UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

A framework for Assertion-Based Timing Verification and PC-Based Restbus Simulation of Automotive Systems

Dissertation

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

by

Gilles Bertrand Gnokam Defo

Paderborn, 2015

Supervisors:
Prof. Dr. Franz-Josef Rammig, University of Paderborn
Prof. Dr. Marco Platzner, University of Paderborn

Date of public examination: 11.09.2015

Abstract

Automotive system innovation is mainly driven by software which can be distributed over a large number of functions typically deployed over several Electronic Control Units (ECUs). This growing design complexity makes the verification and validation process challenging and difficult. Therefore, the development of efficient and effective design methodologies is of great interest for automotive engineers.

A central concept in the development of automotive software is the component-based approach. Currently, the most prominent approach that supports this design paradigm is the AUTomotive Open System ARchitecture (AUTOSAR).

The System-Level Design Language (SLDL) SystemC provides means to simulate the behavior of AUTOSAR software components by means of a discrete-event simulation kernel. Additionally, SystemC comes with a set of libraries such as the SystemC Verification Library (SCV). Meanwhile, the interest of using SystemC has grown in the automotive software development community.

In this thesis we present a SystemC-based design methodology for early validation of time-critical automotive systems. The methodology spans from pure SystemC simulation to PC-based Restbus simulation. To deal with synchronization issues (oversampling and undersampling) that arise during Restbus simulation between the SystemC simulation model and the remaining bus network, we also present a new synchronization approach.

Finally, we make use IP-XACT for SystemC component integration. To capture timing constraints on the simulation model, we propose timing extensions for the IP-XACT standard. These timing constraints can then be used to verify the SystemC simulation model.

Kurzfassung

Innovation in der Automobilindustrie wird durch Elektronik und vor allem durch Software ermöglicht. In der Regel wird eine Vielzahl von verteilten Funktionen realisiert. Typischerweise, wird diese Software über mehrere Steuergeräte verteilt.

Durch die Verteilung und die Vielzahl an Funktionen, ensteht eine immer wachsende Komplexität, die den Verifikations- und Validierungsprozess anspruchsvoller und schwieriger gestaltet. Daher ist für Ingenieure in der Automobilindustrie die Entwicklung von effizienten und effektiven Design-Methoden von großem Interesse.

Ein zentrales Element in der Entwicklung automobiler Software ist der komponentenbasierte Ansatz. Derzeit ist AUTOSAR der wichtigste Standard, der dieses Paradigma unterstützt. Die Systembeschreibungssprache SystemC ist ebenfalls ein Mittel, um AUTOSAR-Komponenten simulieren zu können. Desweiteren stellt SystemC einen Satz von Bibliotheken zur Verfügung wie zum Beispiel die “SystemC Verification Library“ (SCV), und einen diskreten Event-Simulationskern. Inzwischen ist das Interesse an der Verwendung von SystemC in der automobile Softwareentwicklung stark gestiegen.

In dieser Arbeit stellen wir eine SystemC-basierte Entwurfsmethodik für eine frühe Validierung zeitkritischer automobile Systeme vor. Die Methodik reicht von einer reinen SystemC-Simulation bis zu einer PC-basierten Restbussimulation. Um die Synchronisation bezüglich Überabtastung und Unterabtastung zwischen dem SystemC-Simulationsmodell und dem Restbus während der Restbussimulation zu gewährleisten, präsentieren wir ein Synchronisationsverfahren.

Im Rahmen dieser Arbeit, wurde für die Integration von SystemC-Komponenten IP-XACT als Modelierungsstandard verwendet. Um eine Zeitanalyse ermöglichen zu können, stellen wir Erweiterungen für den IP-XACT Standard vor, mit deren Hilfe Zeitanforderungen an das Simulationsmodell erfasst werden können.

Acknowledgements

First and foremost, I would like to thank Prof. Dr. Franz-Josef Rammig for his guidance, his continuous support and valuable suggestions and ideas throughout the development of the concepts of this thesis. I also want to thank Prof. Dr. Marco Platzner for co-supervising this thesis.

Moreover, I would like to thank Jun. Prof. Christian Plessl, Dr. Wolfgang Müller, Dr. Stefan Sauer for being member of the examination board.

In C-LAB, I had the opportunity to contribute to several industrial research projects under the auspices of Dr. Wolfgang Müller in a great research environment. I thank him for his support and our numerous fruitful discussions. I would like to thank all my former colleagues at C-LAB and the University of Paderborn for the friendly and inspiring environment.

Furthermore, I would also like to thank Mr. Stefan Kuntz for the fruitful discussions and comments during the joint research projects, and for his support in the final phase of this thesis.

I am most indebted to my parents Prof. Dr. Edmond Gnokam and Elise Gnokam who have supported me not only during my studies but throughout my whole life in every imaginable way.

Last but not least, I would like to thank Dr. Anna Barát for being an on-going source of support and motivation during the time of this research and dedicate this thesis to my children.

Gilles Bertrand Gnokam Defo

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.1.1 Automotive system innovation . . . 1
1.1.2 Simulation . . . 2
1.1.3 Modeling . . . 3
1.2 Problem statement . . . 3
1.2.1 Restbus simulation for early functional validation . . . 3
1.2.2 Data synchronization between simulator and hardware . . . 4
1.2.3 Specification of timing constraints . . . 5
1.3 Research contribution . . . 5
1.4 Structure of this thesis . . . 6

2 Foundation **9**
2.1 Automotive Control Systems . . . 9
2.1.1 Open-Loop Control . . . 10
2.1.2 Closed-Loop Control . . . 11
2.2 Design of Automotive Control Systems . . . 12
2.2.1 Classification of Real-Time systems . . . 12
2.2.2 Design Methodology (AUTOSAR) . . . 14
2.2.3 Timing Modeling with TADL2 . . . 20
2.2.4 Testing and Verification . . . 26
2.3 Automotive Vehicle Netwoks . . . 28
2.3.1 Controller Area Network (CAN) . . . 28
2.3.2 FlexRay . . . 29
2.4 Design of Electronic Systems . . . 33
2.4.1 Design Modeling with IP-XACT . . . 33
2.4.2 Design Modeling Language with SystemC . . . 37
2.4.3 Formal Property Specification Language with PSL . . . 42

3 Related Work **49**
3.1 IP-XACT . . . 49
3.1.1 Extensions of the IP-XACT Schema . . . 49
3.2 Modeling and simulation of embedded automotive software . . . 50
3.2.1 Restbus Simulation . . . 50
3.2.2 Modeling and simulation with SystemC . . . 51

3.2.3 Design Framework for IP Reuse and Integration 52
3.2.4 AUTOSAR Vs. SystemC 52
3.3 Verification of temporal properties 53
3.3.1 Verifying SystemC using an Intermediate Verification Language and Symbolic Simulation 53
3.3.2 Verifying SystemC using a software model checking approach 53
3.3.3 Monitoring Temporal SystemC Properties 54
3.3.4 Dynamic Assertion-Based Verification 54
3.3.5 Assertion-based Verification of temporal properties 55

4 Methodology 57
4.1 Overall design flow 58
4.1.1 Phase 1: Component assembly 59
4.1.2 Phase 2: Timing requirements formalization and Code generation . . . 59
4.1.3 Phase 3: Timing verification 63
4.1.4 Phase 4: Model equivalence check 64
4.1.5 Phase 5: Restbus simulation 64
4.2 Restbus simulator 65
4.2.1 Architecture of the Restbus Simulator 65
4.2.2 The SystemC simulator 65
4.2.3 Adapter 67

5 Assertion-Based Timing Verification 71
5.1 Background 72
5.1.1 Motivation 72
5.1.2 PSL, Sequential Extended Regular Expression (SERE) 74
5.1.3 IP-XACT 75
5.1.4 DataEvents and Event chains 77
5.1.5 Timing Augmented Description Language 2 (TADL2): Notation 78
5.2 Formalizing Timing Requirements 79
5.2.1 Reason for using both TADL2 and Property Specification Language (PSL) 80
5.2.2 RepeatConstraint 82
5.2.3 StrongDelayConstraint 83
5.2.4 RepetitionConstraint 85
5.2.5 DelayConstraint 87
5.2.6 SporadicConstraints 89
5.2.7 Periodic constraints 90
5.2.8 Synchronization Constraint 92
5.2.9 Order Constraint 95
5.3 Verification of the timing properties 96
5.4 Summary 97

6 Verification of timing properties: case study Brake-By-Wire 99
6.1 Functional decomposition of the BBW model 100
6.2 Instrumenting of the simulation model 103

6.3 Reference model . . . 103
6.4 Specifying the timing requirements . . . 104
6.5 Evaluation results . . . 105
6.5.1 Repeat, StrongDelay and Repetition timing constraints . . . 105
6.5.2 Evaluation of the AgeConstraint and ReactionConstraint . . . 107
6.5.3 Evaluation of synchronization related timing Constraints . . . 108
6.6 Summary and discussion . . . 109

7 Synchronization **111**
7.1 Background . . . 111
7.1.1 Data smoothing: Robust LOWESS/LOESS . . . 111
7.1.2 Multirate Systems . . . 115
7.1.3 Downsampling . . . 117
7.2 Our synchronization approach . . . 121
7.3 Upsampling . . . 123
7.4 Downsampling . . . 124
7.4.1 Main phase . . . 125
7.4.2 Initialization phase . . . 126
7.4.3 Downsampling with peak detection . . . 127
7.5 Summary . . . 130

