early phase. While there are other communication systems already supported or planned (e.g., LIN, CAN) the FlexRay specific parts of the Basic Software are neither implemented nor are they planned for the near future. Furthermore, the open source implementation of the Basic Software is only runnable on compatible hardware, limiting the usability for inclusion in the co-simulation environment.

A query regarding the features of the configuration plug-ins was sent to the developer team in order to estimate their usability and value for facilitating the necessary configuration for the implementation presented in this thesis. The configuration plug-ins are still in a beta phase and no detailed information about their features was replied. So also the option of using the configuration plug-ins is not available at the moment.

2.4 Delay and Timing Analysis of Communication Systems

Delay and timing issues of the communication paths play a very important role for dependable architectures. While event-triggered communication systems provide no guarantees for timings at all, time triggered systems like FlexRay ensure that timing constraints can be met at communication protocol level. However, at a higher level additional factors like scheduling, blocking or buffering influence the delay, leading to hard-to-find timing problems like transient overload, buffer under- and over-flows or missed deadlines.

Analyses of the end-to-end delay are performed by Racu et al. in [40]. Using a network simulation tool, they are able to analyse complex heterogeneous automotive systems with heterogeneous networks. With a case study they illustrate the influence of design decisions on the sensitivity and robustness of the communication network. However in this paper the focus lies on the CAN system and it provides no details concerning the FlexRay network.

The timing in the FlexRay protocol is analysed by Pop et al. in [37]. Using a specialised real-time operating system which provides a static cyclic scheduling algorithm for tasks communicating in the static segment of the FlexRay protocol and a fixed priority scheduling algorithm for additional tasks using the dynamic segment, they are able to perform a worst-case response time analysis and identify the timing properties for all the tasks and messages in the system. A heuristic approach is presented to optimise bus access parameters and with experiments they show how optimisation of the schedule to the requirements of the application makes potentially unscheduleable solutions scheduleable.

The FlexRay analysis is extended by introducing a simple AUTOSAR layer by Guermazi et al. in [27]. With a formal approach they show how a system architecture relying on the FlexRay protocol and using the AUTOSAR communication model for managing PDUs permits a precise characterisation of the worst-case end-to-end delay.

The specification of timing and performance parameters are a major challenge for system integration and standardisation would help finding problems and optimised solutions. However, even the AUTOSAR specification provides no standardised way of specifying timing constraints [42]. An informal chapter in the VFB specification [12] provides approaches for timing specifications, but this chapter is not officially part of the AUTOSAR standard. This illustrates the difficulties in finding a common ground among the heterogeneous pieces of automotive environments.

[5]http://arccore.com/

Chapter 3

Design of the VFB Abstraction Layer

This chapter contains a concept description for the developed VFB abstraction layer. The VFB abstraction enhances an existing FlexRay co-simulation with the possibility of integrating application software resulting in an holistic simulation of an entire automotive application. The application can be developed and simulated on an high abstraction level while still being able to inspect or modify the low level communication hardware. This allows to analyse the influences of changes of the low level system parameters, like the FlexRay configuration or the network topology, as well as influences of the choice of the application architecture, like the distribution of software components among different ECUs, on the functionality and performance of the application.

The purpose of this chapter is to illustrate the design decisions and concepts for the implementation of the VFB abstraction layer. At first, the existing co-simulation environment, including used tools as well as already developed models, is presented. In the following section, the idea of using code generation tools for the implementation is evaluated. Afterwards some use cases for the final system are sketched. An analysis of the use cases leads to requirements on the implementation which are listed in the following section. The last sections provide insight into the concept used for the realisation of the VFB abstraction layer. It is discussed why an object oriented approach was used, how the system is configured and the concept for interfacing the communication system is explained. Finally the approach of reserving and accessing memory for intermediately saving the communication data within the VFB abstraction layer is described.

3.1 Co-Simulation Environment

The co-simulation approach used for the the TEODACS project is a heterogeneous environment. It allows the usage of different languages for different parts of the system. This approach allows to utilise the benefits of different languages and to integrate speed-optimised versions of some models with high-accuracy versions of other parts of the system. Further information on the approach used for the co-simulation environment and additional details can be found in [31] and [30].

In this chapter a survey of the features of the third party tools of the simulation environment which were used for the development of the VFB abstraction layer is given. Furthermore the models of the simulation environment which do already exist or are currently under development are presented.

3.1.1 Used Tools in the Co-Simulation Environment

For the realisation of the simulation environment the employment of third party tools is required. In this section some of the used tools which are relevant for the understanding of this thesis are presented and a survey of their features is given.

3.1.1.1 SystemC

SystemC is an open source extension to the programming language C++. It introduces a concept of parallelism and time and therefore allows the description, simulation and synthesis of hardware modules [3]. For system synthesis, the code of the module must be low level because, similar to other hardware description languages like VHDL or Verilog, only low level (e.g., register-transfer level, behavioural level) code can be synthesised.

The benefits of SystemC are the ability to describe hardware and software components in a single programming language. The simulation speed of models developed with SystemC on a high abstraction layer is rather high. Depending on the purpose individual modules of a system can be modelled in greater or less detail providing fine grained control of the simulation time.

3.1.1.2 System Architect Designer

For the network simulation the tool System Architect Designer (SyAD) is used. It is a simulation environment for system design of heterogeneous microelectronic systems provided by CISC Semiconductor Design+Consulting GmbH[1] and features the simultaneous usage of various simulation languages (e.g., SystemC, VHDL-AMS) allowing for a great amount of flexibility. The overhead of translating models into a simulation environment specific language is saved.

3.1.1.3 CarMaker / AVL InMotion

CarMaker / AVL InMotion is a tool targeted at the simulation of vehicle dynamics (including models for power train, driver, track, control unit functions, brakes, etc.). It is the output of an intensive cooperation between simulation specialist IPG[2] and the expert for test bench technology AVL[3]. CarMaker / AVL InMotion is suited for the simulation on a PC and for hardware-in-the-loop tests on component and large system test benches. Within the TEODACS project, both variants are used. In the laboratory setup a dedicated CarMaker / AVL InMotion hardware node with an interface to the FlexRay bus is installed and in the simulation environment an interface to the CarMaker / AVL InMotion simulator is included.

3.1.2 Overview of Already Existing Simulation Models

For the simulation environment numerous models have already been developed or are currently under development. While most of them were developed at the Institute for Technical Informatics of the Graz University of Technology, some are contributed by project partners. All of the models are integrated in the SyAD simulation tool. They allow forming and simulating complex communication networks. The following list should give an overview of the available models.

- Cable: Models for the simulation of different cable topologies are available. For faster simulation times SystemC models can be used and for a greater amount of detail also VHDL-AMS models were implemented.
- Active Star: To allow the simulation of advanced FlexRay network topologies, a VHDL-AMS model of an active star is currently under development.
- Transceiver: For the FlexRay transceiver also two models exist. A fast SystemC model and a detailed VHDL-AMS model.

[1] http://www.cisc.at/
[2] http://www.ipg.de/carmaker.html
[3] http://www.avl.com/

- Communication Controller: A communication controller has been implemented in SystemC. It is based on the version 2.1 of the FlexRay protocol specification [2].
- CarMaker / AVL InMotion Interface: To enable communication with CarMaker / AVL InMotion, a SystemC module is available which interfaces SyAD as well as CarMaker / AVL InMotion.

In figure 3.1 an overview of the TEODACS concept for the co-simulation environment combining the various simulation models is illustrated and the interaction to the vehicle simulator is shown.

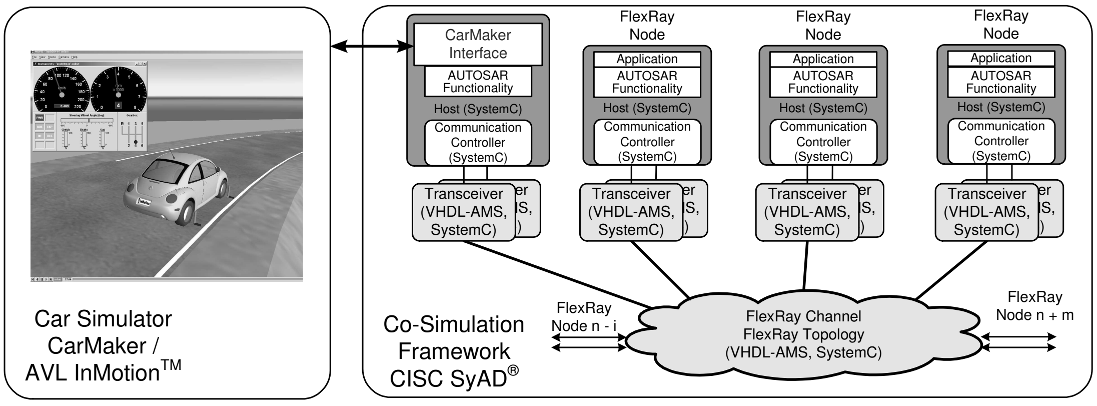


Figure 3.1: Concept of the TEODACS co-simulation platform. [30]

Further details on the co-simulation environment and the used tools and included models as well as a general description of the motivation and goal of the TEODACS project can be found in [16] and [15].

3.2 Evaluation of a Tool-Generated RTE

The AUTOSAR methodology suggests that the interface between the basic software and the application is generated from the software component descriptions. Therefore, an idea was to use an AUTOSAR generation tool and design the implementation in a way that the generated code could be integrated.

In order to to evaluate this method the major tool suppliers in the field of AUTOSAR development dSPACE[4], Elektrobit[5] and Vector[6] were contacted. There was no response from Vector and dSPACE. After enquiry of the ViF at a contact at Electrobit, they offered a free time-limited trial license for their tool EB tresos Studio.

The tool offers great support for the configuration of the AUTOSAR Basic Software. The configuration of all Basic Software modules, including the FlexRay schedule and a mapping of signals to frames, is possible in a clear way. However, concerning the configuration of the VFB, the EB tresos Studio does only provide the possibility to import the description from an existing AUTOSAR system description file. For this work, also the configuration of the VFB is of great importance.

The code generation features of EB tresos Studio are extensive. The code generation is performed by so-called generation modules. Along with the distribution of EB tresos Studio came a set of generation modules which generated code optimised for a special hardware platform. The generated code is highly dependant on libraries of the generation modules which are in turn written for a special hardware.

[4]http://www.dspace.com
[5]http://www.elektrobit.com
[6]http://www.vector.com

Furthermore, the generated code consists mainly of configuration parameters which are realised as preprocessor macros. Both aspects make it hardly possible to integrate the generated code into the existing simulation environment.

Due to the aforementioned limitations it was decided to abandon the option to generate the code. Instead the possibility of an easy and straightforward system configuration for the application developer was emphasised.

3.3 Use cases

This section gives short descriptions for some of the use cases how the VFB abstraction layer can be used by the developers of AUTOSAR applications.

3.3.1 Usage of the Communication System

The application developer is able to easily access the models of the communication hardware. He does not need to care about hardware details and can send and receive data to/from other software components easily. This use case is not limited to the transmission of data, it is also extended to the execution of remote operations according to the AUTOSAR client-server communication. For the application developer it makes no difference whether the software components are running on one ECU or are located on different ECUs.

3.3.2 Efficient Configuration of the System

All software components communicate only via the VFB. The application developer is able to specify the ports, data elements and operations responsible for the communication in an easy and efficient manner. Additionally the configuration of the underlying communication system can be efficiently integrated.

3.3.3 Reaction to Events

The application developer is able do define runnable entities in a straightforward way. A runnable entity is a piece of code which gets executed at specific events. Those events include events of the communication system as well as periodic events and time-outs. The assignment of events to runnable entities is done by the developer of the software component in connection with the definition of runnable entities. Whenever an event which is assigned to a runnable entity occurs, the system executes the respective runnable entity.

3.4 Requirement Analysis

In this section the requirements on the system architecture are described. The requirements are imposed partly by the use cases specified above in section 3.3 but do also include non-functional requirements.

As already mentioned in section 1.3, a toolchain for the system configuration is already in use within the TEODACS project. As this implementation of the VFB abstraction layer is closely related to the already existing toolchain and must integrate efficiently into the existing environment, some constraints on the implementation are given.

One of those constraints is concerning the choice of the programming language for the implementation. A FlexRay communication controller model, which is used for the lower level communication of the AUTOSAR abstraction layer, is already implemented in SystemC. For easy interfacing to this model, SystemC was chosen as language for the VFB abstraction layer too. In the current specification, AUTOSAR requires the implementation language of a software component to be C or C++ [11]. As SystemC is an extension to C++, the usage of SystemC allows for easy cooperation of the software components and the VFB abstraction layer.

For the laboratory setup, tools are available to extract the configuration data from a description of the communication configuration in Field Bus Exchange Format (FIBEX). In order to compare the results of the co-simulation environment with the results of the laboratory setup, a close relation between both approaches must exist. Therefore it is desired that the same configuration files can be used in the laboratory setup as well as in the co-simulation environment. They define the communication parameters of the whole FlexRay communication cluster including a communication schedule. Within this configuration, there is also a mapping between signal names and frames defined, which allows the transmission of multiple signals in a single FlexRay frame. This mode of operation should also be possible in the VFB abstraction layer, enabling it to transmit multiple data elements of a port in a single frame.

Another set of constraints is given by the AUTOSAR specification. The conventions for the names and function prototypes of the methods accessing the communication data are defined in the AUTOSAR specification. The implementation should adhere as closely as possible to these definitions in order to simplify the porting of software components which were written for AUTOSAR compliant environments to the co-simulation environment. Therefore the function prototypes for the communication via VFB ports are given.

The configuration of the VFB abstraction layer should be as simple as possible. This is necessary at the moment for an easy manual editing as well as in a next step for being able to develop or adapt tools which extract the VFB configuration from an AUTOSAR system description file to generate the necessary header files for interfacing the VFB abstraction layer.

3.5 Design Decisions for the System Architecture

AUTOSAR specifies in great detail the VFB and the underlying modules of the Basic Software which are necessary to provide the required functionality. The modules required for the communication via FlexRay were already briefly described in section 2.2.4. However, a complete implementation of these modules is not possible in the context of a diploma thesis. So a concept was developed reducing the implementation effort and including only the essential parts of the AUTOSAR communication stack. Figure 3.2 depicts the AUTOSAR layers and highlights which parts of the FlexRay communication stack were considered relevant for the implemented VFB abstraction layer. Only a subset of the features mentioned in the AUTOSAR specification is realised. The complex module structure is ignored and also the relations to communication irrelevant modules of the Basic Software (e.g., onboard device abstraction, memory hardware abstraction, system diagnosis services) are ignored where possible.

An object oriented approach is used for the realisation of the VFB abstraction layer. This approach is also used for the implementation of functionalities of software components and is different to the concept mentioned in the AUTOSAR specification. A software component in terms of AUTOSAR is a number of global functions and variables. They are generated by the RTE generator and have names which are unique on the ECU for which they are generated. In the simulation environment, however, the code of all software components of all ECUs is linked into a single executable file. So, the AUTOSAR approach could lead to naming conflicts and multiply defined symbols.

As solution used in this implementation, all code for a software component is encapsulated in a class which is a descendant of a common basic *SimulationSoftwareComponent* class, as can be seen in figure 3.3. This allows a software component to have its own namespace and avoids naming conflicts. An additional benefit of this method is that multiple instances of the same software component are no hassle. The problem of keeping track of instances by using RTE_Instance references as defined in the AUTOSAR specification [11] is solved by simply instantiating multiple objects of the same class. Furthermore, as the

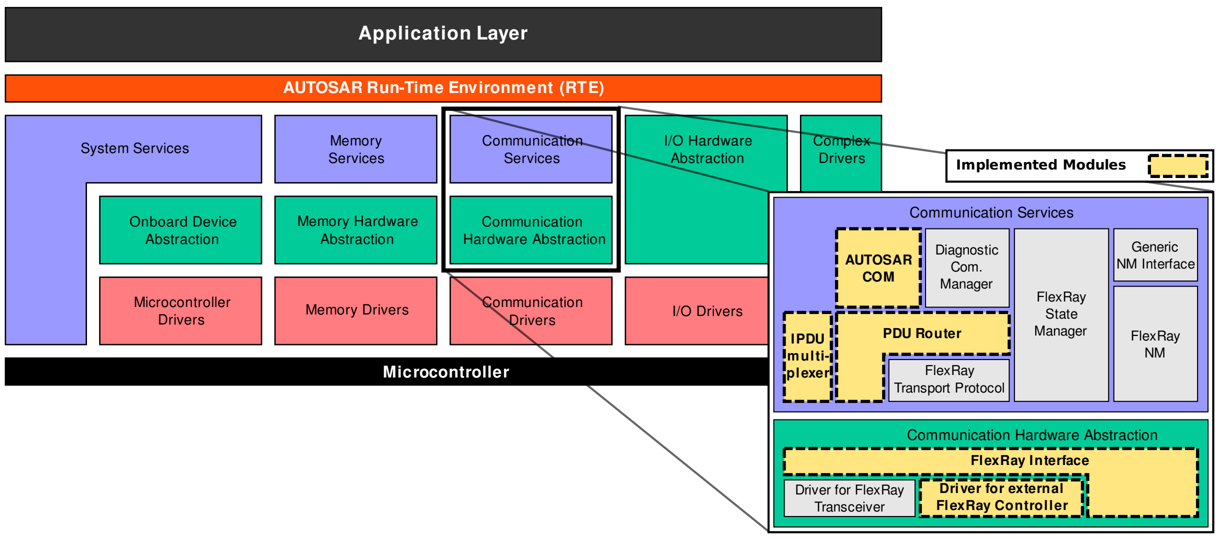


Figure 3.2: Implemented modules of the AUTOSAR architecture.

base class *SimulationSoftwareComponent* is a module in SystemC terms, the Graphical User Interface (GUI) of the used simulation software can provide a clear view of the assignment of software components to ECUs.

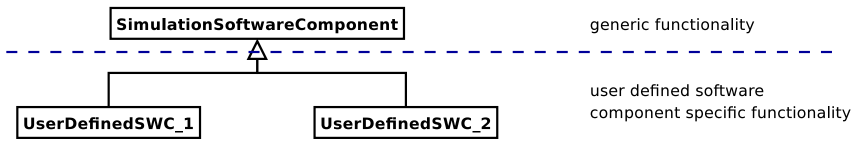

SimulationSoftwareComponent
generic functionality
UserDefinedSWC_1
UserDefinedSWC_2
user defined software component specific functionality

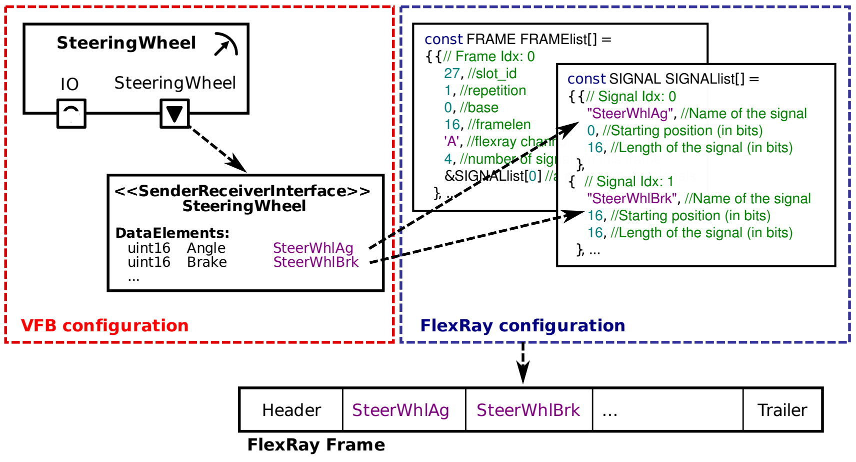
Figure 3.3: Class diagram of the object oriented concept for software component development.