8 Evaluation of Synchronization approach **133**
8.1 Evaluation platform . . . 133
8.1.1 System Overview . . . 133
8.1.2 Hardware architecture . . . 134
8.1.3 Software architecture . . . 136
8.1.4 Applied tools . . . 137
8.2 Evaluation results . . . 137
8.2.1 Impact of the SendQueue-size on transmission delay . . . 140
8.2.2 Variation of the smoothing parameter of Robust LOWESS . . . 140
8.2.3 Impact of the SendQueue size on peak sequence detection . . . 141
8.2.4 Impact of the data processing rate ratio on data synchronization . . . 144
8.2.5 Summary . . . 146

9 Conclusion **147**
9.1 Summary . . . 147
9.2 Outlook . . . 149
9.2.1 Synchronization . . . 149
9.2.2 Timing verification . . . 149

A Verification unit **151**

B Pictorial representation of the IP-XACT Schema Extensions **155**
B.1 Diagrams . . . 155
B.1.1 Elements and sequences . . . 155
B.1.2 Elements and choices . . . 156

B.1.3 Elements, attributes, groups, and attributeGroups 157
B.1.4 Wildcards 158

List of Acronyms **159**

List of Figures **165**

List of Tables **167**

List of Own Publications and Bibliography **169**

Chapter 1

Introduction

1.1 Motivation

1.1.1 Automotive system innovation

Meanwhile safety, security, driving dynamics, and comfort features have significantly been improved in today's automobile. This was made possible by the availability of intelligent sensor and actuator technologies and powerful ECUs, whereby some of the ECUs are nowadays equipped with multi-core processors.

To keep up with the competition on the market, automotive Original Equipment Manufacturers (OEMs) try to shorten their development cycle, whilst keeping or enhancing their products quality.

Typically, these systems are complex real-time embedded systems composed of a large number of functions deployed over several ECUs, whereby the ECUs communicate among each other via network buses like FlexRay or CAN. This leads to an ever growing design complexity, which makes the test and validation process challenging.

Therefore, it is necessary to start the test and validation process as early as possible in the design process.

1.1.2 Simulation

System level simulation

Usually, a significant amount of time is spent during the system architecture design phase of the automotive development process. In this phase hardware- and software partitioning is done. Hereby, the hardware architecture comprises elements such as ECUs, bus systems, sensors and actuators and energy supply. The software infrastructure includes the operating system, bus drivers and additional services.

Simulation can be used to speed up the system architecture design exploration phase. It gives system designers the possibility to test their designs against predefined functional requirements and therewith analyze communication interactions and the overall system design in an early phase of the development process.

SystemC [53] is a well deployed System Level Description Language (SLDL) in the field of Embedded System level design. Strictly speaking, SystemC is not a language but rather a C++ based class library, which is coupled with an integrated verification library: the SystemC Verification Library (SCV) and a discrete-event simulation kernel. Furthermore, SystemC provides means to describe, analyze and verify both hardware and software models at various levels of abstraction (see Section 2.4.2).

Hardware-in-the-Loop simulation

Since in-vehicle driving tests are often time-consuming, expensive and not reproducible, Restbus simulation (RBS) and Hardware-In-the-Loop (HIL) are applied in later phases of the development cycle. RBS and HIL are widespread techniques. They allow developers to validate new hardware and software solutions within their target environment.

HIL simulation provides an effective platform by adding the complexity of the plant under control to the test platform. The complexity of the plant under control is included in the test and development by adding a mathematical representation of all related dynamics of the system. These mathematical representations are referred to as the plant simulation. The embedded system to be tested interacts with the plant during simulation.

Restbus simulation on the other hand is a special HIL variant, whereby this method takes the bus network into account. A typical application for Restbus simulation is the integration of any developed functionality (features) into an existing bus network. To achieve this, the Restbus simulator has to pro-

vide messages from non-existing nodes to the rest of the system during the simulation. It also has to consume message from existing nodes and react accordingly.

1.1.3 Modeling

To benefit from the components-oriented approach when designing complex embedded systems, several modeling languages have been proposed in the literature. In the context of automotive system development there exists a variety of modeling standards, the most prominent ones are the AUTOSAR and the Systems Modeling Language (SysML). Another far but related modeling standard from the field of Electronic Design Automation (EDA) is IP-XACT.

SysML [43] is a general-purpose modeling language for system engineering applications. It supports the specification, analysis, design, verification and validation of a broad range of systems and systems-of-systems. SysML was originally developed by an open source specification project, and includes an open source license for distribution and use.

The AUTOSAR [27] initiative was founded in 2003 and is a union of well-known vehicle manufacturer and supplier of automotive systems with the goal of having a common framework for the development of software components. The AUTOSAR standard defines not only a comprehensive technical infrastructure for these systems, but also one that builds upon the methodology and description formats for the development of AUTOSAR-compliant systems. A further primary goal of AUTOSAR is the integration aspect.

IP-XACT [48] is an XML standard format for IP packaging, reuse and integration. This standard was originally created by Mentor Graphics [45]. It defines a standard way for describing and handling multi-sourced electronic IP components, enabling an automated design integration and configuration within multi-vendor design flows.

1.2 Problem statement

1.2.1 Restbus simulation for early functional validation

SystemC has turned out to be suitable for the modeling and simulation of automotive networks systems [63]. As aforementioned, it provides means to describe both hardware and software and a simulation environment. Furthermore, several verification methodologies and libraries support the test and verification of SystemC models. The most popular ones are: UVM [54] (Uni-

versal Verification Methodology), OVM [20] (Open Verification Methodology) and PSL [52] (Properties Specification Language).

Several companies provide tools and hardware equipment for Hardware-in-the-Loop and Restbus simulation. They all require the use of dedicated hardware to run the design under test. In those simulation environments, the standard PC is only used to host the test automation software. Up to now there is no approach describing how to actually use the standard host PC as execution platform for the design under test.

The main contributions of this thesis consists of providing a framework and design methodology to support the use of a standard PC for the simulation of the design under test during Restbus simulation. Moreover, the Restbus simulator is implemented in SystemC.

1.2.2 Data synchronization between simulator and hardware

The notion of time is one of the key concepts of SystemC, which enables the simulation of concurrent processes. However, the standard SystemC simulation kernel does not provide real-time simulation support. Mechanisms such as preemption and priority based scheduling are not available. Therefore, it is not possible to model the complete software behavior of a real-time critical system.

There exist a wide range of SystemC-based Real-Time Operating System (RTOS) modeling concepts. These concepts consist of implementing the operating system features at a high abstraction level, and later during the refinement process, the RTOS model is typically replaced by a custom RTOS. The simulation overhead introduced by RTOS models has been proven to be negligible [77].

Nevertheless, none of those RTOS modeling concepts can be applied to Restbus simulation or Hardware-in-the-Loop, since they only target the simulation of a real-time environment and not simulation itself in real-time. Further, SystemC simulation time differs from the real network time.

As a consequence, during the simulation of complex models, not all the data generated by the real network nodes might be received and processed on time by the corresponding components in the SystemC simulation. Oversampling and Undersampling might be observed.

1.2.3 Specification of timing constraints

IP-XACT has been proven to be a well-defined standard for hardware design description. It provides a common language and vendor-neutral way to describe IPs. Thus, we opted for its use as an exchange format for the description of the individual Restbus and HIL components.

In this thesis we present a SystemC-based simulation approach, where the SystemC simulation model interacts with the real hardware infrastructure. In the context of the simulation of real-time critical systems, timing can be considered as a functional property. Therefore, the model should be analyzed in order to verify if it conforms to the specified timing requirements. This analysis has to be done before exercising the actual test process.

For component abstraction representation, the IP-XACT standard does not provide support for the entire range of abstraction levels usually considered in the literature. However, Transaction Level Modeling (TLM) and Register Transfer Level (RTL) are supported, which are the most important ones.

Further, the notion of timing constraints already exists in IP-XACT. Delay timing constraints can be specified for RTL models. But unfortunately the current standard does not provide similar elements for transactional models and the standard needs to be extended by additional timing constraints to enable a more detailed timing analysis of complex designs.

1.3 Research contribution

The aim of this thesis is to provide a framework for SystemC-based Restbus simulation of automotive systems for early validation. For this purpose we define a methodology for timing analysis of SystemC models in a pure SystemC simulation environment, before starting the actual Restbus simulation process. The defined methodology is depicted in Figure1.1.

The research contributions in this dissertation can be defined as follows:

1. Proposal of additional timing constraints for the IP-XACT standard version IEEE 1685-2009 [48] to capture timing constraints.
2. Development of a property specification-based verification flow to verify and enhance the simulation models constructed. IEEE 1850 version of PSL [52] (Property Specification Language) was used.
3. Derivation of PSL formulas from specified IP-XACT-timing-extensions. This contribution consists of defining mapping rules from IP-XACT timing constraint specifications into executable PSL formulas.