3.6 Specification of the System Configuration

The configuration concerning the FlexRay parameters is given in a separate file which can be generated by extracting the information of a FIBEX file with tools developed at the ViF. This file defines the FlexRay schedule and a list of signals and a construction plan for the FlexRay frames.

For the VFB a configuration of ports, data elements and operations is necessary. This configuration is specified directly in the software component (see section 4.2.2). For the transmission of data via the FlexRay network, a mapping between the data elements and operations defined in a software component and the signals defined in the FlexRay configuration is necessary. The mapping is also specified directly in the software component. A schematic view of the mapping of VFB configuration and FlexRay signals can be found in figure 3.4.

Whenever a defined data element or operation is mapped to a signal name which is defined in the FlexRay configuration too, it is treated as remote data. Remote data is transmitted via the FlexRay network to other ECUs. In addition to the transmission via the communication system, the data must also be distributed locally to software components running on the same ECU to be conform to the AUTOSAR specification. AUTOSAR requires that from the view of the software component there should be no difference whether the communication is done via a communication system or locally via shared memory. Furthermore, the standard states that two software components that are executed on the same ECU are under no circumstances allowed to communicate by other means than via the defined ports using the VFB.



```
const FRAME FRAMElist[] =
{{// Frame Idx: 0
    27, //slot_id
    1, //repetition
    0, //base
    16, //framelen
    'A', //flexray chan
    4, //number of sig
    &SIGNALlist[0] //a
  }, ...
```

```
const SIGNAL SIGNALlist[] =
{{// Signal Idx: 0
    "SteerWhlAg", //Name of the signal
    0, //Starting position (in bits)
    16, //Length of the signal (in bits)
  },
 {  // Signal Idx: 1
    "SteerWhlBrk", //Name of the signal
    16, //Starting position (in bits)
    16, //Length of the signal (in bits)
  }, ...
```

Figure 3.4: Scheme for the mapping of data elements to FlexRay signals.

3.7 Communication on the VFB

According to AUTOSAR, the communication on the VFB is done by calls of RTE API methods [11]. Generic versions of the methods are included in the *SimulationSoftwareComponent*. According to the VFB configuration, specific RTE API methods are available in the user defined software components. The naming scheme follows the AUTOSAR specification. They are essentially aliases to the generic methods provided by the *SimulationSoftwareComponent* which contain the actual functionality. The class diagram in figure 3.5 displays the relation between the basic *SimulationSoftwareComponent* and the user defined software components with respect to the RTE API for the communication on the VFB.

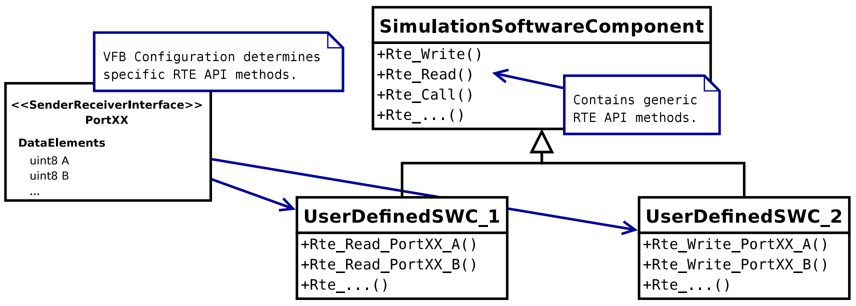


Figure 3.5: Relationship of software component classes with respect to the RTE API methods used for communication on the VFB. [30]

3.8 Interfacing the Communication System

For the communication via FlexRay, a model of a communication controller is used. To enable multiple software components to use a single communication controller, an additional *Ecu* module is used. The software components as well as a communication controller are connected to the *Ecu* module. Therefore, the *Ecu* acts as connection point between the communication controller and the software components.

The *Ecu* module takes care of the communication. It initialises the communication controller and handles the communication between the software components — local communication as well as remote communication. The block diagram in figure 3.6 illustrates this collaboration.

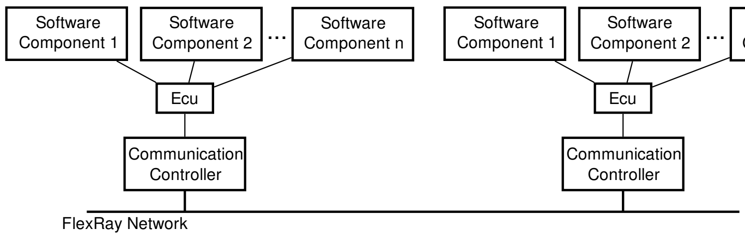


Figure 3.6: Block diagram of different software components interfacing a communication controller via the *Ecu* module.

The connection between the *Ecu* module and the software components and the communication controller respectively is accomplished by using SystemC ports and two SystemC interfaces which get implemented by the *SimulationSoftwareComponent* and *CommunicationController* class respectively. A class diagram showing this relation is depicted in figure 3.7.

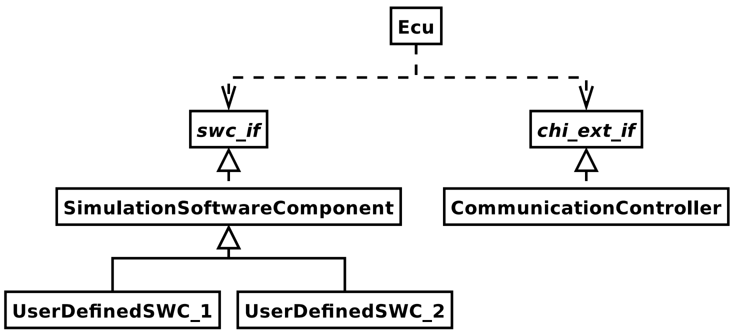
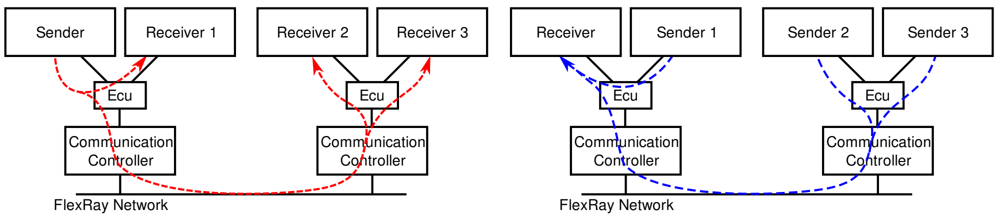


Figure 3.7: Interfaces used for realising the connection between *Ecu*, *CommunicationController* and software components.

Special care must be taken for setups with multiple senders or multiple receivers. Possible scenarios are depicted in figure 3.8. It might happen that data from a sender port of a software component is required by multiple receiving ports which may be part of software components located on the same *Ecu* or on remote *Ecus*. Therefore, the sent data must be distributed locally as well as forwarded to the communication system. A similar scenario arises when a receiving port of a software component requires data from sending ports of multiple software components. They might as well be distributed to different *Ecus* including the local one. So the *Ecu* must handle local communication requests as well as react to incoming data from the communication system.

3.9 Intermediate Saving of Communication Data

For local communication as well as for communication via the communication system, the communication data has to be intermediately saved. The following example illustrates the reasons. For local communication a runnable entity of a sending software component wants to send data to a receiving software component. An arbitrary time later a runnable entity of the receiving software component wants to access the data from the sending software component. The runnable entity may, however, have already finished its job and deallocated the memory of the variable it was sending. So the value has to be



(a) Sending data to multiple receivers. (b) Receiving data from multiple senders.

Figure 3.8: Communication paths among distributed senders and receivers.

intermediately saved by the VFB abstraction layer to allow the receiving software component to access it.

A similar problem arises for communication via the communication system. Here, the sending runnable wants to send its data. But it is not ensured that the communication is ready for transmission of data at this moment. So also in this case the data has to be saved until the communication system is ready. Also on the receiving side, the data has to be saved because the execution of the receiving runnable entity can be asynchronous to the communication system.

The *SimulationSoftwareComponent* contains objects for saving the data and methods to efficiently access the saved data. As there are two different methods of communication, the two different classes *DataElement* and *Operation* are used for intermediately saving communication data. The class *DataElement* takes care of data following the sender-receiver communication model, the class *Operation* takes care of client-server communication data.

For accessing the data, two different approaches are possible. If the data is accessed by a software component, different information about the data to access is available than if it is accessed by the communication system. The different data access methods are described in the following subsections.

3.9.1 Accessing Communication Data from the Software Component

When accessing communication data from a software component to send or receive data via the VFB, a specific RTE API method is called. With this call, the information about the port name and data element name (in case the port uses a sender-receiver interface) or operation name (in case the port uses a client-server interface) is given. The port name is unique for a software component and the data element and operation name is unique for a port. Therefore, the communication data is found in this case by using the port and data element or operation name respectively.

3.9.2 Accessing Communication Data from the Communication System

A different approach is necessary when the communication data is accessed by the communication system. The available information in this case is primarily a FlexRay cycle and slot number. This information can be translated to an index in the FlexRay configuration. The index uniquely represents a frame in the schedule. The communication data is found in this case by using this frame index.

For local communication, neither port name nor data element name of the receiver are known. The only information in this case is a signal name. A local signal name must be unique on a single *Ecu*, so a mapping of signal name to communication data is possible.

Chapter 4

Implementation of the VFB Abstraction Layer

This chapter focuses on details of the implementation. It provides further refinement of the rather abstract concept description already given above. The implementation provides only a subset of the AUTOSAR features. Therefore, the first section explains implementation specific parts and shows some differences to the AUTOSAR specification. As the system configuration is an integral part of the application programming task, the next section of this chapter deals with the configuration of the system. It is specified what parameters are configurable and how the configuration is done. The following section explains the object oriented approach used for the implementation. Details for the implemented classes are given. Finally the last section covers aspects concerning the collaboration between different classes and provides information about the timing related behaviour of the system.

4.1 Implementation Specific Aspects of the RTE

The focus of this thesis lies on providing communication services between software components. So the effort was concentrated on the software component communication related parts of the RTE and not the complete AUTOSAR RTE was implemented. The feature of allowing mode switches of an ECU was discarded and also the support for calibration components was skipped for the sake of reducing complexity and implementation effort. This results in not all AUTOSAR defined events being supported and not all RTE API methods being available. The purpose of this section is to give an overview of the implemented parts of the RTE.

4.1.1 Event Handling

In this implementation not all events defined by AUTOSAR are included. A list of events on which software components can react is given in table 4.1. The event names as defined by AUTOSAR are used.

Event	**Description**
TimingEvent	This event is raised periodically. It is used to trigger the execution of periodic tasks.
DataReceivedEvent	This event is raised when a data element is received. It is available for all data elements of a receiver port.
DataReceivedErrorEvent	This event is raised when an error occurs when receiving a data element. It is available for all data elements of a receiver port.

DataSendCompletedEvent	This event is raised when a data element has been sent. It is available for all data elements of a sender port.
OperationInvokedEvent	This event is raised when a operation on a server is called by a client. It is available for every operation of a server port.
AsynchronousServerCallReturnsEvent	This event is raised when an asynchronous server call is finished. It is available for all operations of a client port.

Table 4.1: Implemented AUTOSAR events.

The rising of events is performed in the data handling objects *DataElement* and *Operation*, described in detail in section 4.3.8 and 4.3.10 respectively.

4.1.2 Implementation of RTE Methods

The user defined software components of the application developers need to access the underlying communication mechanisms. According to AUTOSAR this is done via calls of RTE API functions [11]. Those functions are implemented in the *SimulationSoftwareComponent* class and are also accessible in the user defined software component, as it is a descendant. The available methods are specified in the VFB configuration as mentioned in section 4.2.2. For the sake of simplicity, only calls to the indirect port API are possible. The following table lists the implemented RTE API functions and a short description of their purposes.

RTE API Method	**Description**
Rte_Send, Rte_Write	Initiate a sender-receiver transmission of data elements. AUTOSAR differentiates between queued and unqueued data elements. For queued data elements Rte_Send is used and for unqueued data elements, Rte_Write is used. In this implementation, both methods are identical and can be used for both types of data elements.
Rte_Receive, Rte_Read	Performs a read on a sender-receiver communication data element. Again no difference is made between queued and unqueued data elements. Rte_-Receive and Rte_Read are equivalent.
Rte_Call	Initiate a client-server communication. This method is available for all defined client ports.
Rte_Result	Get the result of an asynchronous client-server call.

Table 4.2: Implemented AUTOSAR RTE API methods.

4.2 Details on the System Configuration

The system configuration can be split up into two parts which are closely related to each other. The first part concerns the configuration of the communication system and the second one concerns the configuration of the VFB.

4.2.1 FlexRay Configuration

The configuration of the FlexRay parameters is given by a header file which contains constants storing the configuration parameters. There are multiple constants defined. Their purpose is explained in detail below.

SIGNALlist: This array defines the transmitted signals. Every signal has an arbitrary chosen name for being able to identify it, a length and a starting position which indicates the offset of the signal in a frame to the beginning of the payload.

FRAMElist: This array is used for the specification of the transmitted frames. Every frame has a channel identifier a slot index and a base and repetition value for cycle/slot multiplexing (see 4.2.1.1). Furthermore, the frame contains a reference to a signal from the *SIGNALlist* and the number of signals which are contained in this frame allowing a construction of the frame payload from signal data.

ECUlist: This array contains one entry per used *Ecu*. It specifies the node specific FlexRay configuration. In addition, concerning the schedule, a reference to a frame from the *FRAMElist* and a number indicating the number of frames which are transmitted by the *Ecu* is given.

testcluster: This constant contains the specification of the global FlexRay configuration parameters. Additionally an index in the *ECUlist* and a value for the number of *Ecus* configured in the FlexRay cluster is contained.

The configuration can be extracted from a FIBEX file. An overview of the structure of the related classes is given in the class diagram in figure 4.1.

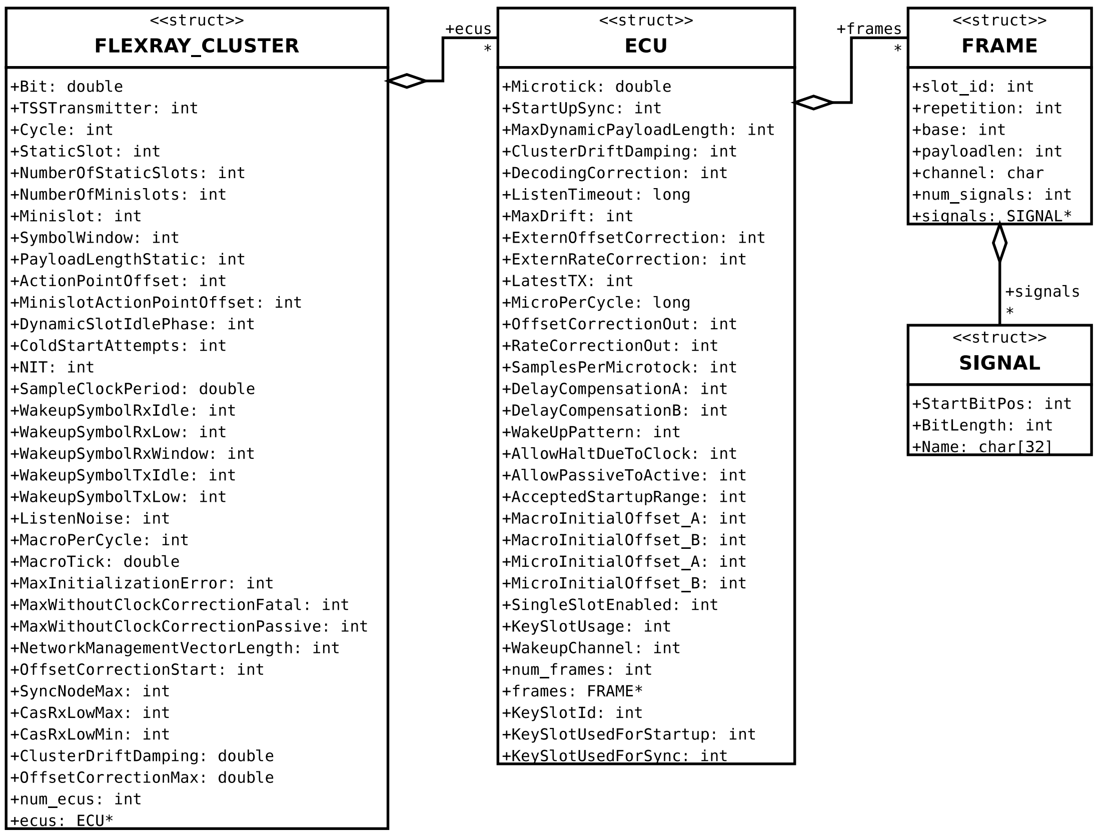


Figure 4.1: Scheme for the FlexRay configuration.

4.2.1.1 Cycle/Slot Multiplexing

To increase efficiency, a cycle/slot multiplexing scheme is supported by FlexRay and AUTOSAR. The AUTOSAR specification differentiates between single sender slot multiplexing and multiple sender slot multiplexing. The single sender slot multiplexing mechanism allows an ECU to transmit different frame contents in the same slot in different cycles while the multiple sender slot multiplexing mechanism allows several ECUs to share the same slot by uniquely assigning certain cycles to each ECU [6].

According to the *FlexRay Communications System - Protocol Specification, v2.1 Revision A* it is required that in the static segment each slot on a channel is owned by exactly one node. This prohibits the use of multiple sender slot multiplexing in the static segment. However, single sender slot multiplexing can still be used [2].

The multiplex of frames is defined by a base cycle and a cycle repetition: The base cycle defines the offset in cycles for the first occurrence of the respective frame. The cycle repetition denotes the frequency of a frame in the multiplexing. The value of the cycle repetition is always a power of two 2^n with $n \in \{0, \dots, 6\}$ to allow a periodic occurrence in the 64 cycles.

This behaviour is also supported in this implementation. It is possible to specify for each frame in the *FRAMElist* along the frame construction plan a base and repetition value which enable the employment of the cycle multiplexing scheme described above. The principles are also depicted in figure 4.2. There is one frame construction plan defined with repetition two and base cycle zero containing the messages $m1$ and $m2$ and another one with repetition two and base cycle one containing the message $m3$. This results in the same frame contents repeating every second communication cycle.

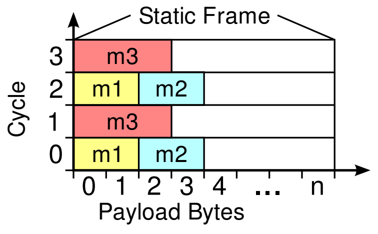


Figure 4.2: Principle of single sender slot multiplexing for static FlexRay frames.

4.2.2 VFB Configuration

In order to keep the configuration of the VFB simple for the application developer, the mechanics are encapsulated by preprocessor macros. The defined macros generate the necessary objects as well as the required RTE API from the given parameters. Also the mapping to signals defined by the FlexRay configuration is done by calling those macros. A comprehensive summary including a detailed description of all supported parameters as well as code examples of VFB configurations can be found in the appendix.

Two types of macros do exist. The first type is used for the configuration of ports, operations and data elements and the second one is required for the configuration of runnable entities. The following two subsections explain further details of the VFB configuration macros.

4.2.2.1 Port Definition Macros

The port definition macros are used to define ports, data elements and operations. The are inserted directly in the class declaration body of the software component. Several different macros exist. The following list provides further details on the functionality of the macros:

- *VFB_SEND_DATAELEMENT* and *VFB_RECEIVE_DATAELEMENT*: These two macros are used to define data elements which are either received or sent via a sender or receiver port respectively. For both of them a queue length may be specified for enabling the queued semantics as described by AUTOSAR. The receiving of data can be blocking or non-blocking. Which variant gets used is also specified as parameter for the macro. If the blocking variant is used, a timeout value is used for continuing execution if no data was received. The data element is mapped to a signal name. If the signal name is left empty, a default signal name is constructed from the port and data element name. This signal name must be defined in the FlexRay configuration if the communication is supposed to be performed via the communication network. Otherwise, the communication is performed locally only. Also the the data type and the endianness, which is used on the communication network for transmitting the data element, must be specified. On the receiving side it must also be specified whether the sign extension mechanism has to be applied for the data element and an initialisation value which is returned when the communication system is not yet operating can be given.

- *VFB_CLIENT_OPERATION* and *VFB_SERVER_OPERATION*: The definition of operations for client-server communication is accomplished by these macros. On the client side, it must be stated whether the call of the operation should be asynchronous. In case of a synchronous execution a timeout value is used for continuing execution on communication errors. On the server side, a runnable must be specified which implements the operation. For both sides, the operation must be assigned a request and response signal name. Again, if they are skipped, default names are constructed from the port and operation name. For remote communication the signal names must match signal names defined in the FlexRay configuration. An additional parameter is needed for specifying the endianness used on the communication network. The last necessary parameter is a value indicating the number of arguments for the operation. Afterwards, the arguments for the operation themselves are specified.

- *VFB_ARGUMENT*: For the specification of arguments to operations for client-server communication, the macro *VFB_ARGUMENT* is used. For the definition necessary values are a data type, a name, a direction and two signal names — one for the transfer from data into the operation, one for the transfer of data from the operation to the caller. The signal names can be skipped. In this case default names are constructed from the port, operation and argument name. If the operation for which the arguments are defined is a remote operation, also the required signal names for the argument must be defined in the FlexRay configuration. For each argument it must be specified whether the sign extension mechanism has to be applied.

4.2.2.2 Runnable Entity Definition Macros

For the definition of runnable entities, also some configuration macros exist. They are supposed to be called from within the constructor of the software component. The following list gives an overview:

- *VFB_RUNNABLE_PERIODIC*: This macro defines a runnable entity which is invoked periodically. The type of the software component as well as a period time and the name of the method are required as parameters.

- *VFB_RUNNABLE_TIMEOUT*: Similar to the periodic variant, this macro takes the type of the software component a timeout value and the name of the method as parameters. After the given timeout, the method is executed once.

- *VFB_RUNNABLE_DATA_RECEIVED* and *VFB_RUNNABLE_SEND_COMPLETE*: For reaction on events triggered by data elements, these two macros are implemented. They take the type of the software component, the port and data element names and the name of the method as parameters. The given method is executed whenever data is received or data is completely sent respectively.

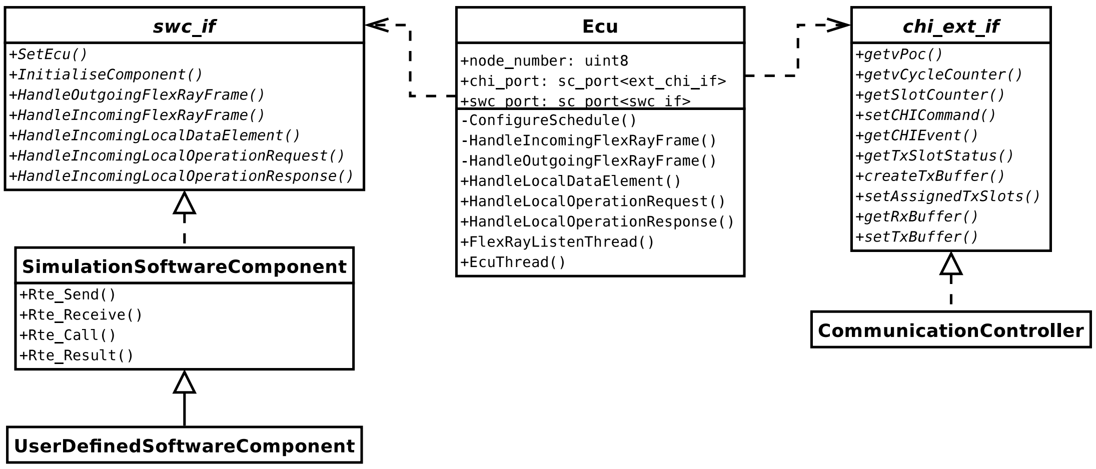
- *VFB_RUNNABLE_OPERATION_INVOKED* and *VFB_RUNNABLE_ASYNCHRONOUS_SERVER_-CALL_RETURNS*: To react to events of client-server ports, these macros provide the required interface. They take the type of the software component, the port and operation names and the name of the method as parameters. The given method is executed whenever an operation is invoked or an asynchronous operation returns.

4.3 Logical Architecture

In this section the decomposition of the system to classes and objects is described. This decomposition is not only for the sake of functional analysis, but also serves to identify common mechanisms and design elements across the various parts of the system. Each of the following sections is dedicated to one class. Where suitable, class diagrams are included for better understanding.

4.3.1 Software Component Interface

The software component interface (*swc_if*) is used to connect software components to the *Ecu* module. It is derived from the SystemC class *sc_interface* and therefore provides the possibility to use the SystemC port mechanism to connect the *swc_if* to the *Ecu*. A class diagram of the software component interface and the relations to other parts of the system can be found in figure 4.3. Whenever new data is available at the *Ecu*, the *Ecu* notifies the connected software components. Therefore, the software component interface defines methods which handle the incoming data. Also when the transmit buffer of a frame has to be filled with data, the connected software components are queried to write their data to the buffer. The following paragraphs contain the defined methods and a description of their functionality.



swc_if
+SetEcu()
+InitialiseComponent()
+HandleOutgoingFlexRayFrame()
+HandleIncomingFlexRayFrame()
+HandleIncomingLocalDataElement()
+HandleIncomingLocalOperationRequest()
+HandleIncomingLocalOperationResponse()
Ecu
+node_number: uint8
+chi_port: sc_port<ext_chi_if>
+swc_port: sc_port<swc_if>
-ConfigureSchedule()
-HandleIncomingFlexRayFrame()
-HandleOutgoingFlexRayFrame()
+HandleLocalDataElement()
+HandleLocalOperationRequest()
+HandleLocalOperationResponse()
+FlexRayListenThread()
+EcuThread()
chi_ext_if
+getvPoc()
+getvCycleCounter()
+getSlotCounter()
+setCHICommand()
+getCHIEvent()
+getTxSlotStatus()
+createTxBuffer()
+setAssignedTxSlots()
+getRxBuffer()
+setTxBuffer()
SimulationSoftwareComponent
+Rte_Send()
+Rte_Receive()
+Rte_Call()
+Rte_Result()
CommunicationController
UserDefinedSoftwareComponent

Figure 4.3: A class diagram showing the assignment of software components and the communication controller to the *Ecu*.

SetEcu: The method SetEcu is used to set a reference back to the *Ecu* which the software component is connected to.

InitialiseComponent: This method provides a hook for user defined software components to perform initialisation tasks. It is called upon simulation start.

HandleOutgoingFlexRayFrame: This method is used by the *Ecu* to ask a software component to write data which it wants to send via the FlexRay network into the transmit buffer of the communication controller.

HandleIncomingFlexRayFrame: Upon reception of a valid FlexRay frame, the receive buffer is provided to the software component and the software component extracts the necessary communication data from the receive buffer. This is done by invoking the method *HandleIncomingFlexRayFrame()*.

HandleIncomingLocalDataElement: This method copies the data from the shared memory given as parameter to the *DataElement* of the software component, if a *DataElement* is declared for the signal identified by the given signal name.

HandleIncomingLocalOperationRequest: Copies the input parameters of the *Operation* to the corresponding local *Operation* object if the software component implements the operation. The operation is invoked afterwards and the return value and output parameters are transmitted back to the caller.

HandleIncomingLocalOperationResponse: Copies the return value and the output parameters of a server operation which was executed by a different software component on the same *Ecu* back to the corresponding internal *Operation* object if available.

4.3.2 Communication Controller Interface

The communication controller interface (*chi_ext_if*) is the interface between the *Ecu* and the communication controller. It represents the implementation specific CHI which is roughly specified in the FlexRay protocol. The functionality was already sketched in section 2.1.3. It provides access to the transmit and receive buffers and protocol status information of the communication controller. The interface gets implemented by the communication controller. The most important methods of the interface which are used by the VFB abstraction layer are explained below.

getSlotCounter: Returns the current value of the slot counter of the *CommunicationController*.

getvCycleCounter: Returns the current value of the cycle counter of the *CommunicationController*.

getvPoc: Returns the current state of the protocol operation control unit. The possible values are listed in the FlexRay specification. This method is used by the *Ecu* when starting the communication system to monitor the state of the connected *CommunicationController*.

setCHICommand: Sends a command to the *CommunicationController*.

getCHIEvent: Takes as parameter the type of event which should be returned and returns an sc_event reference to the internal event of the *CommunicationController*. This mechanism allows the host to react on internal events of the *CommunicationController*.

setAssignedTxSlots: Sets the slots in which the *CommunicationController* transmits data.

getRxBuffer: Returns the receive buffer of the *CommunicationController*. It must be separately checked, that the receive buffer contains valid data.

createTxBuffer: Creates a transmit buffer. The transmit buffer is created for a single channel and a certain slot and cycle. In the static segment a precalculated header CRC can be used. It is also set upon creation of the transmit buffer.

setTxBuffer: Writes data to the transmit buffer of the *CommunicationController*. The transmit buffer needs to be created first with *createTxBuffer()*. After the transmit buffer has been created once, it is possible to write data multiple times (e.g., every cycle).

4.3.3 SimulationSoftwareComponent

The class *SimulationSoftwareComponent* implements the software component interface described above and is used as base class for all user defined software components. It provides general functionality for defining ports and data elements as well as for sending, storing and receiving data. The relations to other classes are shown in the class diagram in figure 4.3.

As the functionalities of this class include the greatest part of the middleware layer their descriptions are divided to several subsections. At first, the internal mechanisms for the definition of ports, data elements and operations are explained. Afterwards some comments on the definition of runnable entities are given, including a description of the concept for executing runnable entities in the context of the *Ecu*. The last subsection is dedicated to the internals of saving the communication data and explains the relationship between the different classes used for handling communication data.

4.3.3.1 Definition of Ports, Data Elements and Operations

This subsection gives further details on the realisation of the configuration of the VFB for a particular software component and is therefore closely related to the port definition macros explained in section 4.2.2.1. The purpose of this subsection is to illustrate the inner mechanics of the definition of ports, data elements and operations.

As already mentioned, on the VFB level, a software component communicates to other components via ports. From the view of a software component four different types of ports do exist: sender port, receiver port, client port and server port. For the implementation of the ports, the *SimulationSoftwareComponent* has four protected attributes called *sender_port*, *receiver_port*, *client_port* and *server_port*. They realise a mapping from a port name to a *Port* object. The *Port* objects again provide a mapping. The *SenderReceiverPort* maps data element names to *DataElement* object whereas the *ClientServerPort* maps operation names to *Operation* objects. A class diagram depicting those relations is given in figure 4.4(a). As precondition, a port name must be unique for a software component and a data element or operation name must be unique for one port. Those constraints are also required by the AUTOSAR specification, so they are fulfilled easily.

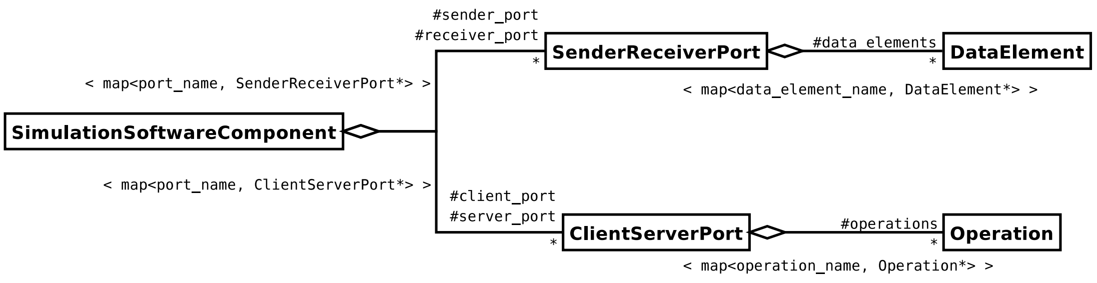
A second mapping method maps a frame index to a data element or operation. Therefore the *SimulationSoftwareComponent* has the protected attributes *sender_frame_data_elements* and *receiver_frame_data_elements* for *DataElements* and *client_frame_operations* and *server_frame_operations* for *Operations*. They map a frame index which uniquely represents a frame in the FlexRay configuration and, thus, also in the FlexRay schedule, to a *DataElement* and *Operation* object respectively. The aforementioned frame index is the index in the global *FRAMElist* configuration array. Within one frame multiple signals can be transmitted. This must be taken into account for the mapping data structure requiring a multimap in this case. The relation is depicted in figure 4.4(b)

Additionally, for local communication a third mapping method is used for mapping a signal name to a *DataElement* or *Operation* object. For this purpose, the *SimulationSoftwareComponent* provides the protected attributes *sender_signal_data_elements*, *receiver_signal_data_elements*, *client_signal_operations* and *server_signal_operations* for this purpose. This mapping is also depicted in figure 4.4(c).

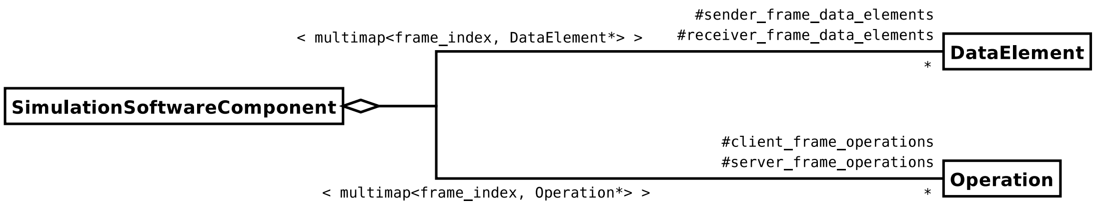
For generating the mappings, the *SimulationSoftwareComponent* declares several member functions. The following list illustrates the functionality of the port and data element definition functions and how the mapping is performed for the various communication elements.

DefineDataElementForSenderPort: Defines a data element which gets sent via a specified port. A *DataElement* object is created and referenced by the mapping data structures. An entry for the data element mapping of the corresponding sender_port is created. This method checks the FlexRay configuration if a signal is defined with the given signal name. If there is and the *Ecu* which the software component is running on does transmit in one the frame in which the signal is transmitted, an entry in the sender_frame_data_elements map is created.

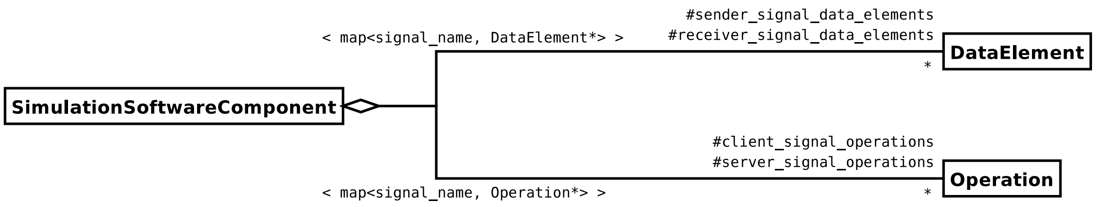
DefineDataElementForReceiverPort: Defines a data element which gets received via a specified port. A *DataElement* object is created and referenced by the mapping data structures. An entry for the data


#sender_port
#receiver_port
< map<port_name, SenderReceiverPort*> >
SenderReceiverPort
#data_elements
DataElement
< map<data_element_name, DataElement*> >
SimulationSoftwareComponent
< map<port_name, ClientServerPort*> >
#client_port
#server_port
ClientServerPort
#operations
Operation
< map<operation_name, Operation*> >

(a) Mapping of port structures to *DataElements* and *Operations*.


#sender_frame_data_elements
#receiver_frame_data_elements
< multimap<frame_index, DataElement*> >
DataElement
SimulationSoftwareComponent
#client_frame_operations
#server_frame_operations
Operation
< multimap<frame_index, Operation*> >

(b) Mapping of frame indices to *DataElements* and *Operations*.


#sender_signal_data_elements
#receiver_signal_data_elements
< map<signal_name, DataElement*> >
DataElement
SimulationSoftwareComponent
#client_signal_operations
#server_signal_operations
Operation
< map<signal_name, Operation*> >

(c) Mapping of signal names to *DataElements* and *Operations*.

Figure 4.4: Mapping schemes for accessing *DataElements* and *Operations*.

element mapping of the corresponding receiver_port is created. This method checks the FlexRay configuration if a signal is defined with the given signal name. If there is, a mapping in receiver_frame_data_elements is created. In addition, as a receiver may receive data from multiple senders including a remote sender and a local sender, a mapping in receiver_signal_data_elements is also created.

DefineOperationForClientPort: Defines an operation for a specific port which can be invoked by this software component. An *Operation* object is created and referenced to by the mapping data structures. At first, the operation is mapped by operation name in the client_port data structure. Furthermore, the operation has two signal names. One for transmitting the operation request and a second one for transmitting the operation response. If the request signal name is found in the FlexRay configuration and the *Ecu* can transmit the signal, a mapping in client_frame_operations is made. If the response signal name is found in the FlexRay configuration, a mapping in server_frame_operations is created. As a always is received from a single server software component, in this case, no entry in server_signal_operations is necessary. If no signal is found in the FlexRay configuration, the communication is supposed to be local and a mapping for the signal name is created in server_signal_operations.

DefineOperationForServerPort: Defines an operation for a specific port which is implemented by this software component. An *Operation* object is created and referenced to by the mapping data structures. The operation object is referenced in the server_port data structure. If the request signal name of the operation is found in the FlexRay configuration, a mapping in client_frame_operations is created. A server may receive requests from multiple clients, so also local communication must be taken into account. This results in an entry for the request signal name in client_signal_operations. If the response signal is found in the FlexRay configuration, a mapping in server_frame_operations is created.

The mappings are all set up upon either the first call to a RTE API method concerning the respective data element or operation or upon a call to a dedicated initialisation function. Also, when a runnable entity is created which references a specific data element or operation this data element or operation the mapping table entries are created according to the specification in the VFB configuration.

Going further down the hierarchy, *Operations* can have *Arguments*. An *Argument* of an *Operation* is similar to a *DataElement*. It is also responsible for saving communication data, but whereas the *DataElement* supports queuing of communication data, the *Argument* provides information about the direction of the data. There is a mapping of a string identifying the argument and an *Argument* object within the operation. The arguments of an operation are ordered and are members of the *Operation* object (see also 4.3.10).

4.3.3.2 Definition of Runnable Entities

Runnable entities are methods of the software component class. In order to be able to pass pointers to those methods to the *Ecu*, which executes the runnable entities, it is necessary to encapsulate the functions in functor objects. The functor object holds a pointer to the software component and a pointer to the member function, providing a unique identity for the runnable entity. The definition of runnable entities, therefore, is implemented as creation of functor objects. After creation of a functor object, it is passed to the *Ecu* which creates a SystemC thread from the functor. The thread concept for software component runnable entities is described later in section 4.4.3. There exist three types of specific functors which inherit from a common *PlainFunctor* class:

- *EventTriggeredFunctor*: This functor type takes a triggering sc_event as parameter on construction and executes the assigned function whenever the sc_event is notified.
- *PeriodicallyTriggeredFunctor*: In the constructor of the functor, a period time is a required parameter. Whenever the simulation time matches a multiple of the period time the assigned function is executed.
- *TimeoutTriggeredFunctor*: This functor type needs to be given a timeout in its constructor. After the given amount of time, the assigned function is executed. In contrast to the *PeriodicallyTriggeredFunctor* the function is only executed once.