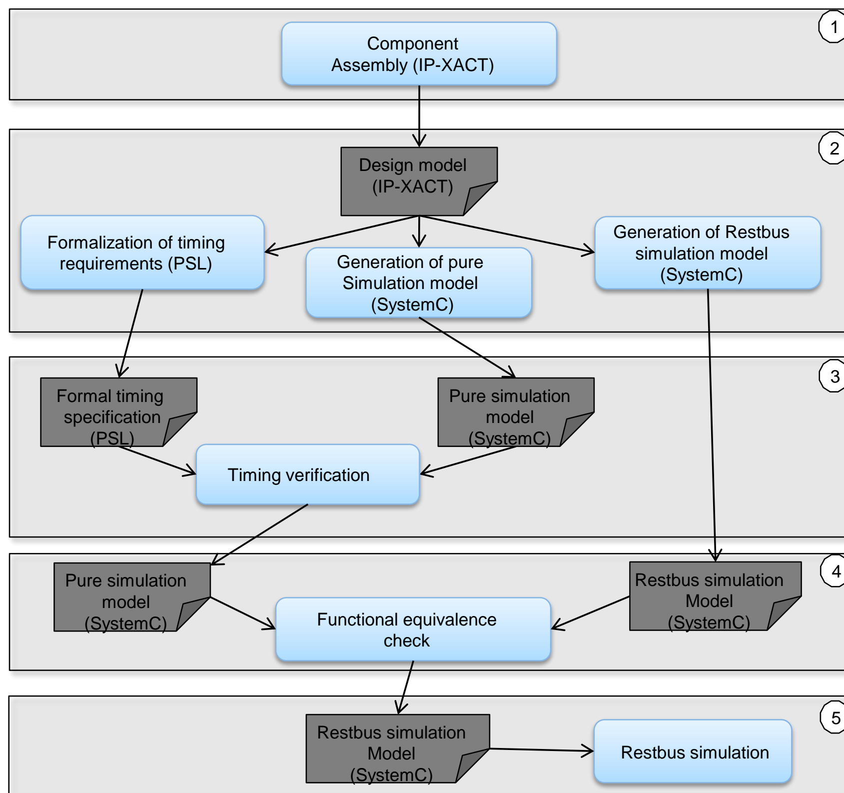


Figure 1.1: Proposed design flow

4. Development of a synchronization algorithm to handle possible sampling rate issues between simulator running on the host PC and the real hardware. The synchronization approach depends on the possible kind of application that can be simulated on the host PC. We classified the applications into the following three categories: Open-loop control, Monitoring, and Closed-loop control.

1.4 Structure of this thesis

Chapter 2 introduces the theoretical background of the fields involved in this thesis. This includes the field of automotive control systems and the design of such systems. Additionally, the automotive vehicle network communication protocols FlexRay and CAN are discussed. At last but not least a brief introduction into the field of Electronic System Level design is given. Chapter 3 briefly discusses research results related to the simulation of automotive network and the design of electronic systems.

The chapters 4, 5 and 7 discuss the contributions of this thesis. Chapter 4 presents our methodology and simulation framework for functional verification of real-time critical automotive systems developed in this thesis. The

methodology includes a timing analysis of the model under investigation, that helps to verify that the model conforms to its timing requirements before starting the actual Restbus simulation process. Furthermore, the timing requirements are captured using IP-XACT.

In Chapter 5, timing extensions for IP-XACT needed to capture timing constraints on the design will be introduced. Chapter 7 is explicitly dedicated to one of the most important contributions of this thesis, which is the development of an approach needed for data synchronization between our Restbus simulator and the network bus.

The chapters 6 and 8 are dedicated to the evaluation of both the timing verification and the synchronization approach. Chapter 6 evaluates the timing verification methodology. The system model used in this case study is a SystemC model of an automotive Brake-By-Wire (BBW) application. The application is distributed over a set of ECUs and includes Anti-lock Braking System (ABS) functionality. Chapter 8 evaluates the synchronization approach. The approach was validated on a HIL test environment consisting of a Steer-By-Wire system. This test environment has been used in several projects [90, 49].

Chapter 2

Foundation

This chapter introduces the theoretical background of the domains involved in this thesis. We will start by giving a brief introduction into the field of automotive control systems and the design of such systems. Following, the automotive vehicle network communication protocols FlexRay and CAN will be discussed, and finally a brief introduction into the field of Electronic System Level design will be given.

2.1 Automotive Control Systems

In today's automotive systems, control systems running on electronic control units (ECU) are used to regulate the operation of other systems. From the control application's perspective, the system being controlled is referred to as the *system plant*. In this sections we will describe the major types of automotive control systems. A control system is described by its fundamental elements, which are:

- Objectives of control
- System components
- Results/Outputs.

The objectives of a control system are the quantitative measures of the tasks to be performed by the system. These describe the desired values of one or more variables and are normally specified by the user [83].

The results are called outputs or controlled variables. Typically, the objective of a control system is to regulate the values of the outputs in a prescribed

manner based on the inputs of the control system. A control system should perform accurately, respond quickly, be stable and immune against noise.

The control system's accuracy specifies the deviation between the system's output and the desired system's output, with a constant-value input command. A quick response describes how fast the control system will track or follow changing input commands. A system's stability specifies how a system behaves, when a sudden change is made by the input signal.

The output of an unstable control system will diverge from its intended value based on its input. For any automotive application, a control system must be stable and controllable over the entire desired operating range.

A good controller design will minimize the chance of unstable operation even under extreme operating conditions. A system should maintain its accuracy by responding only to valid inputs. When noise or other disturbances threaten to change the system plant's output, good design will eliminate response to such inputs from system performance as much as possible [83].

A control system having small (ideally zero) response to noise inputs is said to have good noise immunity. Accuracy, quick response, stability and noise immunity are all determined by the control system configuration and parameters chosen for a particular plant. The purpose of a control system is to determine the output of the system (plant) being controlled in relation to the input and in accordance with the operating characteristics of the controller.

The relationship between the controller input and the desired plant output is called the control law for the system. The desired value for the plant output is often called the set point. The output of an electronic control system is an electrical signal that must be converted to some physical (or other) action in order to regulate the plant. A device that converts the electrical signal to the desired mechanical action is called an actuator.

Although electronic controllers can, in principle, be implemented with either analog or digital electronics, the trend in automotive control is digital. There are two major categories of control systems: open-loop (or feedforward) and closed-loop (or feedback) systems [83].

2.1.1 Open-Loop Control

Figure 2.1 depicts the generic structure of an open-loop control system. As shown in the figure, the components of an open-loop control system include an electronic controller, which has an output to an actuator. The actuator in turn, regulates the plant in accordance with the desired relationship between

the reference input (input command) and the value of the controlled variable (denoted by u) in the plant.

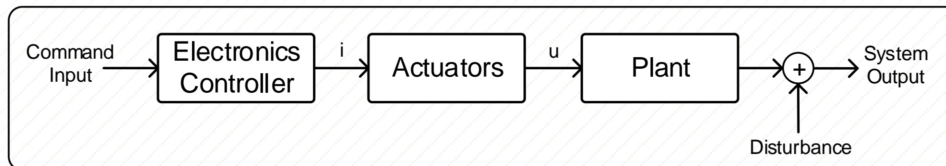


Figure 2.1: Block diagram of an open-loop control system

Many examples of open-loop control systems are encountered in automotive electronic systems, such as fuel control in certain operating modes [83]. As illustrated in Figure 2.1, the command input is sent to the electronic controller, which performs a control operation on the input to generate an intermediate electrical signal (denoted by i).

An open-loop control system is a system in which the computation of the controlled variables only depends on the current state and its system model. It does not use feedback to determine whether its output has achieved the expected goal or not. Thus, the operation of the plant is directly regulated by the actuator. The system's output may also be affected by external disturbances that are not an inherent part of the plant, but are the result of the operating environment.

One of the main drawbacks of the open-loop controller is its inability to compensate for external changes that might occur in the plant or for any disturbances.

2.1.2 Closed-Loop Control

Closed-loop control systems are usually more robust than open-loop control systems. In a closed-loop control system, the actual system output is compared to the desired output value in accordance with the input. As shown in Figure 2.2, the measurements of the output variable being controlled is obtained via a sensor and feedback to the controller.

Each measured value of the controlled variable is compared with the desired value based on a reference input. A deviation signal based on the difference between desired and actual values of the output signal is created, and the controller generates an actuator signal (u) that tends to reduce the error to zero.

In addition to reducing this error to zero, feedback has other potential benefits in a control system. It can affect the control system performance by

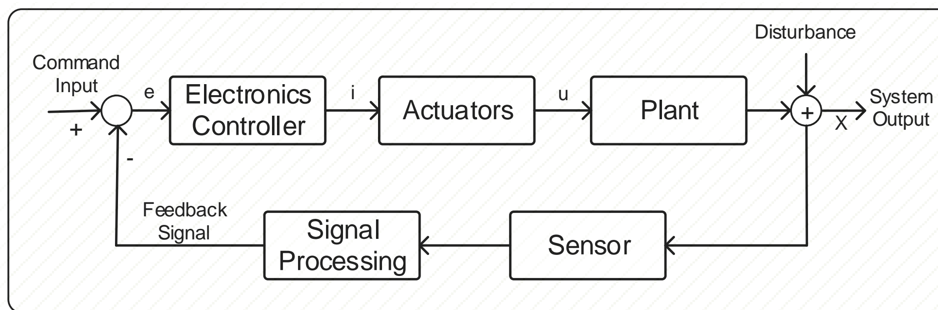


Figure 2.2: Block diagram of a closed-loop control system

improving system stability and suppressing the effects of disturbances in the system.