A class diagram providing an overview of the functors is shown in figure 4.5. For the software component author the whole mechanism is hidden by preprocessor macros.

4.3.3.3 Saving Communication Data

As already mentioned, the software component must intermediately save the data which it receives or sends via ports. Therefore each data element and each argument for an operation is implemented as object which allocates memory for temporarily saving its communication data. A class diagram showing the relations between the different class types is depicted in figure 4.6. The functionalities of the involved classes are explained below in section 4.3.5 to section 4.3.10

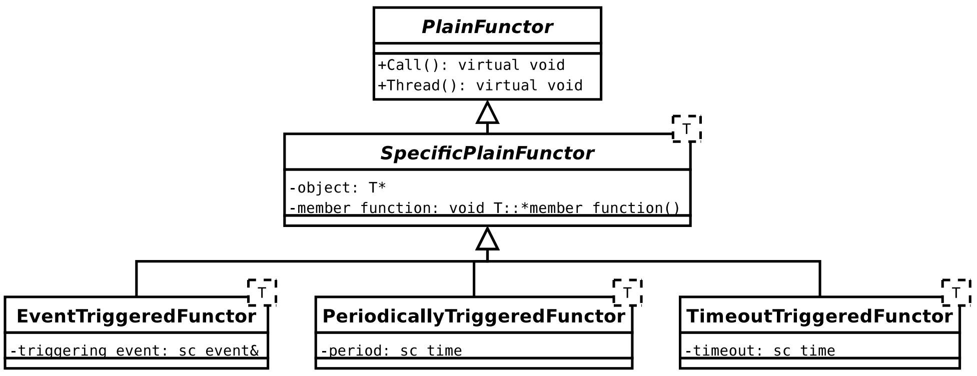


Figure 4.5: Hierarchy of the functor classes used for the definition of runnable entities.

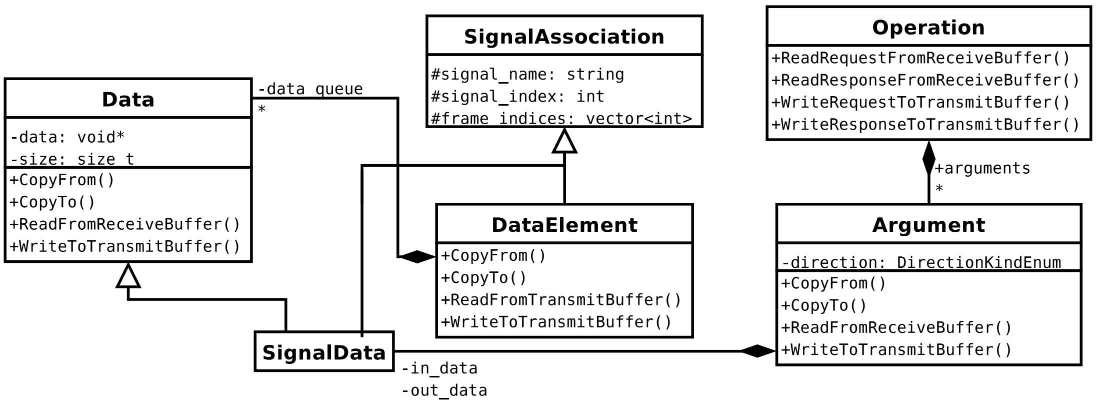


Figure 4.6: Class diagram showing the relation between different intermediate data storage types.

4.3.4 Ecu

The *Ecu* class takes care of the communication between software components. It provides SystemC ports to connect a communication controller and software components. Furthermore, the *Ecu* is used as main processing unit.

To communicate via the communication system, the *Ecu* encapsulates a thread which is sensitive to events which get triggered by the communication controller. AUTOSAR prohibits communication of software components by other means than via the ports defined on the VFB level. Therefore, also the communication between two software components which are connected to the same *Ecu* is performed via the *Ecu* class by using shared memory. The relevant functions of the *Ecu* class are listed below.

SpawnPlainFunctorThread: Creates a SystemC thread from a runnable entity. The runnable entity and a reference to the event which it is triggered by is encapsulated in a functor object and passed as argument.

HandleLocalDataElement: Forwards a *DataElement* to all connected software components to allow them to copy the data of the *DataElement* to their own memory.

HandleLocalOperationRequest: Forwards a request for the execution of an operation to all connected software components. The operation is executed by the software component which implements it.

HandleLocalOperationResponse: After an operation has been executed by a software component, the return value and output parameters must be passed back to the caller. This method passes the data to every connected software component. The software component which requested the execution copies the returned data to its own memory.

FlexRayListenThread: Implements a thread for monitoring the *CommunicationController*. This method immediately reacts to events on the *CommunicationController*.

EcuThread: Implements a thread for the initialisation of the system. This method starts up the *CommunicationController* and creates the monitoring SystemC threads. Furthermore an initialisation of the software components is done.

ConfigureSchedule: Configures the transmit/receive resources of the *CommunicationController*. The *CommunicationController* is assigned the slots which it is allowed to transmit in. Furthermore the transmit and receive buffers are created and initialised in this method.

HandleIncomingFlexRayFrame: Handles an incoming FlexRay frame by forwarding the received buffer to all connected software components. The software component extracts the data which it needs from the buffer.

HandleOutgoingFlexRayFrame: Before a transmit buffer is written to the *CommunicationController*, all connected software components are queried to write the data they are configured to transmit in the frame to the transmit buffer.

4.3.5 Data

The *Data* class handles the intermediate storage of data for *Arguments* and *DataElements*. It internally has reserved memory and provides methods to copy data from and to arbitrary memory. Furthermore, the *Data* class also provides methods for writing the internally stored data to a FlexRay transmit buffer and for reading data from a receive buffer and storing the data in its internal memory.

4.3.6 SignalAssociation

The *SignalAssociation* class handles the association of an object and a signal which was defined in the FlexRay configuration. It allows the storage of a signal name and resolves the given signal name to an index determining the signal in the FlexRay configuration.

4.3.7 SignalData

The *SignalData* class combines the features of the *Data* and the *SignalAssociation*. It has internal memory and also a signal name.

4.3.8 DataElement

The *DataElement* is a descendant of the *SignalAssociation*. A *DataElement* always requires a signal name. In difference to the *SignalData* the *DataElement* contains a queue of *Data* objects. This is necessary to allow for using the queued transmission mechanism as specified by AUTOSAR. The queue length is set when the data element is defined for the VFB configuration. The methods of the *DataElement* class are described in detail in the following list.

CopyFrom: Copies the data from an arbitrary memory location, typically a variable which gets transmitted by the communication system. If the *DataElement* uses a queue, the new data is appended to the queue. Otherwise, the new data overwrites the old data. If the data cannot be copied (e.g., the queue is full), the method returns false.

CopyTo: Copies the internally stored data to an arbitrary memory location, typically a variable which is received from the communication system. If the *DataElement* uses a queue, the oldest data value is returned. Otherwise, there is only one data value and this is returned. If the data cannot be copied (e.g., the queue is empty), the method returns false.

ReadFromReceiveBuffer: Reads data from a receive buffer. If the *DataElement* uses a queue, the new data is appended to the queue. Otherwise, the read data overwrites the old data. After data has successfully been read, a DataReceivedEvent is triggered. If the data cannot be copied (e.g., the queue is full), the method returns false.

WriteToTransmitBuffer: Writes the internally stored data to a transmit buffer. If the *DataElement* uses a queue, the oldest data value is written. After the data has been written successfully to the buffer, a DataSendCompletedEvent is triggered. If the data cannot be written (e.g., the queue is empty), the method returns false.

4.3.9 Argument

Another form of data saving object is given with the *Argument* class. An *Argument* object saves data which is passed as argument to an operation of a client-server interface. The difference of an *Argument* to a *DataElement* is that an *Argument* does not provide the queued form of transmission. Furthermore, an *Argument* has a direction. It can pass data to an operation, return data from an operation or both. If an *Argument* is used to pass data to an operation and to return data from an operation, it must have two signal names and the data must be transmitted in different FlexRay frames. Therefore, the *Argument* internally stores the data in two *SignalData* objects, one is used for the inbound data, one for the outbound data.

The methods of the *Argument* class are similar to the methods of the *DataElement* class. However, for all of them an additional direction parameter is required.

CopyFrom: Copies the data from an arbitrary memory location. Depending on the direction parameter, the data gets either copied to the input or the output *SignalData* of the *Argument*.

CopyTo: Copies the internally stored data to an arbitrary memory location. Depending on the direction parameter, the data gets either copied from the input or the output *SignalData* of the *Argument*.

ReadFromReceiveBuffer: Reads data from a receive buffer. Depending on the direction parameter, the read data gets either copied to the input or the output *SignalData* of the *Argument*. On the server side, typically the input data is read from the receive buffer; on the client side typically the output of an operation is read from the receive buffer.

WriteToTransmitBuffer: Writes data to a transmit buffer. Depending on the direction parameter, either the input or the output *SignalData* of the *Argument* gets written to the buffer. On the server side, typically the output data is written to the transmit buffer; on the client side, typically the input data is written to the transmit buffer.

4.3.10 Operation

The *Operation* class is used to handle the execution of an operation. It contains an ordered list of *Arguments*. When an operation gets called, the argument data which is passed to the operation gets copied to the internal *Argument* objects. When the operation returns, the data which is returned from the operation is copied from the internal *Argument* objects.

In addition to the operation specific arguments as defined in the VFB configuration, two additional special *Arguments* are used for the *Operation*: a request *Argument* and a response *Argument*. They are used as workaround for a limitation of the configuration format of the FlexRay configuration. AUTOSAR supports the usage of update bits for every signal. This update bit is an additional bit indicating whether the transmitted data was updated since the last transmission. However, no update bits are supported by the currently used FlexRay signal configuration. For client-server communication the update bit has an additional purpose. If the update bit of any input argument of an operation is set, it can be assumed, that the client requested to execute the operation. Similar assumptions are possible for output arguments: if an update bit of an output argument is set, the client can assume that the operation was executed by the server and continue processing the response. Especially in case an operation is invoked without any argument, usage of the update bits is required. In this special case, AUTOSAR requires an empty signal with an update bit to be defined to notify the server of the invocation of an operation. This is accomplished in this implementation with the request and response *Arguments*. They require only one bit and signal the server whether an operation was invoked by the client and the client whether the response of the operation is ready.

AddArgument: Adds an argument to the internally maintained ordered list of arguments. Each argument needs to have a unique name to be able to identify it and also a direction.

ReadRequestFromReceiveBuffer: Reads the input arguments for an operation request from a received FlexRay buffer. This usually is only required on the server side of a client server interface. This method also reads the request signal. If the request signal indicates that the operation should be invoked, an OperationInvokedEvent is triggered.

ReadResponseFromReceiveBuffer: Reads the returned data from an operation from a received FlexRay buffer. This is usually done on the client side of a client-server interface. Also reads the response signal. If the response signal indicates that the operation response is ready, an AsynchronousServerCallReturnsEvent is triggered to allow the application to continue execution and to react on the received data.

WriteRequestToTransmitBuffer: Writes an operation request to a FlexRay transmit buffer. This includes all input arguments of the operation and the request signal.

WriteResponseToTransmitBuffer: Writes the response of an operation including all output arguments and the response signal to a FlexRay transmit buffer.

4.4 Process Architecture

This section contains further information about the cooperation of the different classes including timing information. The simulation of the software components using the VFB abstraction layer is untimed. This means, that no information of the duration of different operations is annotated. Therefore, the simulated time does not advance while executing a runnable entity. The communication system, however, does provide accurate timing information. So the whole system can be considered to be modelled on an abstraction level similar to the Task level already mentioned in section 2.3.3.4.

As the implementation is done in SystemC, the concept of threads of SystemC is used. A thread in SystemC is a part of code which is executed until the flow of code reaches a special *wait* statement. The execution is stopped at the wait statement. The *wait* statement is sensitive to either an *sc_event*, a timeout or both. If it gets triggered, the code execution continues. Several SystemC threads are used in this implementation. The following subsections should give an overview of the threads and provide an insight into the cooperation between the threads. Where suitable sequence and activity diagrams are used for better understanding.

4.4.1 Initialisation of the Communication System

A dedicated SystemC thread is used for the initialisation of the communication system. The *EcuThread* is executed on the *Ecu* and started immediately upon simulation start. Before any action is performed by this thread, it waits for an externally configurable waiting time. This waiting time is included to allow other simulation models (e.g., the transceiver) to initialise themselves. After the waiting time has passed, the *EcuThread* initialises the *CommunicationController* and configures the FlexRay schedule of the *CommunicationController* and created the necessary receive/transmit buffers.

After the configuration has completed, the *Ecu* tries to startup the FlexRay communication. The FlexRay protocol provides different forms of startup for a communication controller (see section 2.1.4). Therefore, the *Ecu* has an externally configurable parameter to define whether the *CommunicationController* should act as coldstart node or not. If it is configured as coldstart node, the FlexRay cluster is started by the *CommunicationController*, otherwise it tries to integrate into an existing schedule. As soon as the *CommunicationController* signals a successful startup of the communication, the *EcuThread* spawns additional threads for each FlexRay channel, which handle the processing of the communication controller related work. The *EcuThread* has finished its work afterwards and is terminated. An activity diagram can be found in figure 4.7.

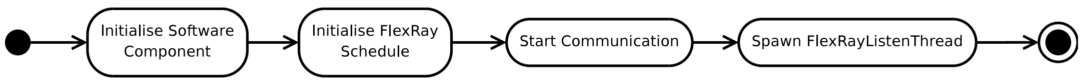

Initialise Software Component
Initialise FlexRay Schedule
Start Communication
Spawn FlexRayListenThread

Figure 4.7: Activity diagram of the EcuThread.

4.4.2 Collaboration of the Software Components and the Communication System

Having explained the involved classes and objects used for realisation of the communication between software components, the purpose of this section is to give insight into the collaboration between the different classes for local communication as well as for remote communication. Furthermore other necessary tasks for remote communication are explained. Some thoughts are given on the calculation of the header CRC. Afterwards the concept of sign extension is explained. Finally, the adaptation of signal data to the network byte order is sketched.

4.4.2.1 Local Communication

The VFB allows the communication between software components which are running on the same ECU. As already mentioned, the local communication is handled by the *Ecu* class. If a software component wants to transmit a data element, it calls an RTE API method. This method makes the *Ecu* forward the data to every connected software component which in turn handles the data element if is is configured to be received by the software component. A sequence diagram showing the procedure can be found in figure 4.8.

For operations a similar approach is used. The difference is, that in addition to the operation request and the input parameters of the operation, also a handling of the return value and output parameters of the operation is needed. So, at first the input data is copied to the software component providing an implementation for the called operation (server). The operation is executed by the server software component and afterwards, the results and output parameters are copied back to the calling software component (client).

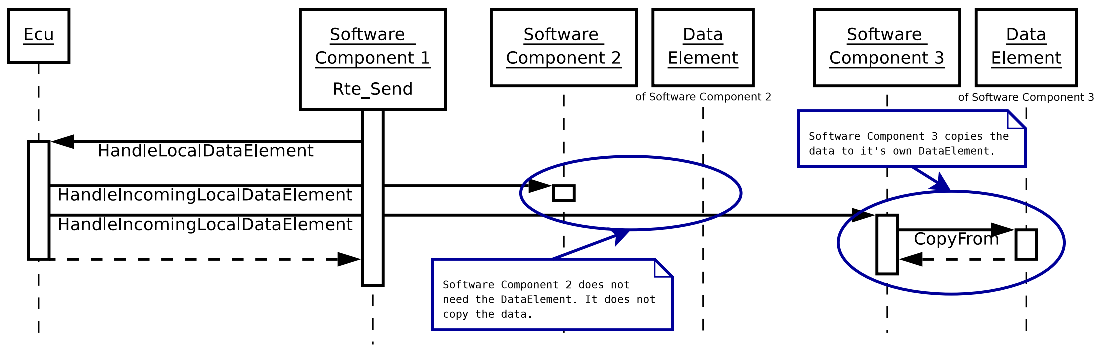

Ecu
Software Component 1
Rte_Send
Software Component 2
Data Element
of Software Component 2
Software Component 3
Data Element
of Software Component 3
HandleLocalDataElement
HandleIncomingLocalDataElement
HandleIncomingLocalDataElement
Software Component 3 copies the data to it's own DataElement.
CopyFrom
Software Component 2 does not need the DataElement. It does not copy the data.

Figure 4.8: Sequence diagram of the transmission of *DataElements* to other software components connected to the same *Ecu*.

4.4.2.2 Remote Communication

The *CommunicationController* is monitored by a dedicated SystemC thread. For each channel an instance of the *FlexRayListenThread* is started after the initialisation of the communication controller. An activity diagram of this thread is given in figure 4.9. After the completion of each static and dynamic transmission slot, the *CommunicationController* raises an event. The *FlexRayListenThread* reacts to those events.

Whenever a valid frame was received in the n^{th} slot, the *FlexRayListenThread* calls the *HandleIncomingFlexRayFrame* method of every connected software component. This method takes care of the extraction of relevant signal data from the receive buffer of the communication controller.

When the *CommunicationController* notifies the *Ecu* that the slot n has been received, it is already processing the following slot. Therefore, the *Ecu* can at this point in the schedule assemble the slot $n+2$. If the node is configured to send in the $(n+2)^{th}$ slot, the *FlexRayListenThread* calls the *HandleOutgoingFlexRayFrame* method of every connected software component. This method writes all relevant signal data of a software component to a buffer in the *Ecu* given as parameter by the *FlexRayListenThread*. Afterwards, the buffer is copied to the transmit buffer of the *CommunicationController*.

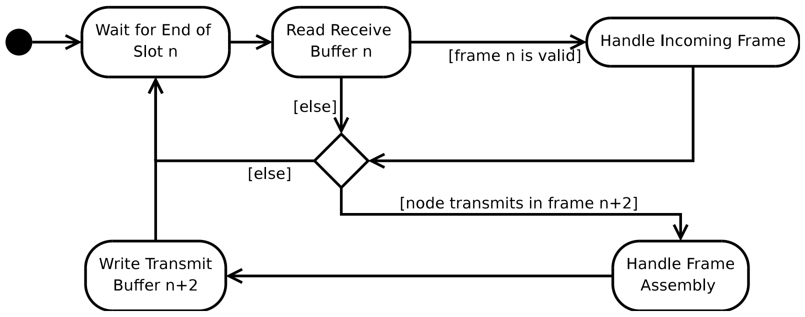


Figure 4.9: Activity diagram of the *FlexRayListenThread*.

The following two paragraphs explain the collaboration for sending and receiving a data element via the FlexRay communication system.

Sending a Data Element

The transmission of data is initiated by a sending software component by calling an RTE API method.

This method copies the data to the communication data saving object (*DataElement* or *Operation*) of the software component. Later when the communication system is ready to transmit a specific slot, the *FlexRayListenThread* running on the *Ecu* asks all connected software components to copy their data for the specific slot to the transmit buffer of the communication controller. A sequence diagram in figure 4.10 shows a schematic of the process of copying a data element to the transmit buffer of the FlexRay communication controller.

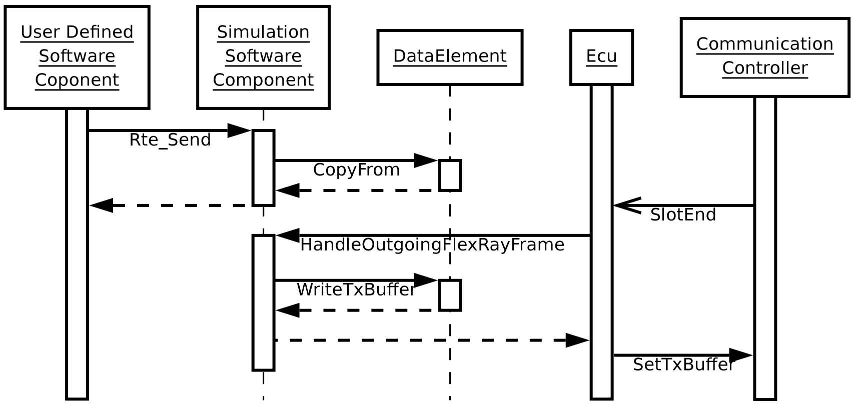

User Defined Software Coponent
Simulation Software Component
DataElement
Ecu
Communication Controller
Rte_Send
CopyFrom
SlotEnd
HandleOutgoingFlexRayFrame
WriteTxBuffer
SetTxBuffer

Figure 4.10: Sequence diagram of sending data via the FlexRay communication system.

Receiving a Data Element

The software component wants to receive data with a call to a RTE API method. In example depicted in figure 4.11, a blocking read is performed. No data is available so the RTE API method does not return. When the communication system signals, that a FlexRay frame was received, the *Ecu* asks all connected software components to extract data which is relevant for them from the receive buffer of the communication controller. After data has been copied to the *DataElement*, the *DataElement* signals that data is available again. Therefore, the RTE API method can continue and read the required data from the *DataElement* and return control to the calling runnable entity of the software component.

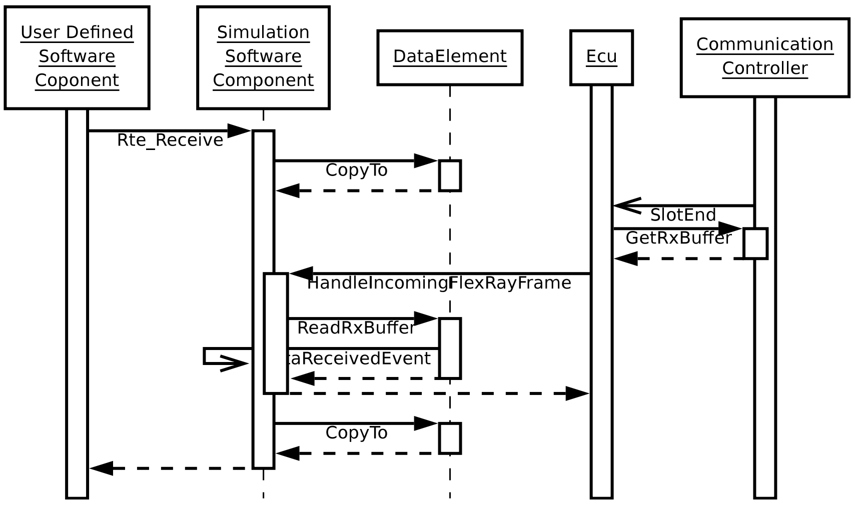


Figure 4.11: Sequence diagram of receiving data via the FlexRay communication system. There is no data waiting and the receiving call is blocking.

4.4.2.3 Header Checksum Calculation

The FlexRay protocol requires to have a CRC checksum calculated over some fields of the frame header. Among its other uses, the header CRC field of a FlexRay frame is intended to provide protection against improper modification of the sync frame indicator or startup frame indicator fields by a faulty communication controller. The communication controller that is responsible for transmitting a particular frame shall not compute the header CRC field for that frame. Rather, the communication controller shall be configured with the appropriate header CRC for a given frame by the host [2].

Therefore the *Ecu* must be able to calculate the header checksum. For static frames, the header checksum can be precalculated when the schedule is configured because all relevant fields (sync frame indicator or startup frame indicator, frame id and payload length) are already known at this time and statically configured. For dynamic frames, however, the payload length is not fixed and therefore the header CRC field must be calculated on the fly immediately before writing the buffer to the transmit buffer of the *CommunicationController*. This is also performed in the *FlexRayListenThread* after querying the software components for their communication data.

4.4.2.4 Sign Extension

For the transmission of signed integer data types, problems arise when using less bits for the transmission of the data via the communication network than are used for calculation in the ECU. For example the smallest datatype an ECU can handle has eight bits and an eight bit signed integer has the value -3_{dec}. Its binary representation is 11111101_{bin}. The absolute value never is higher than 10_{dec}, so it is decided to transmit the data using five bits of a FlexRay frame. On the receiver side, for -3_{dec}, the binary value 11101_{bin} is received. As the ECU requires at least eight bits to be able to handle data, the received value has to be extended to 11111101_{bin} again. This process is called sign extension and defined in the AUTOSAR specification [7].

On the other hand, an unsigned eight bit integer having the value of 29_{dec} has the binary representation 00011101. So if it is decided that, also for this integer, only the least significant five bits are required and transmitted via the FlexRay network, on the receiver side, the received data would as well be 11101_{bin}. If sign extension is applied in this case, the result is also 11111101_{bin} (-3_{dec}) which is different to the original value and not correct. The two examples are also illustrated in figure 4.12.

So it is clear that a value transmitted via the communication network can be interpreted in different ways and that sign extension has to be applied to signed data types but must not be applied to unsigned data types. Therefore the receiving *Ecu* has to be aware whether sign extension must be used for a particular signal or not. The information is specified when defining the data element or argument in the VFB configuration.

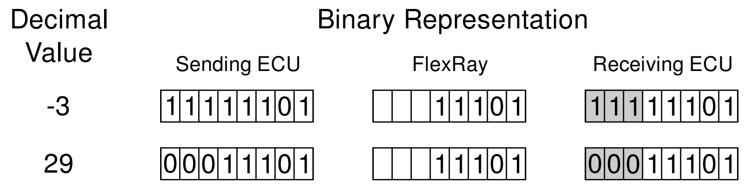


Figure 4.12: Different interpretation of received data for signed and unsigned datatypes.

4.4.2.5 Endianness Conversion

When interacting with buffers of the communication system the endianness of the data plays an important role. It must be ensured, that the data transmitted via the communication system can be interpreted by the communication partner in a correct way. Neither AUTOSAR nor FlexRay force the nodes to use a specific network byte order. The AUTOSAR specification defines a configuration parameter for specifying the network byte order. It can either be globally set to big-endian or little-endian or AUTOSAR can be configured to allow setting the endianness on a per-signal basis. As this provides the greatest flexibility, in this implementation the configuration on a per-signal basis is used.

The AUTOSAR Specification of Communication depends on the OSEK Communication Specification [35]. So the methodology for assembly of frames from signal data also follows the OSEK Communication Specification in this implementation. Figure 4.13 depicts the byte orders as specified by the OSEK specification.

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 0	7	6	5	4	3	2	1	0
Byte 1	15 ← 2	14 ← 1	13 LSB ← 0	12	11	10	9	8
Byte 2	23 ← 10	22 ← 9	21 ← 8	20 ← 7	19 ← 6	18 ← 5	17 ← 4	16 ← 3
Byte 3	31	30	29	28	27	26	25	24 MSB ← 11
Byte 4	39	38	37	36	35	34	33	32

(a) Little-endian byte order.

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 0	7	6	5	4	3	2	1	0
Byte 1	15	14	13 MSB ← 11	12 ← 10	11 ← 9	10 ← 8	9 ← 7	8 ← 6
Byte 2	23 ← 5	22 ← 4	21 ← 3	20 ← 2	19 ← 1	18 LSB ← 0	17	16
Byte 3	31	30	29	28	27	26	25	24
Byte 4	39	38	37	36	35	34	33	32

(b) Big-endian byte order.

Figure 4.13: Byte orders as specified in the OSEK Communication Specification. [35]

4.4.3 Software Component Runnables

A software component can define runnable entities. A runnable entity is a piece of code of a software component which gets executed if a certain event occurs. Events may be based on conditions of the communication system or based on timeouts. The realisation of this AUTOSAR concept is internally implemented such that for every runnable entity a SystemC thread is created. The SystemC threads are suspended immediately after creation and are left in a waiting state. If the event which they are waiting for is raised, the code of the runnable entity is executed. Afterwards, the thread moves back to the waiting state again.

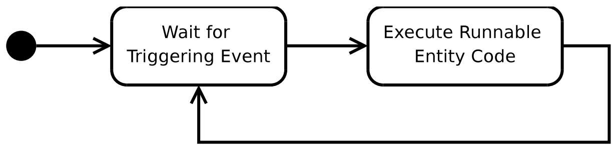


Figure 4.14: Activity diagram of a thread of a thread for a runnable entity of a software component.

4.5 Creation of a Simulation Scenario

In this section the creation of a simulation scenario using the presented implementation should be explained. It is meant as quick reference for the application developer.

4.5.1 Using Sender-Receiver Communication

The first part of this section gives an example for the implementation of a software component and the configuration of the FlexRay schedule for a scenario using sender-receiver communication.

4.5.1.1 Implementation and Integration of a Software Component

This part of the quick reference deals with the creation of a software component. For illustrating the needed steps, an example software component is created which contains a sender port which sends two data elements.

As already mentioned, the software components functionality is bundled in user defined software component classes which are descendants of the *SimulationSoftwareComponent* class. Therefore, the first step when creating a software component is to declare a class which inherits the functionality of the *SimulationSoftwareComponent*:

```
class SWC_ExampleSender : public SimulationSoftwareComponent {
```

Afterwards, the software component has to be extended by the VFB configuration including port, data element and operation definitions:

```
  public:

    /* **
     * Port API definitions.
     */
    VFB_SEND_DATAELEMENT(Port0, DataElement0, 1, Port0_DataElement0,
               int, END_LITTLE_ENDIAN);
    VFB_SEND_DATAELEMENT(Port0, DataElement1, 1, Port0_DataElement1,
               sint8, END_LITTLE_ENDIAN);
```

In this example, there is a single sending port with the name *Port0* defined which sends two data elements called *DataElement0* and *DataElement1*. For transmission, the signals *Port0_DataElement0* and *Port0_-DataElement1* are used. The first data element is of type *int* and the second data element has the type *sint8*. Both data elements are transmitted with little endian byte order on the communication system.

The VFB configuration macros must be inserted directly in the class body of the software component because internally they need to declare the RTE API methods for the defined data element or operation. Aside from the RTE API methods, an additional initialisation method is declared. To further clarify the process of generating the RTE API calls from the configuration macros, with the help of the data element *DataElement0* declared above, the generated methods are illustrated:

- *Rte_Send_Port0_DataElement0*, *Rte_Write_Port0_DataElement0*: These methods reflect the RTE API for sending data via the defined port. They internally call the *Rte_Send* method of the *SimulationSoftwareComponent* which actually handles the transmission.
- *Initialise_Port0_DataElement0*: This method creates the *DataElement* object and the mapping of the port name and data element name as well as of the frame index or signal name to the created *DataElement* object. It is called once at the first attempt to call one of the sending RTE API methods.

Another part of the configuration of the software component is the declaration of methods which can be used as runnable entities. Except for methods implementing operations of a client-server interface, they are simple c++ methods without return value and arguments. Within the runnable entity, the application is able to send data on the defined ports by calling the RTE API methods. The implementation does not have to be done in the header file directly but can be moved to a dedicated source file. It is included here directly in the header for the sake of simplicity.

```
    /**
     * RunnableEntity declarations.
     */
    void SenderRunnable0() {
      int data = 0x01020304;
      Rte_Send_Port0_DataElement0(data);
    };
```

In the class constructor, finally the runnable entities can be defined:

```
    /**
     * Class constructor.
     */
    SWC_ExampleSender(sc_module_name nm) : SimulationSoftwareComponent(nm) {
      /**
       * Runnable entity definitions.
       */
      VFB_RUNNABLE_PERIODIC(SWC_ExampleSender, 0.0052, SenderRunnable0);
    };
};
```

The example above defines that the *SenderRunnable0* method of the software component must be invoked periodically every 0.0052 *sec*.

To integrate the software component into the co-simulation environment, a SyAD module has to be created from the SystemC module implemented above. SyAD needs to know, that the software component implements the *swc_if* interface in order to be able to connect it to an *Ecu* module. In SyAD this is done by declaring an interface port which has the same name as the implemented interface. For interaction with other modules, additional ports can be added.

4.5.1.2 Setting up the FlexRay Cluster

Using the available SyAD modules for the various parts of the communication system, it is possible to easily create a topology for the simulation of the interaction between software components. For the purpose of illustration, in addition to the *SWC_ExampleSender* software component implemented above, a second software component which receives the data called *SWC_ExampleReceiver* was created which prints the data it receives to the SystemC console. Its code can be found in the appendix in section B.1. In figure 4.15 a scenario including the two software components is depicted. They are connected to two different nodes which are communicating via FlexRay. Additional parameters which must be set in SyAD are the node numbers and the waiting time. Furthermore, both nodes have to be configured to be coldstart nodes.

As next step, the FlexRay configuration of the cluster can be specified. The general FlexRay parameter configuration is not in the scope of this document. However, the schedule related configuration parameters depend on the usage of the communication system by the software components. Therefore, this example includes information about the FlexRay configuration but the information is limited to the FlexRay schedule.

For the software components described above, there are two signals defined: *Port0_DataElement0* and *Port0_DataElement1*. They are both transmitted via the FlexRay network and must be present in the FlexRay configuration. Depending on the size of the data to be transmitted, they may be transferred within a single FlexRay frame or in different frames. The example above defines the *Port0_DataElement0* as *int*. Typically an *int* requires 32 bit of memory. The *Port0_DataElement1* is an *sint8* datatype and requires additional 8 bit. Probably, both data elements can be transmitted in a single frame. This, however, depends on other parameters of the FlexRay configuration (basically on the parameter *gdPayloadLengthStatic*). For now a big enough static payload length is assumed.

The example below presents the configuration for the transmission of both signals in a single FlexRay frame. Only the parts of the configuration which are relevant for the configuration of the schedule are included.

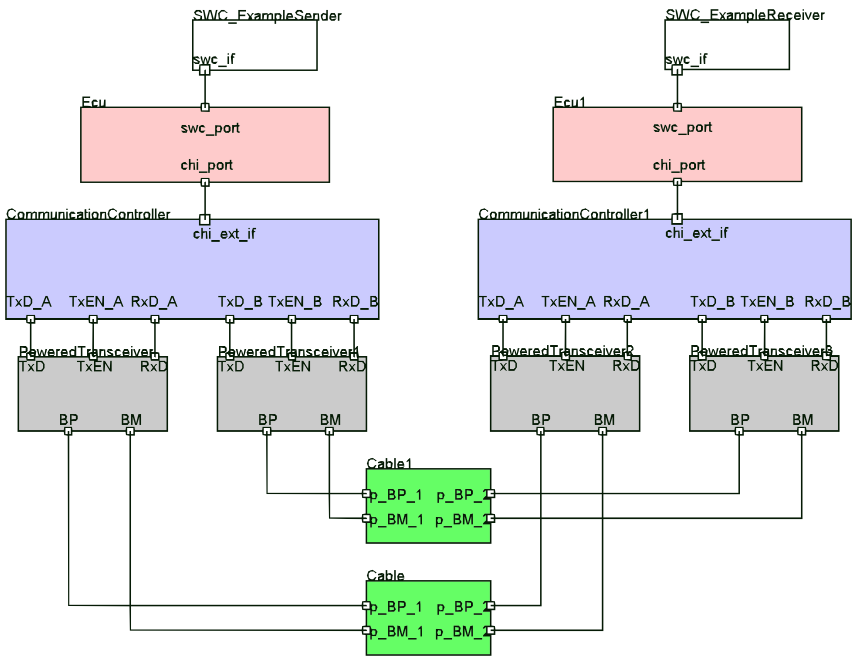


Figure 4.15: Example of a simple simulation scenario including a sending and a receiving software component.

```
const SIGNAL SIGNALlist[] =
{
    {                        // Signal Idx: 0
        "Port0_DataElement0",    //Name of the signal
        0,                       //Starting position (in bits)
        32,                      //Length of the signal (in bits)
    }
  ,
    {                        // Signal Idx: 1
        "Port0_DataElement1", //Name of the signal
        32,                      //Starting position (in bits)
        8,                       //Length of the signal (in bits)
    }
};

const FRAME FRAMElist[] =
{
    {                        // Frame Idx: 0
        4,                       //slot_id
        1,                       //repetition
        0,                       //base
        5,                       //framelen
        'A',                     //flexray channel
        2,                       //number of signals in this frame
        &SIGNALlist[0]           //a pointer to the signals
    }
  ,
    {                        // Frame Idx: 1
        7,                       //slot_id
        1,                       //repetition
        0,                       //base
        0,                       //framelen
        'A',                     //flexray channel
        0,                       //number of signals in this frame
        NULL                     //a pointer to the signals
    }
}

const ECU ECUlist[] =
{
    {
        "Sender Node",             //Name
//  [...]
        1,                         //Number of frames trasmitted by this ECU
        &FRAMElist[0],             //A pointer to the correct frames
        4,                         //KeySlotId
        1,                         //KeySlotUsedForStartup
        1,                         //KeySlotUsedForSync
    }
  ,
    {
        "Receiver Node",           //Name
//  [...]
        1,                         //Number of frames trasmitted by this ECU
        &FRAMElist[1],             //A pointer to the correct frames
        7,                         //KeySlotId
        1,                         //KeySlotUsedForStartup
        1,                         //KeySlotUsedForSync
    }
}
```

In the configuration specified above, also the receiving node is assigned a frame. Because for a start of the FlexRay cluster at least two nodes are required, the second node also has to transmit frames. Both nodes have assigned a key slot and use this slot for sending startup and synchronisation frames. While the sending node also transmits data in this slot after a successful startup, the receiving node simply transmits a frame without data (null frame).

4.5.2 Using Client-Server Communication

The approach for realising a client-server based communication between two software components is similar to the approach used for sender-receiver communication. Also in this case, a software component class has do be defined.

The VFB configuration in this case consists of a client port on the software component which needs to execute a remote operation and a server port on the software component which implements an operation respectively. In connection with the port, also the operation including all of its arguments are defined. In this case the *Operation0* of *Port0* is defined with three arguments and gets implemented in the method *ServerRunnable0*. In the example for all signals the default signal names are used.

```
class SWC_ExampleServer : public SimulationSoftwareComponent {
  public:
    /**
     * Port API definitions.
     */
    VFB_SERVER_OPERATION(Port0, Operation0, ServerRunnable0, , , END_LITTLE_ENDIAN, 3,
            VFB_ARGUMENT(sint8, InInt0, DIR_IN, , , true),
            VFB_ARGUMENT(sint32, InInt1, DIR_IN, , , true),
            VFB_ARGUMENT(sint32, OutInt0, DIR_OUT, , , true));
```

As next step the implementation of the operation has to be provided on the server software component. The arguments to the method which implements an operation have to match the arguments specified in the VFB configuration.

```
    /**
     * RunnableEntity declarations.
     */
    Std_ReturnType ServerRunnable0(const sint8 InInt0, const sint32 InInt1,
                                   sint32& OutInt0) {
        OutInt0 = InInt0 + InInt1;
        return RTE_E_OK;
    };
```

Finally, the method still has to be linked to the OperationInvokedEvent of the communication system. Therefore a runnable entity is created which reacts on this event and executes the according method:

```
    /* **
     * Class constructor.
     */
    SWC_ExampleServer(sc_module_name nm) : SimulationSoftwareComponent(nm) {
      /* **
       * Runnable entity definitions.
       */
      VFB_RUNNABLE_OPERATION_INVOKED(SWC_ExampleServer, Port0, Operation0, ServerRunnable0);
    };
};
```

Chapter 5

Applications Using the VFB Abstraction Layer

In this chapter two demo applications are presented. They are included to support the abstract descriptions of the previous chapters with practical examples and to demonstrate the usability of the system.