The generic closed-loop control system illustrated in Figure 2.2 has some of the components shown in an open-loop control system, including the plant to be controlled, actuator(s), and control electronics. In addition, however, this system includes one or more sensors and some signal-conditioning electronics. The signal conditioning used in a closed-loop control system plays a role similar to that played by signal processing in measurement instrumentation.

That is, it transforms the sensor output as required to achieve the desired measurement of the plant output. Compensation for certain sensor defects (e.g. limited bandwidth) is possible, and in some cases necessary, to enable the comparison between the plant output and the desired value. Electronic control systems are classified by the way in which the error signal is processed to generate the control signal. The major control systems include proportional (P), proportional-integral (PI), and proportional-integral-differential (PID) controllers [83].

2.2 Design of Automotive Control Systems

2.2.1 Classification of Real-Time systems

A real-time computer system is a system, where the correctness of the system behavior depends not only on the logical results of the computations, but also on the physical time, when these results are produced [61]. By system behavior we mean the sequence of system's outputs over time.

The classification of real-time systems depends on the design perspectives. In [62] for instance, five different perspectives are defined and classified as follows:

1. Hard Real-Time Vs Soft Real-Time
2. Fail-Safe Vs Fail-Operational
3. Guaranteed-Response Vs Best effort
4. Resource-adequate Vs Resource-inadequate
5. Event-triggered Vs Time-triggered.

The first two classifications are influenced by the characteristics of the environment, that is, on the factors outside the computer system. The last three classifications are based on the tightness and strictness of the deadlines within the computer system, this means, on factors inside the computer system, such as peak load and the fault scenario [62].

Hard real-time systems describe the category of systems, where it is absolutely imperative for a system to provide its response within its required deadline. Missing the deadline would lead to useless results or even dangerous system state. In Soft real-time systems on the other hand, missing a deadline will only lead to a deterioration of the system's quality [62].

The classification Fail-safe Vs Fail-operational is related to the characteristics of the controlled object and not the computer system itself. Fail-safe systems are real-time systems, where one or more safe states can be reached in case of system failure. Therefore, fail-safe systems are required to have a high error-detection coverage. This high error-detection coverage is typically reached by means of a special external device called watchdog. In case of failure, the watchdog forces the controlled object into a safe state. Fail-operational systems however are systems where a safe state cannot be identified, e.g.: a flight control system on an airplane. In such systems, the computer must remain operational and provide a minimal level of service in case of failure to avoid a catastrophe [62].

Guaranteed-response systems are systems that make it possible to reason about the adequacy of the design independently of probabilistic arguments like peak loads and fault scenario. The probability of failure of a perfect system with guaranteed response is reduced to the probability that the assumptions about the peak load and the number and types of faults do not hold in reality. This probability is called assumption coverage [78]. Guaranteed response systems require careful planning and extensive analysis during the design phase. If such an analytic response guarantee cannot be given, we speak of a best-effort design. Moreover best-effort systems are typically only used for non safety-critical real-time systems. [62].

Guaranteed-response systems are based on the principle of resource adequacy, i.e., there are enough computing resources available to handle the

specified peak load and the fault scenario. Many non safety-critical real-time system designs are based on the principle of resource inadequacy. It is assumed that the provision of sufficient resources to handle every possible situation is not economically viable, and that a dynamic resource allocation strategy based on resource sharing and probabilistic arguments about the expected load and fault scenarios are acceptable. Hard real-time systems must be designed according to the guaranteed response paradigm that requires the availability of adequate resources [62].

Event-triggered (ET) and Time-triggered (TT) real-time systems differentiate themselves in the way they are internally triggered. They are not characterized by the behavior of their environment. A trigger hereby is a an event that causes the start of some activity in the computer, such as, the execution of a task (processing activity) or the transmission of a message (communication). In an ET control system, all communication and processing activities are initiated whenever a significant event other than the regular event of a clock tick occurs. In a TT system, all activities are initiated by the progression of time. Further, every observation of the controlled object is time-stamped with this global time [62].

2.2.2 Design Methodology (AUTOSAR)

The driving factors for the development of automotive systems are mainly related to customer requests and related to the regulation entities. On the one hand, customers express the need for more and more features whilst increasing safety and security aspects. On the other hand, environmental constraints have to be fulfilled from the regulation entity's perspective. This leads to an increasing design complexity.

Software is a driving factor for automotive innovation. A key concept in the development of automotive software is the component-based approach and the most prominent approach is AUTOSAR [27]. AUTOSAR is an initiative of a union of well-known vehicle manufacturers and suppliers of the automobile industry and was founded in 2003. AUTOSAR provides a common standard for the development of automobile software. Furthermore, the standard defines not only a comprehensive technical infrastructure for automotive systems, but it also builds upon a methodology and description formats for the development of AUTOSAR-compliant systems [73].

The development methodology is model-driven. The software architecture, as well as the ECU hardware and the network topology, are modeled in a formal way, which is defined by a metamodel that supports the software development process. All available modeling elements are specified by the AUTOSAR metamodel [27].

Figure 2.3 depicts the architecture of the AUTOSAR software layers. AUTOSAR distinguishes between three software layers, running on top of the ECU hardware. These layers are: the Application layer, the Runtime Environment (RTE) and the Basic Software (BSW).

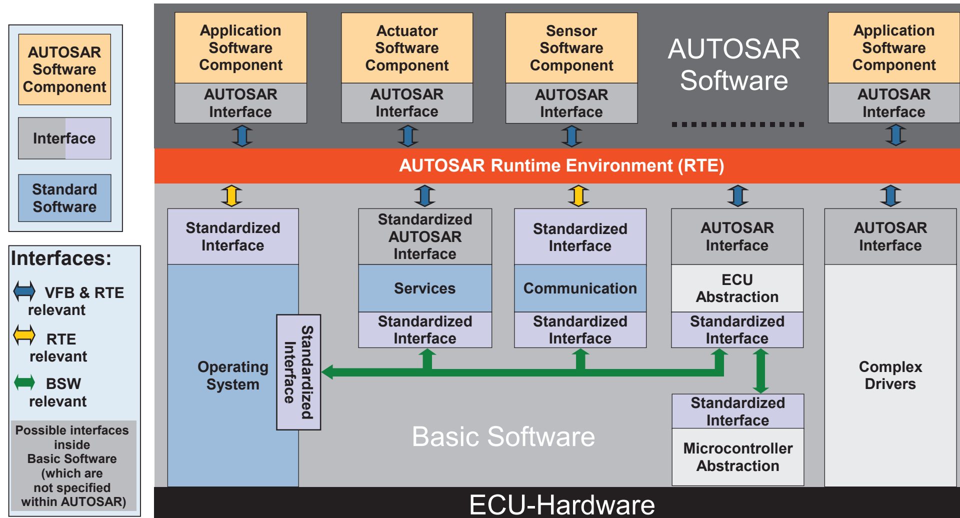


Figure 2.3: AUTOSAR layered software architecture

Application Layer

The AUTOSAR Software layer consists of Software Components (SW-Cs) that are mapped on to the ECU. A SW-C is the fundamental element in AUTOSAR. A SW-C can be either a *Composition* or *Atomic*.

While Atomic SW-C encapsulate the implementation of their functionality and behavior and expose well-defined connection points, the purpose of Composition SW-C is to allow the encapsulation of specific functionality by aggregating existing SW-C. Software components can be refined into Application-, Sensor/Actuator- and Service software component type [29].

Runtime Environment

From a system design level perspective, the AUTOSAR RTE acts as a middleware for inter- and intra-ECU communication.

The RTE provides a communication abstraction to the AUTOSAR Software layer by providing standardized interfaces and services for both inter-ECU and intra-ECU communication. The software components running on top of the RTE are application dependent, therefore, the RTE layer is partly gener-

ated and partly configured. As a result, the RTE will differ from one ECU to another [30].

Basic Software

The AUTOSAR BSW is a standardized software layer, which provides services to the AUTOSAR SW-C and is required to run the functional part of the software. It does not fulfill any functional task itself and is situated below the AUTOSAR RTE (see Figure 2.3). The Basic Software contains standardized and ECU specific components. Furthermore, this layer provides System, Memory and Communication related services to the application software. Moreover, the BSW layer incorporates a Micro-controller- and ECU abstraction layer [28].

AUTOSAR Timing Extensions

The AUTOSAR timing extension specification [31] provides consolidated and consistent representation of relevant timing dependencies and corresponding timing constraints. For the specification of the timing requirements, the timing extension specification defines a timing model that can be used as specification basis for a contract based development process, in which the development can be carried out by different organizations at possibly distributed locations.

Basically, there are two different interpretation when dealing with timing information. It can either be a restriction for the timing behavior of the system (*TimingConstraint*) or a *TimingGuarantee* for the timing behavior of the system (*TimingDescription*). Minimum or maximum latency bound for a certain sequence of timing events can be taken as constraint, for instance during component integration. A component related timing event can be guaranteed to occur periodically within a certain bound.

In the timing extension specification two basic elements play a key role, namely the timing description event (*TimingDescriptionEvent*) and the timing description event chains (*TimingDescriptionEventChain*). In essence, a *TimingDescriptionEvent* is an abstract representation of a specific observable system behavior, which can be observed during the system's operation, while a *TimingDescriptionEventChain* describes a causal relationship between two timing events. Each event chain has a well-defined stimulus and response, where the stimulus and response elements describe the start and end point of the event chain. These elements can be hierarchically decomposed into an arbitrary number of chain segments.

Furthermore, by means of timing description event chains, the specification of the interrelation between the stimulus of a system and its corresponding response can be formalized and used to constrain the given system dynamic. The AUTOSAR timing extension model [31] distinguishes at the highest level between the following types of timing constraints:

EventTriggeringConstraints characterize a type of timing constraints that can be used to describe the occurrence of the referenced timing event. This constraint can be further refined into:

- PeriodicEventTriggering: specifies the characteristics of a timing description event which occurs periodically.
- SporadicEventTriggering: specifies the characteristics of a timing description event which occurs sporadically.
- BurstPatternEventTriggering: describes a burst of occurrences of a single event and its repetition.
- ConcretePatternEventTriggering: specifies the characteristics of a timing description event which occurs as a concrete pattern and its repetition.

LatencyTimingConstraints are used to specify the amount of time allowed to elapses between the occurrences of any two timing description events. It is always associated with a TimingDescriptionEventChain.

AgeConstraints are used to specify a minimum and maximum age that is tolerated when data is received.

SynchronizationTimingConstraints are used to specify a synchronicity constraint among the occurrences of two or more timing description events.

OffsetTimingConstraints are used to specify an offset between the occurrences of two timing description events.

ExecutionOrderConstraints are used to specify the order of execution of executable entities and ExecutionTimeConstraints are used to specify minimum and maximum execution time constraints of executable entities.

AUTOSAR Timing Views

The AUTOSAR methodology is subdivided into well-defined process steps. Furthermore, the methodology defines all artifacts needed by and provided for each process step. The AUTOSAR timing extension specification groups

the timing related methodology steps by boundary in five views called *VfbTiming*, *SwcTiming*, *SystemTiming*, *BswTiming* and *EcuTiming*.

VfbTiming: A key concept of AUTOSAR is the Virtual Functional Bus (VFB). The VFB abstracts all communication layers encapsulating the underlying architecture of the ECUs and network topology. The concrete implementation of the VFB on an ECU is the RTE. At the modeling level, connections between the ports of the SW-Cs are modeled by means of the so-called connectors.

This view deals with timing information at a logical level, and is related to the interaction of the software components. End-to-end timing constraints can be captured in this view, allowing an early formalization of timing constraints.

The physical distribution of the software components is not considered in this view. A further restriction of the *VfbTiming* view is the fact that each component is treated as black-box, which means the internal behaviour is not considered. Thus, VFB timing description only refers to ports and connections of software components as depicted in Figure 2.4.

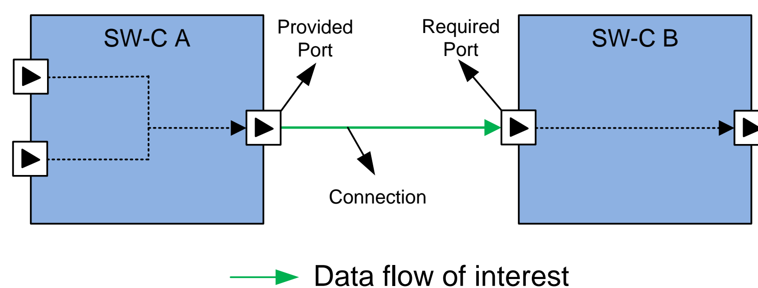


Figure 2.4: Virtual Functional Bus Timing

SwcTiming: The SwcTiming view is used to capture timing information related to the internal behavior (SwcInternalBehavior) of atomic software components. The internal behavior of an atomic software component is modeled

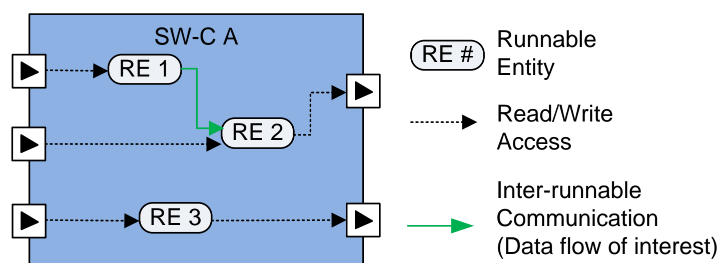


Figure 2.5: Software Component Timing

using the so-called Runnable Entities (RE) element as depicted in Figure 2.5.

Thus, this view is useful for specification engineers that are interested in the internal behavior of the atomic SW-C represented as black boxes in the VfbTiming view. This can be achieved by referring to the activation, start, and termination of the execution of runnable entities.

SystemTiming: In contrast to the VfbTiming and SwcTiming views, the SystemTiming view considers additional information, such as the system topology, software deployment and signal mapping. This additional information is used as system configuration input. Based on that configuration, software components can be mapped on to ECUs with corresponding communication matrices. After the mapping step, the communication between two SW-Cs might change. It can now be either local if both SW-Cs remain on the same ECU and remote if they are mapped to different ECUs. In the latter case the communication goes over the RTE, through the BSW communication stack and the network bus (see Figure 2.6).

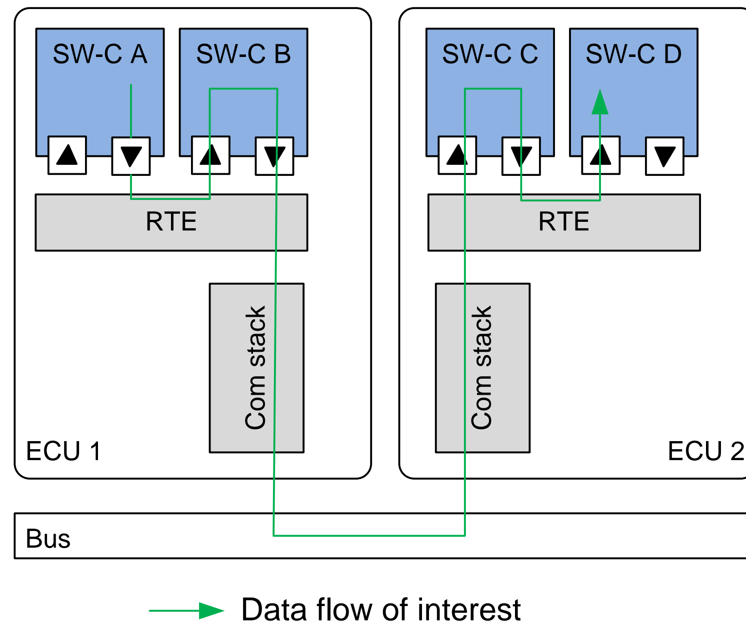


Figure 2.6: System Timing

BswTiming: The BswTiming view addresses timing details related to the Basic software internal behavior of a single basic software module description. According to the AUTOSAR methodology, a BSW module description is generated for each BSW module during the ECU configuration phase. BswTiming is similar to SwcTiming (see Figure 2.7) except for the fact that it deals with the BSW module entities (BSWME) instead of dealing with SW-C internal behavior. Therefore analogously, BswTiming focuses on the activation, start and end of the execution of the BSW module entities.

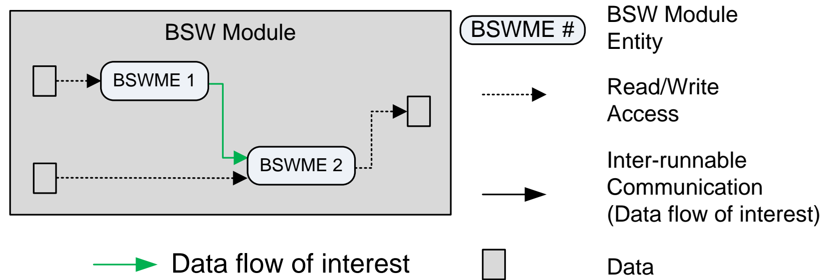


Figure 2.7: Basic Software Timing

EcuTiming: The EcuTiming view addresses the timing description of all software component instances deployed on a specific ECU. Additionally, ECU related interaction is considered including bus communication and so forth. This view is comparable to the SystemTiming view except for that the focus lies on one specific ECU. The information is attached to the ECU configuration description, which comprises information containing ECU related extract from the system configuration. Figure 2.8 shows an example data flow in the ECUTiming view.

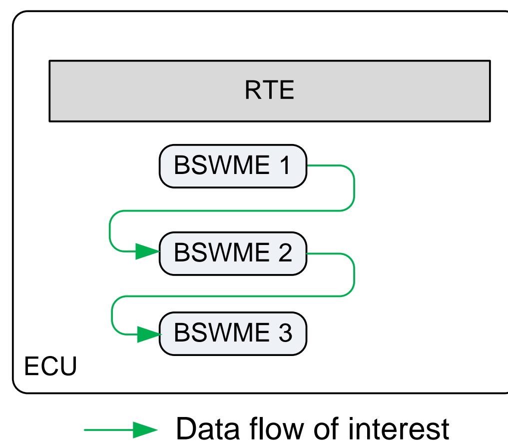


Figure 2.8: ECU Timing

2.2.3 Timing Modeling with TADL2

The Timing Augmented Description Language 2 (TADL2) [80] is an outcome of the ITEA2 [1] project TIMMO-2-USE [49]. TADL2 is an extension of Timing Augmented Description Language (TADL), which was defined in the TIMMO project [90].