5.1 Drive By Wire Replay Scenario

For a demonstration of the laboratory setup, a scenario was created which allows to control the driver of the CarMaker / AVL InMotion simulation environment via a steering wheel. The commercial steering wheel modified by the FH Joanneum includes a FlexRay communication controller and uses the FlexRay network for transmitting steering data. As receiver of the data a CarMaker / AVL InMotion hardware node is also connected to the bus. On this node, the vehicle dynamics are simulated. CarMaker / AVL InMotion includes an automatic driver which usually controls the vehicle. This automatic driver provides values for steering angle, gas, brake and gear number. The steering wheel is equipped with an accelerator, a brake and some buttons which are used to switch gear up or down. So the automatic driver can be overridden by a human driver interacting with the steering wheel. The scenario is suited perfectly for hardware-in-the-loop tests of the FlexRay controllers and the steering wheel.

The simulation output is displayed on an additional computer which is connected to the CarMaker / AVL InMotion hardware via Ethernet. The display includes graphs for various parameters, a graphic display of speedometer and tachometer and a 3D graphic model of the driving situation.

The whole scenario was reconstructed in the simulation environment. The CarMaker / AVL InMotion hardware node is replaced with a version of CarMaker / AVL InMotion running on a personal computer. An interface to the SystemC simulation environment is available, allowing the integration of CarMaker / AVL InMotion in the simulation environment. On the SystemC side, this interface consists of simple SystemC ports. To enable the data flow from the simulated FlexRay network to CarMaker / AVL InMotion an additional interface is required. A communication controller module and an *Ecu* module are used for the connection to the FlexRay bus. A software component on the *Ecu* is reacting to the reception of data on the FlexRay bus and forwards the relevant data via SystemC ports to the SystemC ports of the interface to CarMaker / AVL InMotion. The steering wheel is replaced by a replay node. This allows to monitor and record the traffic in the laboratory setup and use exactly the same data as stimulus in the simulation environment.

The steering angle sensor of the steering wheel produces many glitches. Therefore a filtering of the data is necessary before forwarding them to CarMaker / AVL InMotion. In the hardware setup the filtering is performed on the steering wheel itself with a simple exponential mean average algorithm, so the steering wheel can be characterised as smart sensor node.

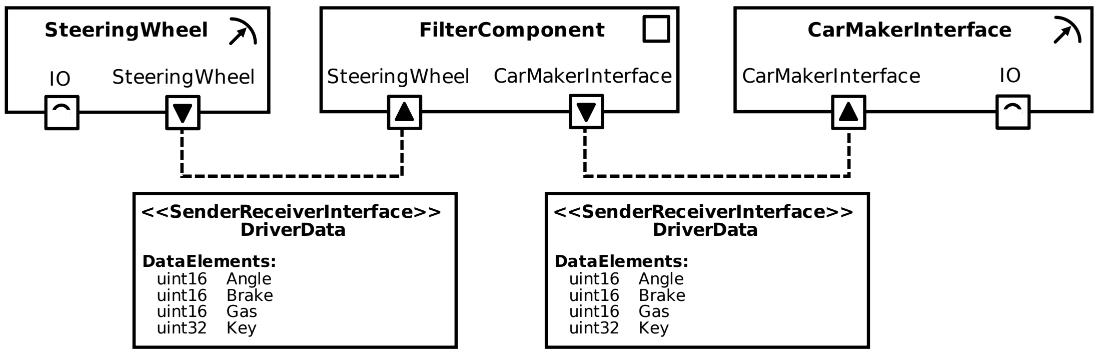


Figure 5.1: VFB view of the drive by wire scenario.

In order to compare the performance of this smart sensor node and a an external filter node the scenario described above is extended. So far the steering wheel transmitted the already filtered data. Now, in addition to the filtered data the raw data of the steering angle sensor is transmitted. In the simulation environment an additional node is included. It consists of a communication controller module an *Ecu* module and a software component which performs the filtering and transmits the filtered data again over the FlexRay bus. The data from the filter node is then forwarded to CarMaker / AVL InMotion like in the simple scenario. A VFB view of the scenario is displayed in figure 5.1. A screenshot of the simulation environment showing the assignment of software components to *Ecu*s is depicted in figure 5.2. No cable model was included resulting in ideal cable behaviour. For reference and the sake of completeness the VFB configuration code used in the software components can be found in section B.2.

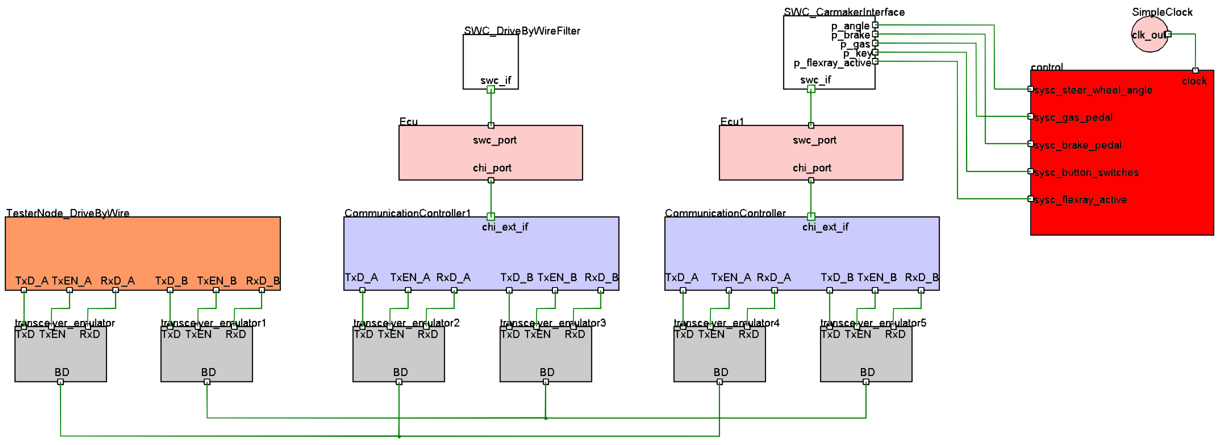
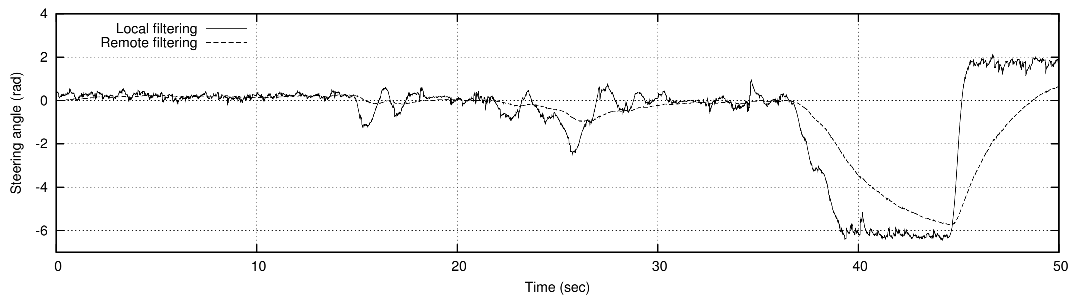


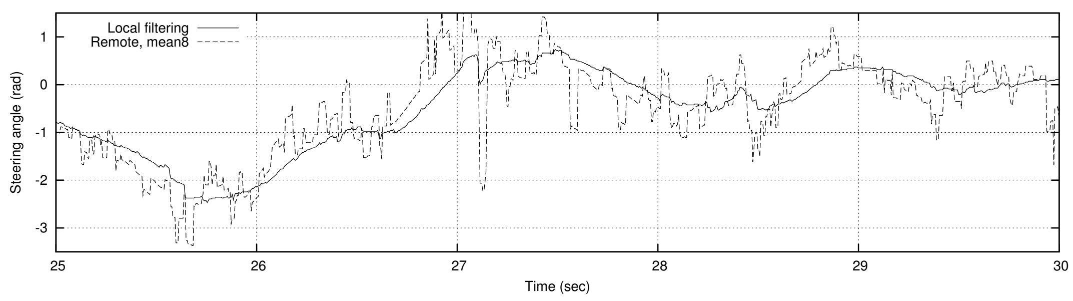
Figure 5.2: Screenshot of the simulation setup used for the Drive By Wire scenario.

The comparison showed the expected results. The filter implemented on the sensor itself can operate with a higher sampling rate. Therefore, implementing exactly the same filter algorithm on a separate filter node is far too slow and unusable. When using a filter algorithm which uses less input values, the delay becomes better but due to the high number of outliers, the signal quality significantly decreases. The differences are depicted in figure 5.3.

The benefit of using a cross-domain co-simulation platform for this analysis is to enable architecture decisions before the implementation of the system, thus saving re-design efforts. It provides great flexibility for modeling different architectures or filter concepts and illustrates the influences on the assembled system. Further details on the experiment can also be found in [15].



(a) Influence of function mapping for a given filter model.



(b) Influence of the filter implementation with respect to signal quality (outliers).

Figure 5.3: Comparison of smart sensor node and remote filtering. [15]

5.2 Automatic Windscreen Wiper Controller Scenario

This example scenario models the communication for a simple windscreen wiper which can be controlled by a rain sensor. It has two operating modes — automatic and interval mode. In automatic mode, the speed of the wiper is controlled by the rain sensor. in interval mode, the user manually specifies the speed of the wiper. Via a user interface the user can either set it to automatic mode or set an interval which translates to a speed level value for the wiper. Additionally, the user can turn on and turn off the windscreen wiper.

The scenario features the usage of the client-server communication. It consists of five software components. The *RainSensor* software component implements an operation which allows to poll the sensor value. The sensor value for the rain intensity is read from a file.

The *UserInterface* software component simulates the input of the user. The user is periodically polled for an action. Based on this action a command on the *WiperController* is executed. It is possible and suggested to simulate the user input with data read from a file. This allows unattended and faster simulation.

The *WiperController* implements the control logic. Based on the current state it polls the *RainSensor* and sends a speed value to the *Wiper* components. To demonstrate the possibility of addressing multiple receivers when using the sender-receiver communication model, two separate *Wiper* components are instantiated. The *Wiper* reads the speed signal and writes it to the screen and optionally for later analysis also to file. A VFB configuration view of the application is depicted in figure 5.4.

The application is mapped to a communication system consisting of four *Ecus*. The sensor as well as the two wipers are separate nodes. The user interface and the controller are located on the same *Ecu*. This assignment requires four signals to be transmitted via the FlexRay network. The following table lists the transmitting *Ecu*, which slot it uses for transmission and the names of the signals of each frame. The slot numbers were chosen arbitrarily.

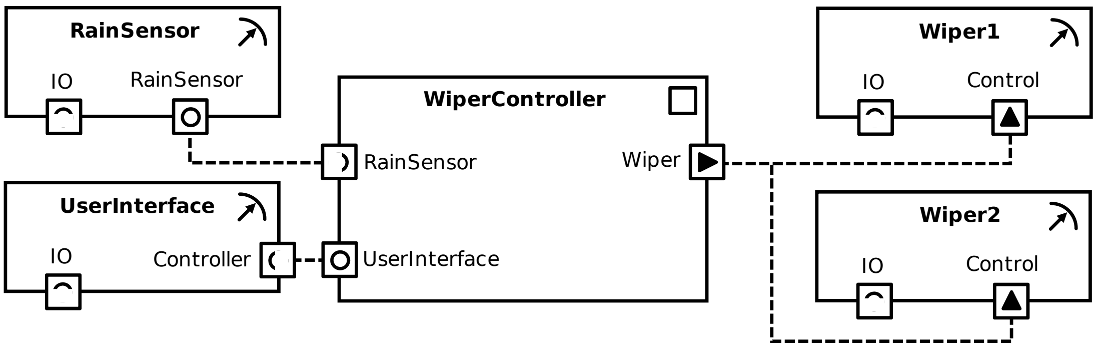


Figure 5.4: VFB view of the automatic windscreen wiper scenario.

Ecu	Tx Slot	Signal
0	7	RainSensor_GetRainAmount_response
		RainSensor_GetRainAmount_RainAmount
1	13	RainSensor_GetRainAmount_request
		Wiper_Speed
2	9	
3	11	

Table 5.1: FlexRay schedule for the automatic windscreen wiper controller scenario.

For testing the functionality of the system a simulation with artificial data for rain intensity was performed. The rain intensity is an arbitrarily chosen number without any physical meaning within the interval from 0 to 1500. The resulting speed of the windscreen wiper is bound to ten levels from 0 to 9. It also has no physical meaning. The results are depicted in figure 5.5. The diagram shows the rain intensity and the speed of the windscreen wiper. The speed levels are in the range from 0 to 9 and are scaled in the diagram from 0 to 900 to allow depicting both parameters in a single diagram. As the speed setting does depend on the current state of the user interface, the events on the user interface are denoted at the bottom of the diagram.

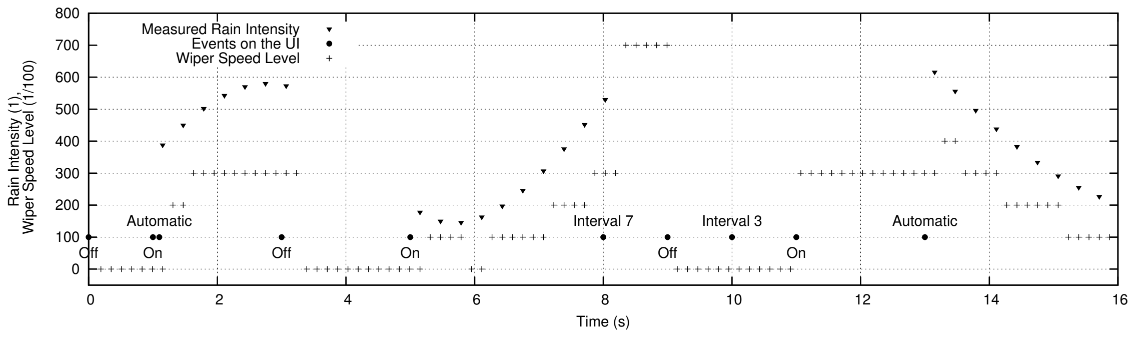
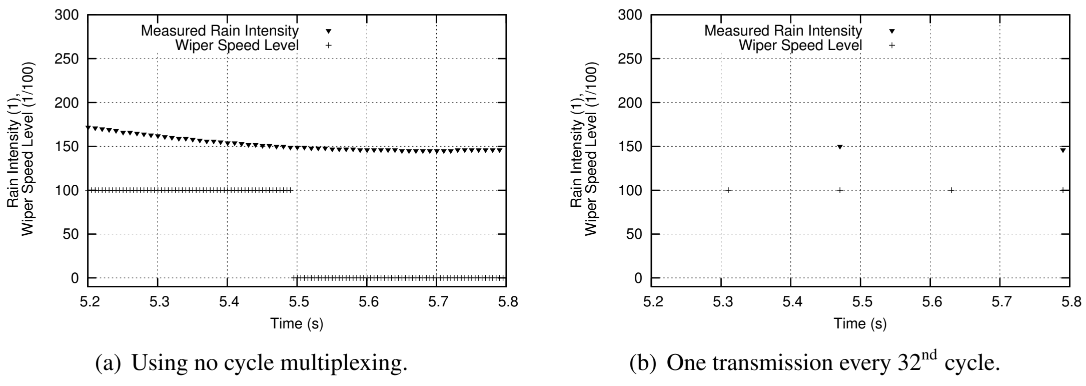


Figure 5.5: Windscreen wiper speed depending on rain intensity and settings on the user interface.

The sensor is queried for data only when the windscreen wiper is turned on and in automatic mode. This does, however, not reduce the load on the communication system as the data is transmitted in the static segment. A method to reduce the overhead would be to move the data to the dynamic segment resulting in a loss of reliability. So it is necessary to find a trade-off between dependability and efficiency. Because the rain intensity does typically not change rapidly, another option is to use cycle multiplexing, allowing the free space to be used for different data in the unused cycles and providing higher efficiency but still the dependability of the time-triggered communication approach. Figure 5.6 depicts the differences between using no cycle multiplexing and using cycle multiplexing for all frames defined in the FlexRay configuration. With cycle multiplexing enabled, they are configured to be transmitted every 32nd cycle only.



(a) Using no cycle multiplexing. (b) One transmission every 32nd cycle.

Figure 5.6: Influence of cycle multiplexing on the responsiveness of the system.

Using the slot configuration from table 5.1 above, the delay times from the query for the rain intensity and the transmission of the wiper speed can be calculated. Without using cycle multiplexing, the request for the *RainSensor* to transmit the rain intensity is transmitted in cycle 0, slot 13. The response from the *RainSensor* is received by the controller in cycle 1, slot 7. In cycle 1 slot 13, the wiper speed value depending on the last received rain intensity value is transmitted to the *Wipers*. This results in an effective delay of one cycle.

When cycle multiplexing is involved as described above, the response by the to a *RainSensor* request sent in cycle 0, slot 13 is transmitted in cycle 32, slot 7. So in cycle 32 slot 13 the calculated wiper speed value can be sent to the *Wipers* leading to a delay of 32 cycles. In the configuration chosen for performing this experiment, the cycle time was configured to 5 ms. So a delay of 32 cycles matches 160 ms. This probably is no problem for a rain sensor.

An optimisation to reduce delays is possible. By splitting the transmission of the rain intensity polling request and the wiper speed value to two different frames and carefully choosing the slots it would be possible to transmit the calculated wiper speed value in the same cycle as the request to the *RainSensor* resulting in a delay of less than one cycle. This, however, reduces the maximum age of the data, but does not influence the sampling rate for the rain intensity which still would be once per cycle and every 32nd cycle when using cycle multiplexing respectively.

Chapter 6

Conclusion and Future Work

This chapter contains a conclusion of the important points of the thesis. Finally, the last section of this chapter presents directions for further enhancements of the presented work.

6.1 Conclusion

This thesis deals with the development of a concept for a VFB abstraction layer for a FlexRay co-simulation environment. In the course of giving overview of related work, main aspects of the co-simulation environment are explained. Therefore also an introduction to the relevant standards — FlexRay and AUTOSAR is included. A survey of existing approaches on software simulation on different abstraction layers, from detailed instruction set simulation to fast abstract functional simulation, is given and tools and existing models used for in the co-simulation environment are mentioned including their features and value for the co-simulation environment.

The main part of the thesis presents the developed concept. The object oriented VFB abstraction layer provides an easy and straightforward way of developing distributed applications for a co-simulation environment. The communication between different parts of the application is independent from the communication system because the abstraction layer makes the communication mechanisms transparent to the application developer. The introduced abstraction layer provides interfaces like described in the AUTOSAR specification of the VFB to the application developer allowing for easy porting of AUTOSAR compliant applications to the simulation environment. Finally, example applications were developed to demonstrate the functionality of the system.

6.2 Future Work

The purpose of this thesis was to illustrate a concept how to integrate an abstract VFB layer into an existing co-simulation environment. Being a concept implementation only the developed abstraction layer still provides much room for improvements. This section mentions ideas for further enhancements.

6.2.1 Cover Other Aspects of the AUTOSAR Standard

For now, the VFB abstraction layer deals mainly with the task of communication between software components. To further improve the compatibility to AUTOSAR it would be necessary to extend the implementation to cover other aspects of the AUTOSAR specification. Examples for other system parts specified by AUTOSAR are memory management or operating system services. As the AUTOSAR specification is rather large and complex, and only a small part has been covered, there certainly is still enough work for other thesis.

6.2.2 Tool Supported Configuration

Currently the configuration of the VFB is done by manually editing header files. This process is error-prone and tedious. Therefore a possibility to extract the VFB configuration data from AUTOSAR system description files, similar to the generation step proposed as methodology by AUTOSAR, would enhance the usability.

The AUTOSAR configuration format is a rather complex Extensible Markup Language (XML) format. For extracting the VFB configuration, only a subset of all configuration parameters would be needed. However, an XML parser to read the file would be required and writing a tool to generate code from the configuration files still poses a significant amount of work.

6.2.3 Inclusion of Timing Annotations for Application Code

The presented model does not include timing annotations for the run-time of application code. To gain further insight on the application behaviour the model could be extended to allow for a timed simulation of the application code. By executing all tasks in the context of a single processing unit, first steps in this direction were already taken, but work is still needed to fully support the annotation of timing information.

6.2.4 Extension to Other Bus Systems

A modification to allow the usage of other bus systems than FlexRay would be an interesting extension. AUTOSAR supports in addition to FlexRay also CAN and LIN. However, much effort would be needed, as there exist no models for communication controllers of those bus systems which could easily be integrated. Also on the VFB abstraction layer major changes would be required, making this an rather laborious task.

Appendix A

VFB Configuration Macros

This chapter describes the macros used for the configuration of the VFB. The configuration macros define the interface which is necessary to be known to the application developer in order to use the VFB abstraction layer. Therefore, this chapter serves as reference to the application developer and explains in great detail all of the implemented configuration macros.

For sending data, the macro ***VFB_SEND_DATAELEMENT*** defines a data element which gets sent via a sender port. It has the signature:

```
#define VFB_SEND_DATAELEMENT(_PORT_NAME, _DATAELEMENT_NAME,
        _QUEUE_LENGTH, _SIGNAL_NAME, _DATATYPE, _BYTE_ORDER)
```

The following parameters are required:

_PORT_NAME: Indicates the port the data element belongs to. It is used for the generation of RTE API methods. The port name must be given as unquoted string. If the port does not exist, it is created.

_DATAELEMENT_NAME: Represents the name of the data element. It is used for the generation of RTE API methods. The data element name must be passed as unquoted string.

_QUEUE_LENGTH: Sets the length of the queue for data elements which use the queued transmission mode. If unqueued transmission is desired, this parameter has to be set to 1.

_SIGNAL_NAME: Maps the data element to a signal. The signal name must be passed as unquoted string. If the signal name is left empty, a default signal name is constructed from the port and data element name. This signal name must be defined in the FlexRay configuration if the communication is supposed to be performed via the communication network. Otherwise, the communication is performed locally only.

_DATATYPE: Provides the data type for the data. It is required for the RTE API generation. The type must be defined and valid within the software component.

_BYTE_ORDER: Specifies the network byte order for the data element (see 4.3.5). Valid values are *END_LITTLE_ENDIAN* and *END_BIG_ENDIAN*.

On the receiver side, the macro ***VFB_RECEIVE_DATAELEMENT*** defines a data element which can receive data via a receiver port. It has the signature:

```
#define VFB_RECEIVE_DATAELEMENT(_PORT_NAME, _DATAELEMENT_NAME,
        _QUEUE_LENGTH, _BLOCKING, _TIMEOUT, _SIGNAL_NAME, _DATATYPE,
        _SIGN_EXTENSION, _INIT_VALUE, _BYTE_ORDER)
```

The following parameters are required:

_PORT_NAME: Indicates the port the data element belongs to. It is used for the generation of RTE API methods. The port name must be given as unquoted string. If the port does not exist, it is created.

_DATAELEMENT_NAME: Represents the name of the data element. It is used for the generation of RTE API methods. The data element name must be passed as unquoted string.

_QUEUE_LENGTH: Sets the length of the queue for data elements which use the queued transmission mode. Requires an integer value. If unqueued transmission is desired, this parameter has to be set to 1.

_BLOCKING: If the queued transmission mode is used, the data element can be received in a blocking or a non blocking way. If this boolean parameter is set to true, the call to the RTE API method for reading the data is blocking. Otherwise a non-blocking receive is performed. When there is no data available in case of a non-blocking receive, an error is returned by the RTE API method.

_TIMEOUT: This parameter is used to specify a timeout for a blocking receive. The value is interpreted as time in seconds and may be specified as integer or float/double value.

_SIGNAL_NAME: Maps the data element to a signal. The signal name must be passed as unquoted string. If the signal name is left empty, a default signal name is constructed from the port and data element name. This signal name must be defined in the FlexRay configuration if the communication is supposed to be performed via the communication network. Otherwise, the communication is performed locally only.

_DATATYPE: Provides the data type for the data. It is required for the RTE API generation. The type must be defined and valid within the software component.

_SIGN_EXTENSION: Specifies whether sign extension should be performed when receiving the data element (see 4.4.2.4).

_INIT_VALUE: If a runnable entity tries to receive a data element before the communication system is ready, this value is returned by the call to the RTE API method for the receive. The specified value has to be implicitly convertible to the type specified with the *_DATATYPE* parameter.

_BYTE_ORDER: Specifies the network byte order for the data element (see 4.3.5). Valid values are *END_LITTLE_ENDIAN* and *END_BIG_ENDIAN*.

Another macro called ***VFB_CLIENT_OPERATION*** is used for the definition of an operation on the client side of a client-server interface. The operation is invokable from a runnable entity by calling the respective RTE API method. The signature for the macro is:

```
#define VFB_CLIENT_OPERATION(_PORT_NAME, _OPERATION_NAME,
        _ASYNCHRONOUS, _TIMEOUT, _REQUEST_SIGNAL_NAME,
        _RESPONSE_SIGNAL_NAME, _BYTE_ORDER, _ARGC, ...)
```

For the generation the following parameters are required:

_PORT_NAME: Indicates the port the operation belongs to. It is used for the generation of RTE API methods. The port name must be given as unquoted string. If the port does not exist, it is created.

_OPERATION_NAME: Represents the name of the operation. It is used for the generation of RTE API methods. The operation name must be passed as unquoted string.

_ASYNCHRONOUS: The execution of an operation can be performed synchronously or asynchronously. If the boolean parameter *_ASYNCHRONOUS* is set to true, a call to the RTE API function immediately returns. Otherwise, the call returns after a response has been received from the server or a timeout has occured. In case of asynchronous execution, the client can react to the *AsynchronousServerCallReturnsEvent* of the *Operation* object.

_TIMEOUT: In case of a synchronous execution of an operation, this parameter specifies how long to wait in the RTE API method for a response from the server. After the timeout has passed, the control is returned to the caller with a timeout error. The parameter value is interpreted as time in seconds and may be specified as integer or float/double value.

_REQUEST_SIGNAL_NAME, *_RESPONSE_SIGNAL_NAME*: Provides a signal name for the request and response signals of the operation (see 4.3.10). The signal name must be passed as unquoted string. If the signal name is left empty, a default signal name is constructed from the port and operation name. This signal name must be defined in the FlexRay configuration if the communication is supposed to be performed via the communication network. Otherwise, the communication is performed locally only.

_BYTE_ORDER: Specifies the network byte order for the operation (see 4.3.5). All arguments of the operation use the same byte order. Valid values are *END_LITTLE_ENDIAN* and *END_BIG_ENDIAN*.

_ARGC: As a operation can have a variable number of arguments, this parameter is used to indicate how many arguments for the operation are specified. This parameter must be an integer literal.

`...`: Finally, a list of arguments can be passed to this macro. The argument list defines the arguments for the operation and is created by using the *VFB_ARGUMENT* macro for every argument to specify.

On the server side the respective operation must be implemented for this purpose the macro ***VFB_-SERVER_OPERATION*** defines an operation on the server side of a client-server interface. The operation can later be defined to be invoked upon the reception of an *OperationInvokedEvent*. The signature for the macro is:

```
#define VFB_SERVER_OPERATION(_PORT_NAME, _OPERATION_NAME,
        _RUNNABLE_NAME, _REQUEST_SIGNAL_NAME, _RESPONSE_SIGNAL_NAME,
        _BYTE_ORDER, _ARGC, ...)
```

For the generation of the methods handling the invocation the following parameters are required:

_PORT_NAME: Indicates the port the operation belongs to. The port name must be given as unquoted string. If the port does not exist, it is created.

_OPERATION_NAME: Represents the name of the operation. The port name and the operation name are used to identify an operation. The operation name must be passed as unquoted string.

_RUNNABLE_NAME: When the client requests the execution of an operation on the server, on the server a runnable entity has to be executed on the server. This parameter is the name of the runnable entity on the server. It must be the name of a method of the software component. The arguments of the method must match the arguments which are specified for the operation. The runnable entity name must be passed as unquoted string.

_REQUEST_SIGNAL_NAME, *_RESPONSE_SIGNAL_NAME*: Provides a signal name for the request and response signals of the operation (see 4.3.10). The signal name must be passed as unquoted string. If the signal name is left empty, a default signal name is constructed from the port and operation name. This signal name must be defined in the FlexRay configuration if the communication is supposed to be performed via the communication network. Otherwise, the communication is performed locally only.

_BYTE_ORDER: Specifies the network byte order for the operation (see 4.3.5). All arguments of the operation use the same byte order. Valid values are *END_LITTLE_ENDIAN* and *END_BIG_ENDIAN*.

_ARGC: As a operation can have a variable number of arguments, this parameter is used to indicate how many arguments for the operation are specified. This parameter must be an integer literal.

`...`: Finally, a list of arguments can be passed to this macro. The argument list defines the arguments for the operation and is created by using the *VFB_ARGUMENT* macro for every argument to specify.

For the specification of arguments to operations for client-server interfaces, an additional macro ***VFB_ARGUMENT*** is used. Its signature is

```
#define VFB_ARGUMENT(_TYPE, _NAME, _DIRECTION, _SIGNAL_NAME_IN,
        _SIGNAL_NAME_OUT, _SIGN_EXTENSION)
```

It is invoked from within the operation definition macro and requires the following parameters:

_TYPE: Specifies the datatype for the argument. The value of this parameter must be defined and valid within the software component.

_NAME: Gives a descriptive name for the argument. This is used for the identification of arguments by name in the *Operation* object. The argument name must be specified as unquoted string.

_DIRECTION: Specifies the direction for the argument. Valid values for this parameter are:

DIR_IN: The argument passes data to the operation.

DIR_OUT: The argument returns data from the operation.

DIR_INOUT: The argument is used for input and output.

_SIGNAL_NAME_IN, *_SIGNAL_NAME_OUT*: Maps the argument to a signal. Depending on the direction of the argument, one or both signal names are used. The signal name must be passed as unquoted string. If the signal name is left empty, a default signal name is constructed from the port, operation and argument name and the direction information. This signal name must be defined in the FlexRay configuration if the communication is supposed to be performed via the communication network. Otherwise, the communication is performed locally only.

_SIGN_EXTENSION: Specifies whether sign extension should be performed when receiving the argument data from the communication system (see 4.4.2.4).

The macro ***VFB_RUNNABLE_PERIODIC*** defines a runnable entity which is invoked periodically. The following parameters are needed for creating this type of runnable entity:

_SOFTWARE_COMPONENT_TYPE: The class name of the software component which the runnable entity belongs to. The parameter must be specified as unquoted string.

_PERIOD: Sets the period for the invocation of the runnable entity.

_METHOD_NAME: This is the name of the method of the software component which is used as runnable entity and executed periodically. The method must be publicly accessible and have neither arguments nor a return value.

Similar to the periodic variant, the macro ***VFB_RUNNABLE_TIMEOUT*** defines a runnable entity, which is executed only once after a given timeout. This is an implementation specific runnable variant which is not defined by AUTOSAR. The following parameters are needed for the creation:

_SOFTWARE_COMPONENT_TYPE: The class name of the software component which the runnable entity belongs to. The parameter must be specified as unquoted string.

_TIMEOUT: Sets the timeout for the invocation of the runnable entity.

_METHOD_NAME: This is the name of the method of the software component which is used as runnable entity and executed after the timeout. The method must be publicly accessible and have neither arguments nor a return value.

For reaction on events triggered by data elements, three different macros exist. On the sending side, the macro ***VFB_RUNNABLE_SEND_COMPLETE*** allows to define a runnable entity which gets executed when data was completely passed to the communication system. The macros ***VFB_RUNNABLE_DATA_RECEIVED*** and ***VFB_RUNNABLE_DATA_RECEIVED_ERROR*** are available on the receiving side. The defined runnable entities are executed when data was successfully received or an error occured when trying to receive data respectively. The macro takes the following parameters:

_SOFTWARE_COMPONENT_TYPE: The class name of the software component which the runnable entity belongs to. The parameter must be specified as unquoted string.

_PORT_NAME: Gives the name of the port to which the data element belongs.

_DATAELEMENT_NAME: Specifies the name of the data element.

_METHOD_NAME: This is the name of the method of the software component which is used as runnable entity and executed when the respective event occurs for the specified data element. The method must be publicly accessible and have neither arguments nor a return value.

To react to events of client-server ports, the macros ***VFB_RUNNABLE_OPERATION_INVOKED*** and ***VFB_RUNNABLE_ASYNCHRONOUS_SERVER_CALL_RETURNS*** provide the required interface. The created runnable entity is executed on the server side whenever a request for execution was received from a client and on the client side when the response from the server was transmitted back from the server respectively. The following parameters are needed to create a runnable entity for a client-server interface:

_SOFTWARE_COMPONENT_TYPE: The class name of the software component which the runnable entity belongs to. The parameter must be specified as unquoted string.

_PORT_NAME: Gives the name of the port to which the data element belongs.

_OPERATION_NAME: Specifies the name of the operation.

_METHOD_NAME: This is the name of the method of the software component which is used as runnable entity and executed when the respective event occurs for the specified data element. The method must be publicly accessible. On the server side, the runnable entity must also have been specified in the operation definition macro *VFB_SERVER_OPERATION* as *_RUNNABLE_NAME*. Also the arguments for the method must match the argument list given for the operation definition. On the client side, the method must have neither arguments nor a return value.

Appendix B

Source Code Excerpts

Having a look at the source code sometimes can give more insight than an extensive explanation in words. Therefore this chapter provides selected excerpts of the source code which are used for the illustration of various aspects of the thesis.

B.1 Simple Sender Receiver Scenario

This section contains the code for the example scenario explained in section 4.5. The code for the used software components as well a full FlexRay configuration are given below.

```
class SWC_ExampleSender : public SimulationSoftwareComponent {
  public:
    /* **
     * Port API definitions.
     */
    VFB_SEND_DATAELEMENT(Port0, DataElement0, 1, Port0_DataElement0,
              int, END_LITTLE_ENDIAN);
    VFB_SEND_DATAELEMENT(Port0, DataElement1, 1, Port0_DataElement1,
              sint8, END_LITTLE_ENDIAN);

    /* **
     * RunnableEntity declarations.
     */
    void SenderRunnable0() {
      int data = 0x01020304;
      Rte_Send_Port0_DataElement0(data);
    };

    /* **
     * Class constructor.
     */
    SWC_ExampleSender(sc_module_name nm) : SimulationSoftwareComponent(nm) {
      /* **
       * Runnable entity definitions.
       */
      VFB_RUNNABLE_PERIODIC(SWC_ExampleSender, 0.0052, SenderRunnable0);
    };
};
```

Listing B.1: Implementation of a simple software component sending data via a sender port.

```
class SWC_ExampleReceiver : public SimulationSoftwareComponent {
  public:
    /* **
     * Port API definitions.
     */
    VFB_RECEIVE_DATAELEMENT(Port0, DataElement0, 1, false, 0,
              Port0_DataElement0, int, true, 0, END_LITTLE_ENDIAN);
    VFB_RECEIVE_DATAELEMENT(Port0, DataElement1, 1, false , 0,
              Port0_DataElement1, sint8, true , 0, END_LITTLE_ENDIAN);

    /* **
     * RunnableEntity declarations.
     */
    void ReceiverRunnable0() {
      int data;
      Rte_Read_Port0_DataElement0(data);
      cout << " Received Data: " << data << endl;
    };

    /* **
     * Class constructor.
     */
    SWC_ExampleReceiver(sc_module_name nm) : SimulationSoftwareComponent(nm) {
        /**
         * Runnable entity definitions.
         */
      VFB_RUNNABLE_DATA_RECEIVED(SWC_ExampleReceiver, Port0,
              DataElement0, ReceiverRunnable0);
    };
};
```

Listing B.2: Implementation of a simple software component receiving data via a receiver port.

For the sake of completeness, once also a complete FlexRay configuration should be given. The listing below represents the FlexRay configuration for the simple sender-receiver example from section4.5.

```
const SIGNAL SIGNALlist[] =
{
    {                         // Signal Idx: 0
        "Port0_DataElement0",     //Name of the signal
        0,                        //Starting position (in bits)
        32,                       //Length of the signal (in bits)
    }
  ,
    {                         // Signal Idx: 1
        "Port0_DataElement1",     //Name of the signal
        32,                       //Starting position (in bits)
        8,                        //Length of the signal (in bits)
    }
};

const FRAME FRAMElist[] =
{
    {                         // Frame Idx: 0
        4,                        //slot_id
        1,                        //repetition
        0,                        //base
        5,                        //framelen
        'A',                      //flexray channel
        1,                        //number of signals in this frame
        &SIGNALlist[0]            //a pointer to the signals
    }
    ,
    {                         // Frame Idx: 1
        7,                        //slot_id
        1,                        //repetition
        0,                        //base
        0,                        //framelen
        'A',                      //flexray channel
        0,                        //number of signals in this frame
        NULL                      //a pointer to the signals
    }
};

const ECU ECUlist[] =
{
    {
        "Sender Node",            //Name
        0.025000,                   //Microtick
        1,                          //StartUpSync
        16,                         //MaxDynamicPayloadLength
        1,                          //ClusterDriftDamping
        36,                         //DecodingCorrection
        401202,                     //ListenTimeout
        601,                        //MaxDrift
        0,                          //ExternOffsetCorrection
        0,                          //ExternRateCorrection
        271,                        //LatestTX
        200000,                     //MicroPerCycle
        259,                        //OffsetCorrectionOut
        601,                        //RateCorrectionOut
        2,                          //SamplesPerMicrotock
        10,                         //DelayCompensation A
        10,                         //DelayCompensation B
        63,                         //WakeUpPattern B
        1,                          //AllowHaltDueToClock
        0,                          //AllowPassiveToActive
        109,                        //AcceptedStartupRange
        8,                          //MacroInitialOffset A
        8,                          //MacroInitialOffset B
        34,                         //MicroInitialOffset A
        34,                         //MicroInitialOffset B
        0,                          //SingleSlotEnabled
        3,                          //KeySlotUsage
        0,                          //WakeupChannel
        1,                          //Number of frames trasmitted by this ECU
        &FRAMElist[0],              //A pointer to the correct frames
        4,                          //KeySlotId
        1,                          //KeySlotUsedForStartup
        1,                          //KeySlotUsedForSync
    }
  ,
    {
        "Receiver Node",            //Name
        0.025000,                   //Microtick
        5,                          //StartUpSync
        16,                         //MaxDynamicPayloadLength
        1,                          //ClusterDriftDamping
        36,                         //DecodingCorrection
        401202,                     //ListenTimeout
        601,                        //MaxDrift
        0,                          //ExternOffsetCorrection
        0,                          //ExternRateCorrection
        271,                        //LatestTX
        200000,                     //MicroPerCycle
        259,                        //OffsetCorrectionOut
        601,                        //RateCorrectionOut
        2,                          //SamplesPerMicrotock
        10,                         //DelayCompensation A
        10,                         //DelayCompensation B
        63,                         //WakeUpPattern B
        1,                          //AllowHaltDueToClock
        0,                          //AllowPassiveToActive
        109,                        //AcceptedStartupRange
        8,                          //MacroInitialOffset A
        8,                          //MacroInitialOffset B
        34,                         //MicroInitialOffset A
        34,                         //MicroInitialOffset B
        0,                          //SingleSlotEnabled
        3,                          //KeySlotUsage
        0,                          //WakeupChannel
        1,                          //Number of frames trasmitted by this ECU
        &FRAMElist[1],              //A pointer to the correct frames
        7,                          //KeySlotId
        1,                          //KeySlotUsedForStartup
        1,                          //KeySlotUsedForSync
    }
};

const FLEXRAY_CLUSTER testcluster =
{
    0.100000,                   //Bit
    6,                          //TSSTransmitter
    5000,                       //Cycle
    40,                         //StaticSlot
    80,                         //NumberOfStaticSlots
    279,                        //NumberOfMinislots
    6,                          //Minislot
    0,                          //SymbolWindow
    8,                          //PayloadLengthStatic
    6,                          //ActionPointOffset
    3,                          //MinislotActionPointOffset
    1,                          //DynamicSlotIdlePhase
    10,                         //ColdStartAttempts
    123,                        //NIT
    0.012500,                   //SampleClockPeriod
    59,                         //WakeupSymbolRxIdle
    55,                         //WakeupSymbolRxLow
    301,                        //WakeupSymbolRxWindow
    180,                        //WakeupSymbolTxIdle
    60,                         //WakeupSymbolTxLow
    2,                          //ListenNoise
    5000,                       //MacroPerCycle
    1.000000,                   //MacroTick
    0,                          //MaxInitializationError
    2,                          //MaxWithoutClockCorrectionFatal
    2,                          //MaxWithoutClockCorrectionPassive
    2,                          //NetworkManagementVectorLength
    4987,                       //OffsetCorrectionStart
    15,                         //SyncNodeMax
    81,                         //CasRxLowMax
    29,                         //CasRxLowMin
    1.000000,                   //ClusterDriftDamping
    6.449000,                   //OffsetCorrectionMax
    2,                          //Number of ECUs in the cluster
    ECUlist                     //Pointer to the ECU structures
};
```

Listing B.3: Complete FlexRay cluster configuration for a simple sender-receiver scenario.