[1] ITEA is the EUREKA Cluster programme supporting innovative, industry-driven, pre-competitive R&D projects in the area of Software-intensive Systems & Services (SiSS). (https://itea3.org/)

The goal of these projects was to master different types of timing requirements, and deal with dynamic behavior of complex real-time automotive systems throughout the different phases of the design process. TADL2 differs from TADL in numerous ways: it introduces a clear semantic and adds new concepts for multiple time bases, symbolic time expression and probabilistic timing. In fact, the first version of the AUTOSAR timing model was defined based on TADL, and during TIMMO-2-USE, a harmonization took place between TADL2 and the AUTOSAR timing extensions.

Moreover, TADL2 defines a set of basic and derived timing constraints in hand with their syntax and semantic to avoid unambiguous interpretation. These syntax and semantic definitions are complemented with a metamodel. It introduces the notion of event occurrences in a running or simulated system to specify the timing properties. These events reference to discrete observable events or event chains of the system under investigation.

Event Occurrences and Event-Chains

Any form of state change in a running system that can be constrained with respect to time is represented as an event (*Event*). The event takes place at distinct points in time, and these points are called *occurrences* of the event. This means, a running system can be observed by identifying the forms of state changes that need to be monitored. The times when the changes occur can then be logged for each observation point.

An event chain (*EventChain*) is a container for a pair of events, that must be causally related.

This notion of observation also applies to a hypothetical predicted run of a system or a system model. From a timing perspective, the only information that needs to be in the output of such a prediction is a sequence of times for each observation point, indicating the times at which each event is predicted to occur.

In system models, events appear syntactically as names indicating the state changes of interest. Semantically, an event name is a variable standing for some statically unknown set of occurrences. Events represent state changes that can be observed when a system is executed, or simulated, or perhaps only mathematically predicted.

In the TADL2 semantics, an occurrence is a *timestamp* expressed using a real value. However, for more complex timing constraints, TADL2 also makes use of an additional information to specify an occurrence.

A color annotation can also be used. The color can be set by the producer of an event, and may be utilized during the verification process to identify

TADL2 Notation	Description
$a \wedge b$	constraint a and constraint b are both true
$a \Rightarrow b$	a is false or a and b are both true
$a \Leftrightarrow b$	a and b have the same truth value
$\forall x : c$	c is true for all possible values of x
$\exists x : c$	c is true for at least one value of x
$\lvert X \rvert$	number of elements in set X
$X \subseteq Y$	all elements in X are also in Y
$\forall x \in Y : c$	for each x in Y, c is true
$\exists x \in Y : c$	there is an x in Y such that c is true
$X \leq Y$	X is a subsequence of Y
$x \leq y$	x occurred earlier than y
$x = Y(i)$	x is the element number i in Y counting from zero
$[X]$	set of all occurrences between smallest and greatest occurrences in X
$\lambda([X])$	the length of the continuous intervals in X
$\lambda(X)$	the total length of all continuous intervals in X

Table 2.1: Notation used in the definition of the TADL2 semantics

related occurrences. Colors are drawn from some abstract, possibly infinite types whose only restriction is that it must support an equality test on its values. Thus, an event occurrence can also be semantically expressed using a (timestamp, color) pair, although in some contexts the color information might not be necessary.

Table 2.1 gives a list of operators used in the TADL2 notation. Syntactic and semantic objects like events, constraints and time are referenced by simple variable names such as ”c” for constraint, ”x” for an event occurrence, or ”Y” for a set or sequence of event occurrences.

The TADL2 provides a large set of constructs that can be used to constrain time occurrence of events. These constructs basically specify the restrictions for:

- recurring delays between a pair of events,
- repetition of a single event,
- and synchronicity of a set of events.

ArithmeticExpression and TimingExpression

An *ArithmeticExpression*, denoted by aexp, is a term built from literals, arithmetic variables and arithmetic operators. It stands for a value in a set of real

numbers extended with positive and negative infinity. The grammar notation used here is the standard BNF form, with keywords in boldface and non-terminal symbols written using lower-case names.

A *TimingExpression* is identical to an arithmetic expression. However, its grammar is extended to allow expressions to use alternative time bases and a variety of time units.

TADL2 constraints

The concepts defined by TADL2 were taken as the basis for our approach for dynamic assertion-based verification defined in Chapter 5. This section will only highlight some of the timing constraints defined by the TADL2. A more detailed description of the timing constraints chosen for our approach will be given in Chapter 5.

Basically, TADL2 defines a bunch of basic and complex forms of timing constraints. Some of the basic timing constraints are for instance the so-called *RepeatConstraints* and *DelayConstraints*. Most of the complex timing constraints can be expressed as a composition of other constraints. Examples of such complex timing constraints are the so-called *SynchronizationConstraint* and the *SporadicConstraint*.

Moreover, one basic attribute used for the definition of the timing constraints is the so-called *span* attribute. The *span* attribute is used to define the index of the correlated event occurrences.

The following descriptions of timing constraints are unchanged excerpts from [80]. Furthermore, the shaded areas in the respective figures highlight the event occurrences that are subject to the constraint specification.

The RepeatConstraint

A RepeatConstraint describes a repeated distribution of occurrences of a specific event. The semantic and attributes of this timing constraint are shown in Table 2.2: This constraint defines the basic notion of repeated occurrences. For a span attribute equal to 1 and for identical lower and upper attributes, the accepted behavior must be strictly periodic. Further, if the *span* value is 1 but the value of *lower* is strictly smaller than the value of *upper*, then the accepted behavior may deviate from a periodic one, making the time window within which each event occurrence may appear as wide as $upper - lower$ time units. For a span value greater than 1 the constraint should be applied to each span+1 event occurrence, but places no restriction on the distances within shorter sequences [80].

	RepeatConstraint
Attributes	event: Event lower: TimingExpression upper: TimingExpression span: int
Semantic	$\forall X \leq event : \lvert X \rvert = span + 1 \Rightarrow lower \leq \lambda([X]) \leq upper$

Table 2.2: Semantic definition of the TADL2 RepeatConstraint.

Figure 2.9 depicts a set of event occurrences satisfying a RepeatConstraint with span of 2. The corresponding TADL2 formula can be expressed as follows: $\forall X \leq event : \lvert X \rvert = 3 \Rightarrow lower \leq \lambda([X]) \leq upper$, where the values of the attributes lower and upper are to be defined. $\lvert X \rvert$ denotes the number of elements in the sub-sequence X of event occurrences and $\lambda([X])$ denotes the length of the time interval in X. As it can be seen in the figure, only the

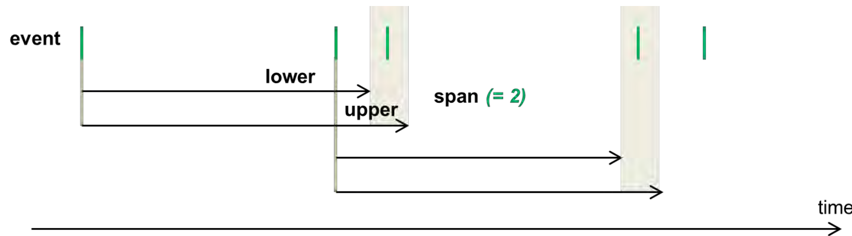


Figure 2.9: A set of event occurrences satisfying a RepeatConstraint with span of 2

last event occurrence in the sub-sequence X is constrained. Furthermore, the timing point of that occurrence can be referred to as $\lambda[X]$.

The DelayConstraint

Looking at the communication between two components (a source and a target), a DelayConstraint specifies timing bounds between an event occurrence at the source and a corresponding event occurrence at the target, so that each event occurrences at the source must be matched by an event occurrence at the target [80].

	DelayConstraint
Attributes	source: Event target: Event lower: TimingExpression upper: TimingExpression
Semantic	$\forall x \in source : \exists y \in target : lower \leq y - x \leq upper$

Table 2.3: Semantic definition of the TADL2 DelayConstraint.

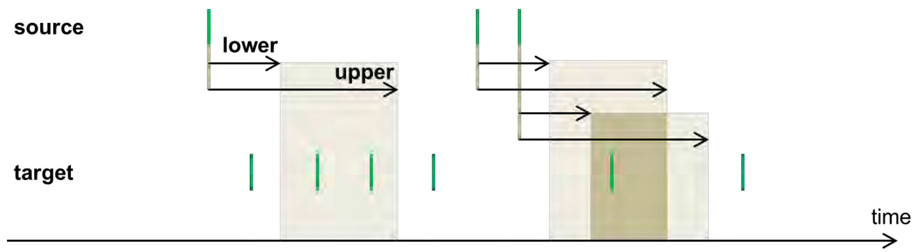


Figure 2.10: A set of event occurrences satisfying a DelayConstraint [80]

This notion of delay is entirely based on the duration between source and target occurrences. One-to-many and many-to-one source-target patterns are also possible. Figure 2.10 shows an example of a set of events satisfying a DelayConstraint. The formal definition on the DelayConstraint can be seen in Table 2.3.

The SynchronizationConstraint

A SynchronizationConstraint describes how tightly the occurrences of a group of events shall follow each other. This form of timing constraint only speci-

	SynchronizationConstraint
Attributes	event: Event[2..*] tolerance: TimingExpression = infinity
Semantic	$\exists X : \forall i : DelayConstraint(X, event_i, 0, tolerance)$ $\wedge DelayConstraint(event_i, X, -tolerance, 0)$

Table 2.4: Semantic definition of the TADL2 SynchronizationConstraint. [80]

fies a time frame or tolerance window of each occurrence of a group of events. The length of the tolerance window is specified by the tolerance attribute.

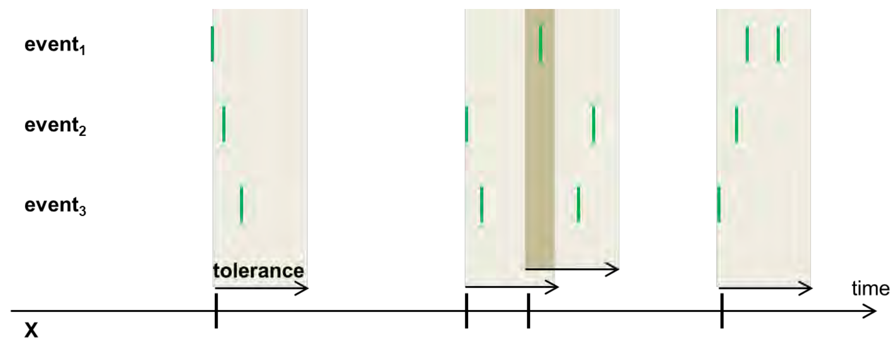


Figure 2.11: A set of target event occurrences satisfying a SynchronizationConstraint. Overlapping tolerance windows and multiple event occurrences [80]

Figure 2.11 shows a generic example of a group of event occurrences which satisfy a SynchronizationConstraint. As shown in the figure, events may oc-

cur multiple times within the same tolerance window and the windows may also overlap and thus share event occurrences. [80]

The formal definition on the SynchronizationConstraint can be seen in Table 2.4. As it can be seen in the table, the SynchronizationConstraint is a variant of the DelayConstraint that adds a tolerance attribute and changes the target event link into a list of events. A system behavior satisfies a SynchronizationConstraint if and only if there is a set of times X such that for all events $event_i$, the same system behavior concurrently satisfies
$DelayConstraint(X, event_i, 0, tolerance)$ and
$DelayConstraint(event_i, X, -tolerance, 0)$.

2.2.4 Testing and Verification

The main purpose of a verification task is to check whether or not a model or system is consistent with respect to its specification or requirement. Furthermore, verification activities are performed several time during the design phase according to the verification plan.

In the context of the verification of electronic design, a multitude of approaches have been proposed and are being applied in the EDA industry. Basically, these approaches can be sorted in two main categories. Some of them are static and others are dynamic.

Static verification techniques perform a thorough analysis on a certain aspect of the design model. Static verification characterizes verification techniques, where the design model under verification is investigated in an analytical way by means of formal specifications. Theorem proving, Equivalence Checking and Model Checking [13, 59] belong to this category.

Dynamic verification is usually associated with the simulation of the design model at a certain abstraction level. Although simulation is not enough to prove consistency of formal entities, as it is the case for formal verification methods, it can be applied in situations where the model under investigation cannot be formalized in a way as requested by the respective formal method. Further, simulation-based based verification is typically performed using a testbench as simulation environment. There are several methodologies specifying how to build such a testbench [20, 74, 7, 8].

The thesis focuses on dynamic verification in the context of Model-Based Design (MBD). Throughout the different phases of MBD, several levels of modeling of both the plant and controller are required, in order for the functional behavior of the model to match that of the generated code [10].

To reduce development time and introduce technologies faster to the market, many companies have been turning more and more to MBD. In MBD, the development process centers around a system model, from requirements capture and design to implementation and test.

Traditional design paradigms in the automotive industry often delay control system design until late in the process. In some cases, control system design requires several costly hardware iterations. To reduce costs and improve time to market, emphasis is placed on modeling and simulation as early as possible in the development process [97].

Different steps are supported by a variety of approaches from Model-In-the-Loop (MIL), to Software-In-the-Loop (SIL), HIL, Rapid-Control-Prototyping (RCP), or Component-In-the-Loop (CIL). Each process is used to address different stages of the development process. The first step is the simulation, where neither the controller nor the plant operates in real-time. This step, usually used toward the beginning of the process, allows engineers to study the performance of the system and design the control algorithm(s) in a virtual environment, running computer simulations of the complete system, or subsystem.

RCP provides the engineer with the ability to quickly test and iterate their control strategies on a real-time computer with actual hardware. The control strategy is simulated in real-time on a processor that augments or replaces the real embedded controller, allowing the user to investigate and refine the control algorithms while operating with the real system under control. RCP is now a commonly used method to develop and test control strategies.

In the SIL phase, the actual Production Software Code is incorporated into a mathematical simulation that contains models of the Physical System. This is done to permit inclusion of software functionality for which no model(s) exists or to enable faster simulation runs.

HIL is a technique for running a mathematical simulation model of a system on a real-time computer, integrated with actual controller hardware and software, such that the controller acts as though it were integrated into the real system. This is used for testing and validation of embedded electronic controllers, prior to testing in a vehicle.

CIL is used to assess the impact of an entire system (e.g., engine plant and control) on other portions of the system (e.g., vehicle). The system to be analyzed is real hardware while the rest of the components are emulated from models.

2.3 Automotive Vehicle Netwoks

2.3.1 Controller Area Network (CAN)

The Controller Area Network (CAN) [17, 73, 89] is an International Standardization Organization (ISO) standard. The CAN standard is a serial communications bus originally developed for the automotive industry to replace the complex wiring harness with a two-wire bus. Its domain of application ranges from high speed networks to low cost multiplex wiring. In automotive electronics, engine control units, sensors, anti-skid-systems, etc. are typically connected using a CAN bus with transmission rates ranging from 10 kbit/s up to 1 Mbit/s. The transmission rate depends on the bus segment. For instance, 1 Mbit/s can only be reached at a maximum bus length of 40 meters.

The CAN protocol has a high immunity to electrical interference, and includes sophisticated mechanisms for self-diagnosis, error-detection and error handling mechanism. These features have led to CAN's popularity in a variety of industries beyond the automotive industries, such as: marine, medical, manufacturing, and aerospace.

Furthermore, CAN is a multi-master bus with an open, linear structure with one logic bus line and equal nodes. The number of nodes is not limited by the protocol, and the bus nodes do not have a specific address. Instead, the address information is contained in the identifiers of the transmitted messages (CAN Frame), indicating the message content and the priority of the message.

Moreover, CAN supports multicasting and broadcasting. The error-detection and error handling mechanism basically consists of automatically retransmitting erroneous messages, that have been detected. Temporary errors are recovered. Permanent errors are followed by autonomous deactivation of the defect nodes. This approach therefore guarantees a system-wide data consistency.

The bus access is handled via the advanced serial communications protocol Carrier Sense Multiple Access/Collision Detection with Arbitration on Message Priority (CSMA/CD+AMP). CSMA means that each node on a bus must wait for a prescribed period of inactivity before attempting a message transmission.

In the CD+AMP communication protocol, collisions are resolved through a bit-wise arbitration, based upon a configured priority of each message in the identifier field of a CAN Frame. The higher priority identifier always wins bus access. In the following section, the structure of CAN frame will be described.

The CAN frame

The first version of the CAN standards was the so-called Low-Speed CAN (ISO 11519). This version was intended for applications up to 125 kbit/s with a standard 11-bit identifier. The following version, ISO 11898 with the same indentifier length was later standardized in 1993. However, this version provides a transmission rate ranging from 125 kbit/s to 1 Mbit/s. The more recent amendment of the second version (ISO 11898) was published in 1995. This version introduces an extended 29-bit identifier. The ISO 11898 11-bit version is often referred to as Standard CAN Version 2.0A, while the ISO 11898 amendment is referred to as Extended CAN Version 2.0. The structure of the standard CAN frame format can be seen in Figure 2.12, while Figure 2.13 depicts the various part of the extended CAN frame format [17, 89].

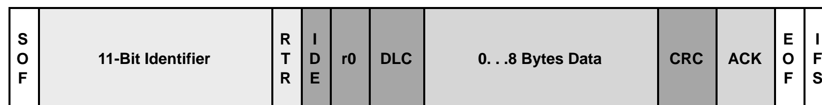


Figure 2.12: Standard CAN frame with 11-Bit identifier

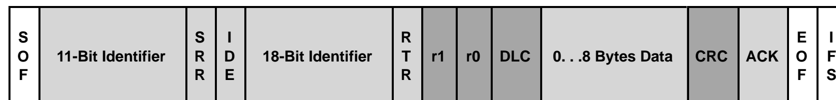


Figure 2.13: Extended CAN frame with 29-Bit identifier

CAN Network Topology

The CAN protocol operates on a broadcast bus. There is no built-in support for other topologies. However, third-party solutions with gateways or switches for star topologies exist [73]. In advanced applications, such as an automotive system, several CAN bus segments are interconnected using one or more gateways. A gateway is provided with software that knows which data should be forwarded between the different bus segments it is connected to.