B.2 VFB Configuration of the Drive By Wire Software Components

This section holds the VFB configuration code for the software components used in the drive by wire scenario described in section 5.1. It includes a receiving and a sending port.

```
class SWC_DriveByWireFilter : public SimulationSoftwareComponent {
  public:
    /* **
     * Port API definitions.
     */
    VFB_RECEIVE_DATAELEMENT(SteeringWheel, Angle, 1, false , 0,
        Lenkrad_Angle, uint16, false , 0, END_BIG_ENDIAN);
    VFB_RECEIVE_DATAELEMENT(SteeringWheel, AngleUnfiltered, 1, false , 0,
        Lenkrad_Angle_Unfiltered, uint16, false , 0, END_BIG_ENDIAN);
    VFB_RECEIVE_DATAELEMENT(SteeringWheel, Brake, 1, false , 0,
        Lenkrad_Brake, uint16, false , 0, END_BIG_ENDIAN);
    VFB_RECEIVE_DATAELEMENT(SteeringWheel, Gas, 1, false , 0,
        Lenkrad_Gas, uint16, false , 0, END_BIG_ENDIAN);
    VFB_RECEIVE_DATAELEMENT(SteeringWheel, Key, 1, false , 0,
        Lenkrad_Key, uint32, false , 0, END_BIG_ENDIAN);

    VFB_SEND_DATAELEMENT(CarMakerInterface, Angle, 1,
        Filtered_Angle, uint16, END_BIG_ENDIAN);
    VFB_SEND_DATAELEMENT(CarMakerInterface, Brake, 1,
        Filtered_Brake, uint16, END_BIG_ENDIAN);
    VFB_SEND_DATAELEMENT(CarMakerInterface, Gas, 1,
        Filtered_Gas, uint16, END_BIG_ENDIAN);
    VFB_SEND_DATAELEMENT(CarMakerInterface, Key, 1,
        Filtered_Key, uint32, END_BIG_ENDIAN);

    /* **
     * RunnableEntity declarations.
     */
    void FilterRunnable();

    /* **
     * Class constructor.
     */
    SWC_DriveByWireFilter(sc_module_name nm) : SimulationSoftwareComponent(nm) {
      /* **
       * Event definitions.
       */
      VFB_RUNNABLE_DATA_RECEIVED(SWC_DriveByWireFilter,
          SteeringWheel, Angle, FilterRunnable);
    };
};
```

Listing B.4: VFB Configuration of the filter software component used in the drive by wire scenario.

B.3 VFB Configuration of the RainSensorController

For demonstration of the configuration of the VFB for a software component which uses the client-server communication mechanism, the header file for the RainSensorController mentioned in section 5.2 is given in the following listing. It includes the configuration of the VFB with the declaration of operations for a server port and also features the generation of default signal names.

```
class SWC_WiperController : public SimulationSoftwareComponent {
  public:

    /* **
     * Port API definitions.
     */
    VFB_SERVER_OPERATION(Controller, TurnOn, TurnOn, , , END_LITTLE_ENDIAN,0);
    VFB_SERVER_OPERATION(Controller, TurnOff, TurnOff, , , END_LITTLE_ENDIAN,0);
    VFB_SERVER_OPERATION(Controller, SetAutomatic, SetAutomatic, , , END_LITTLE_ENDIAN,0);
    VFB_SERVER_OPERATION(Controller, SetInterval, SetInterval, , , END_LITTLE_ENDIAN,1,
        VFB_ARGUMENT(uint8, Interval, DIR_IN, Controller_SetInterval_Interval, , false));

    VFB_CLIENT_OPERATION(RainSensor, GetRainAmount, false , 0,
            RainSensor_GetRainAmount_request,
            RainSensor_GetRainAmount_response, END_LITTLE_ENDIAN, 1,
        VFB_ARGUMENT(uint16, RainAmount, DIR_OUT, ,
                RainSensor_GetRainAmount_RainAmount, false));

    VFB_SEND_DATAELEMENT(Wiper, Speed, 1, Wiper_Speed, uint8, END_LITTLE_ENDIAN);

    /* **
     * RunnableEntity declarations.
     */
    Std_ReturnType TurnOn();
    Std_ReturnType TurnOff();
    Std_ReturnType SetAutomatic();
    Std_ReturnType SetInterval(uint8 interval);

    void PollRainSensor();

    /* **
     * Class constructor.
     */
    SWC_WiperController(sc_module_name nm) : SimulationSoftwareComponent(nm) {

      /* **
       * Event definitions.
       */
      VFB_RUNNABLE_OPERATION_INVOKED(SWC_WiperController, Controller,
            TurnOn, TurnOn);
        VFB_RUNNABLE_OPERATION_INVOKED(SWC_WiperController, Controller,
            TurnOff, TurnOff);
        VFB_RUNNABLE_OPERATION_INVOKED(SWC_WiperController, Controller,
            SetAutomatic, SetAutomatic);
        VFB_RUNNABLE_OPERATION_INVOKED(SWC_WiperController, Controller,
            SetInterval, SetInterval);

        VFB_RUNNABLE_PERIODIC(SWC_WiperController, 0.00503, PollRainSensor);
    };
};
```

Listing B.5: VFB Configuration of the controller software component used in the rain sensor scenario.

Appendix C

Acronyms

API	Application Programming Interface
AUTOSAR	AUTomotive Open System ARchitecture
CAN	Controller Area Network
CHI	Controller Host Interface
CRC	Cyclic Redundancy Check Code
DIO	Digital Input Output
ECU	Electronic Control Unit
FIBEX	Field Bus Exchange Format
GDB	GNU Project Debugger
GUI	Graphical User Interface
IDE	Integrated Development Environment
ISS	Instruction Set Simulator
LIN	Local Interconnect Network
MOST	Media Oriented Systems Transport
OSEK	Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen (English: Open Systems and their Interfaces for the Electronics)
PCI	Protocol Control Information
PDU	Protocol Data Unit
RTE	Run-Time Environment
RTOS	Real-Time Operating System
SDU	Service Data Unit
SLDL	System Level Description Language
TDMA	Time Division Multiple Access

TEODACS	Test, Evaluation and Optimization of Dependable Automotive Communication Systems. TEODACS is the name of the research project that is ongoing at the ViF and deals with the analysis of the FlexRay communication system.
TLM	Transaction Level Modelling
VHDL	VHSIC Hardware Description Language
VHDL-AMS	VHDL with Analogue and Mixed-Signal extensions
VHSIC	Very-High-Speed Integrated Circuit
ViF	Kompetenzzentrum Das virtuelle Fahrzeug (English: Virtual Vehicle Competence Center)
VFB	Virtual Function Bus
XML	Extensible Markup Language

Bibliography

[1] *ISO 11898 Standard — Controller Area Network*. Techn. rep., International Organisation for Standardization, 2003.

[2] *FlexRay Communications System - Protocol Specification, v2.1 Revision A*. Techn. rep., FlexRay Consortium, 2005.

[3] *IEEE 1666 Standard System C Language Reference Manual*. Techn. rep., Open SystemC Intitiative, 2005.

[4] *LIN Specification Package, Revision 2.1*. Techn. rep., LIN Consortium, 2006.

[5] *AUTOSAR - Layered Software Architecture, v2.2.2*. Techn. rep., AUTOSAR GbR, August 2008.

[6] *AUTOSAR - Requirements on FlexRay, v2.0.5*. Techn. rep., AUTOSAR GbR, August 2008.

[7] *AUTOSAR - Specification of Communication, v3.0.3*. Techn. rep., AUTOSAR GbR, August 2008.

[8] *AUTOSAR - Specification of FlexRay Driver, v2.2.2*. Techn. rep., AUTOSAR GbR, August 2008.

[9] *AUTOSAR - Specification of FlexRay Interface, v3.0.3*. Techn. rep., AUTOSAR GbR, August 2008.

[10] *AUTOSAR - Specification of FlexRay Transceiver Driver, v1.2.3*. Techn. rep., AUTOSAR GbR, August 2008.

[11] *AUTOSAR - Specification of the RTE, v2.0.1*. Techn. rep., AUTOSAR GbR, June 2008.

[12] *AUTOSAR - Specification of the Virtual Function Bus, v1.0.2*. Techn. rep., AUTOSAR GbR, August 2008.

[13] *AUTOSAR - Technical Overview, v2.2.2*. Techn. rep., AUTOSAR GbR, August 2008.

[14] *MOST Specification, Rev 3.0*. Techn. rep., MOST Cooperation, May 2008.

[15] Armengaud, E., M. Karner, C. Steger, R. Weiss, M. Pistauer, and F. Pfister: *A Cross Domain Co-Simulation Platform for the Efficient Analysis of Mechatronic Systems (to be published)*. In *SAE '10: SAE World Congress & Exhibition*, pp. 1–14, April 2010.

[16] Armengaud, E., D. Watzenig, C. Steger, H. Berger, H. Gall, F. Pfister, and M. Pistauer: *TEODACS: A new Vision for Testing Dependable Automotive Communication Systems*. In *SIES '08: Proceedings of the international symposium on industrial embedded systems*, pp. 257–260, June 2008.

[17] Becker, M., H. Zabel, W. Müller, and U. Kiffmeier: *Integration abstrakter RTOS-Simulation in den Entwurf eingebetteter automobiler E/E-Systeme*. In *MBMV '09: Proceedings of the workshop on Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, March 2009.

[18] Braun, G., A. Nohl, A. Hoffmann, O. Schliebusch, R. Leupers, and H. Meyr: *A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation.* Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 23(12):1625–1639, Dec. 2004, ISSN 0278-0070.

[19] Broy, J. and K. Muller-Glaser: *The Impact of Time-triggered Communication in Automotive Embedded Systems.* pp. 353–356, July 2007.

[20] Broy, M.: *Challenges in Automotive Software Engineering.* In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pp. 33–42, New York, NY, USA, 2006. ACM, ISBN 1-59593-375-1.

[21] Cai, L. and D. Gajski: *Transaction Level Modeling: An Overview.* In *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pp. 19–24, October 2003.

[22] Donlin, A.: *Transaction Level Modeling: Flows and Use Models.* In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pp. 75–80, New York, NY, USA, 2004. ACM, ISBN 1-58113- 937-3.

[23] Formaggio, L., F. Fummi, and G. Pravadelli: *A Timing-Accurate HW/SW Co-simulation of an ISS with SystemC.* In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pp. 152–157, New York, NY, USA, 2004. ACM, ISBN 1-58113- 937-3.

[24] Fummi, F., S. Martini, G. Perbellini, and M. Poncino: *Native ISS-SystemC Integration for the Co-Simulation of Multi-Processor SoC.* In *DATE '04: Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 1530–1591, Washington, DC, USA, 2004. IEEE Computer Society, ISBN 0-7695-2085-5-1.

[25] Führer, T., B. Muller, W. Dieterle, F. Hartwich, R. Hugel, and M. Walther: *Time Triggered Communication on CAN.* Techn. rep., Robert Bosch GmbH, 2000.

[26] Gerstlauer, A., H. Yu, and D.D. Gajski: *RTOS Modeling for System Level Design.* In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, pp. 130–135, Washington, DC, USA, 2003. IEEE Computer Society, ISBN 0-7695-1870-2.

[27] Guermazi, R. and L. George: *Worst case end-to-end response times of periodic tasks with an AUTOSAR/FlexRay infrastructure.* In *RTN'08: Proceedings of the 7th International Workshop on Real-Time Networks*, Prague, Czech Republic, 2008.

[28] Hardung, B., T. Kölzow, and A. Krüger: *Reuse of Software in Distributed Embedded Automotive Systems.* In Navet, N. and F. Simonot-Lion (eds.): *Automotive Embedded Systems Handbook*, ch. 8. CRC Press, Inc., Boca Raton, FL, USA, 2008, ISBN 084938026X, 9780849380266.

[29] Hau, Y. and M. Khalil-Hani: *SystemC-based HW/SW Co-Simulation Platform for System-on-Chip (SoC) Design Space Exploration.* In *ICED '08: Proceedings of the International Conference on Electronic Design*, pp. 1–6, December 2008.

[30] Karner, M., M. Krammer, S. Krug, E. Armengaud, C. Steger, and R. Weiss: *Heterogeneous Co-Simulation Platform for the Efficient Analysis of FlexRay-based Automotive Distributed Embedded Systems (to be published).* In *WFCS '10: Proceedings of the 8th IEEE International Workshop on Factory Communication Systems*, pp. 1–10, Nancy, France, May 2010.

[31] Karner, M., C. Steger, R. Weiss, E. Armengaud, D. Watzenig, and G. Knoll: *Verification and Analysis of Dependable Automotive Communication Systems based on HW/SW Co-Simulation.* In *ETFA*

'08: IEEE International Conference on Emerging Technologies and Factory Automation, pp. 444–447, September 2008.

[32] Krause, M., O. Bringmann, A. Hergenhan, G. Tabanoglu, and W. Rosenstiel: *Timing Simulation of Interconnected AUTOSAR Software-Components.* In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pp. 474–479, San Jose, CA, USA, 2007. EDA Consortium, ISBN 978-3-9810801-2-4.

[33] Krause, M., D. Englert, O. Bringmann, and W. Rosenstiel: *Combination of Instruction Set Simulation and Abstract RTOS Model Execution for Fast and Accurate Target Software Evaluation.* In *CODES/ISSS '08: Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pp. 143–148, New York, NY, USA, 2008. ACM, ISBN 978-1-60558-470-6.

[34] Mills, C., S.C. Ahalt, and J. Fowler: *Compiled Instruction Set Simulation.* Software: Practice and Experience, 21(8):877–889, 1991.

[35] OSEK/VDX Steering Committee: *OSEK/VDX Communication Specification, v3.0.3.* Techn. rep., July 2004.

[36] OSEK/VDX Steering Committee: *OSEK/VDX Operating System Specification, v2.2.3.* Techn. rep., February 2005.

[37] Pop, T., P. Pop, P. Eles, Z. Peng, and A. Andrei: *Timing Analysis of the FlexRay Communication Protocol.* In *ECRTS '06: Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pp. 203–216, Washington, DC, USA, 2006. IEEE Computer Society, ISBN 0-7695-2619-5.

[38] Posadas, H., J.A. Adamez, E. Villar, F. Blasco, and F. Escuder: *RTOS Modeling in SystemC for real-time embedded SW simulation: A POSIX model.* Design Automation for Embedded Systems, 10(4), December 2005.

[39] Pretschner, A., M. Broy, I.H. Kruger, and T. Stauner: *Software Engineering for Automotive Systems: A Roadmap.* In *FOSE '07: 2007 Future of Software Engineering*, pp. 55–71, Washington, DC, USA, 2007. IEEE Computer Society, ISBN 0-7695-2829-5.

[40] Racu, R., R. Ernst, K. Richter, and M. Jersak: *A Virtual Platform for Architecture Integration and Optimization in Automotive Communication Networks.* In *SAE '07: SAE World Congress & Exhibition*, Detroit, MI, USA, April 2007. SAE International.

[41] Reshadi, M., P. Mishra, and N. Dutt: *Instruction set compiled simulation: a technique for fast and flexible instruction set simulation.* In *DAC '03: Proceedings of the 40th Conference on Design Automation*, pp. 758–763, New York, NY, USA, 2003. ACM, ISBN 1-58113-688-9.

[42] Richter, K.: *The AUTOSAR Timing Model - Status and Challenges -.* In *ISOLA '06: Proceedings of the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pp. 9–10, Washington, DC, USA, November 2006. IEEE Computer Society, ISBN 978-0-7695-3071-0.

[43] Schirner, G., A. Gerstlauer, and R. Domer: *Abstract, Multifaceted Modeling of Embedded Processors for System Level Design.* In *ASP-DAC '07: Proceedings of the 2007 conference on Asia South Pacific design automation*, pp. 384–389, Washington, DC, USA, 2007. IEEE Computer Society, ISBN 1-4244-0629-3.

[44] Schröder-Preikschat, W.: *Automotive Betriebssysteme.* In *PEARL '04: Proceedings of the workshop on Eingebettete Systeme*, November 2004.

[45] Tsikhanovich, A., E. Aboulhamid, and G. Bois: *Timing Specification in Transaction Level Modeling of Hardware/Software Systems*. In *MWSCAS '07: Proceedings of the 50th Midwest Symposium on Circuits and Systems*, pp. 249–252, August 2007.

[46] Wietzke, J.: *Embedded Systeme, embedded Probleme - Zunehmender Qualitätsverlust bei der Entwicklung eingebetteter Systeme*. Elektronik Automotive, 1:63–67, 2007.

[47] Yu, H. and D.D. Gajski: *RTOS Modeling in System Level Synthesis*. Techn. rep., University of California, Irvine, August 2002.

[48] Zabel, H., W. Müller, and A. Gerstlauer: *Accurate RTOS Modeling and Analysis with SystemC*. In Müller, W., W. Ecker, and R. Dömer (eds.): *Hardware-dependent Software*, ch. 9. Springer Netherlands, 2009.

[49] Zimmermann, W. and R. Schmiedgall: *Bussysteme in der Fahrzeugtechnik*. Vieweg+Teubner, September 2008.