2.3.2 FlexRay

FlexRay is a fast, deterministic and fault-tolerant vehicle network bus standard for automotive use. This standard was developed by a consortium of

well-known OEMs of the automotive industry. The consortium was disbanded in 2010 after the release of the actual protocol specification version 3.0. The core members of the consortium were: BMW, Bosch, Daimler-Chrysler, General Motors, Ford, NXP Semiconductors, Philips Semiconductor, and Volkswagen. In the scope of this thesis, the version 2.1 Revision A [41] was used for validation purposes.

Communication schedule

The FlexRay protocol is based on a periodically recurring communication cycle. Within one communication cycle the standard offers the choice between two media access schemes, that is: a static Time Division Multiple Access (TDMA) scheme, and a dynamic mini-slotting based scheme referred to as Flexible Time Division Multiple Access (FTDMA). Furthermore, the configuration of a FlexRay network schedule has to be done at design time. Then, once configured the bus communication schedule cannot be changed at system run-time.

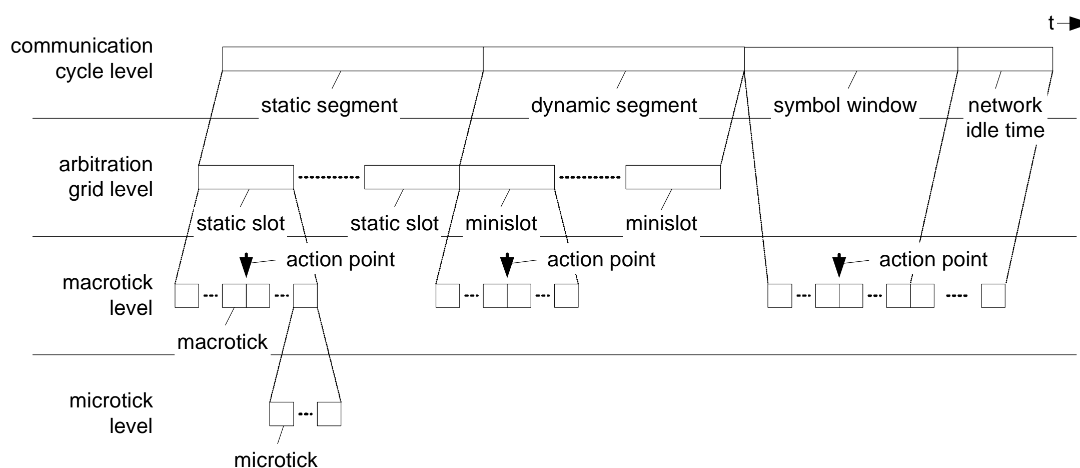


Figure 2.14: Timing hierarchy within the communication cycle [41]

The communication cycle is a fundamental notion for the media access scheme. As illustrated in Figure 2.14, the FlexRay communication cycle can be broken into four segments. The segments are the *static segment*, the *dynamic segment*, the *symbol window* and the *Network Idle Time*. Furthermore, the timing hierarchy can be decomposed in four timing hierarchy levels (see Figure 2.14).

Within the static segment, communication is arbitrated based on the TDMA scheme. Whereas FTDMA is applied in the dynamic segment to arbitrate transmission. The symbol window is a communication period within which special control information can be transmitted on the network. The network idle time is a communication-free period that concludes the communication

cycle. This period can be used by the nodes (ECUs) for synchronization or internal computation for instance.

The next lower level, the arbitration grid level, contains the arbitration grid that forms the backbone of FlexRay media arbitration. In the static segment the arbitration grid consists of consecutive time intervals, called static slots, in the dynamic segment the arbitration grid consists of consecutive time intervals, named minislots.

The arbitration grid level builds on the so-called *macrotick level* that is defined by the so-called *Macrotick*. Designated macrotick boundaries are called action points. These are specific instants at which transmissions shall start in all segments of the communication cycle. In the dynamic segment, the action point also specifies the instant at which a transmission shall end. For more detailed information about the remaining levels, please refer to [41].

FlexRay frame format

The FlexRay protocol supports a data transmission rate of up to 20 Mbit/s. Figure 2.15 depicts the structure of a FlexRay frame. As depicted in the figure, a FlexRay frame consists of three main parts. These parts are the header, the payload segment containing up to 254 bytes of data, and the Trailer segment containing CRC information of 24 bits.

The header of 5 bytes includes the identifier of the frame and the length of the data payload. The use of the identifier field allows to move a functionality implemented by a software component, that generates a frame X, from one ECU to another ECU without having to change the configuration of the receiving ECU. The configuration of the sending node needs to be modified since frame are dedicated to specific communication slots.

However, this is only possible for configurations, where messages produced by distinct software components are not packed into the same frame. In FlexRay, different messages can be packed into a single Frame for the purpose of saving bandwidth. This procedure is referred to as frame-packing or protocol data unit (PDU)-multiplexing (see [41]).

FlexRay network topology

As aforementioned, the FlexRay communication protocol is fault-tolerant. Fault-tolerance can be achieved by using a dual-channel bus network. In a dual-channel configuration, the second channel can be either used as a redundant channel as to increase the system throughput.

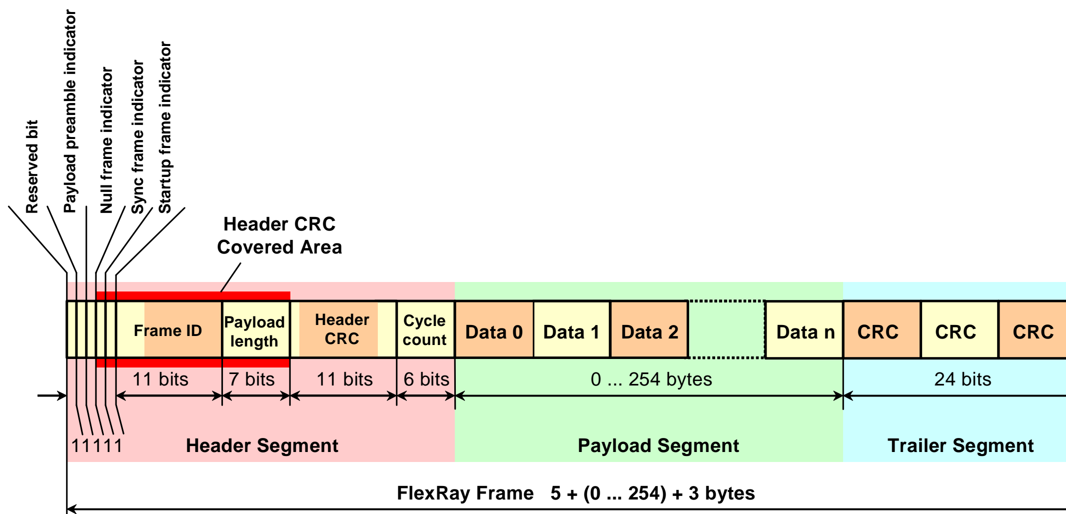


Figure 2.15: FlexRay frame format [41]

FlexRay supports two basic network topologies, namely, the bus- and star-network. Both topologies can be applied to single and dual channel configurations. Furthermore, various hybrid combinations of these topologies are also possible [73, 12, 84, 41].

There is a large number of possible hybrid topologies: Figure 2.16 depicts an example for one type of hybrid topology. The example shows a cluster of seven communication nodes denoted by Node A, B, C, D, E, F, G. The nodes A, B, C, and D are connected using point-to-point connections to a star coupler. Whereas, the remaining nodes (Node E, F, and G) are connected to each other using a bus topology. This bus is also connected to the star coupler, allowing nodes E, F, and G to communicate with the other nodes.

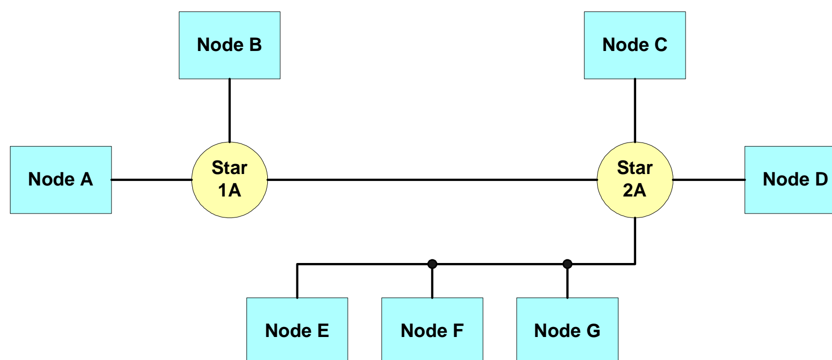


Figure 2.16: Active star topology combined with a passive bus topology [41]

2.4 Design of Electronic Systems

2.4.1 Design Modeling with IP-XACT

IP-XACT is an IEEE standard exchange format originally developed by the SPIRIT consortium [48]. The standard comes along with an XML schema definition that prescribes how to document electronic designs and components in order to facilitate automated configuration and integration of Intellectual Propertys (IPs) in a tool chain.

Its current version was approved as IEEE 1685-2009. The typical structure of an IP-XACT design environment is illustrated in Figure 2.17. IP-XACT

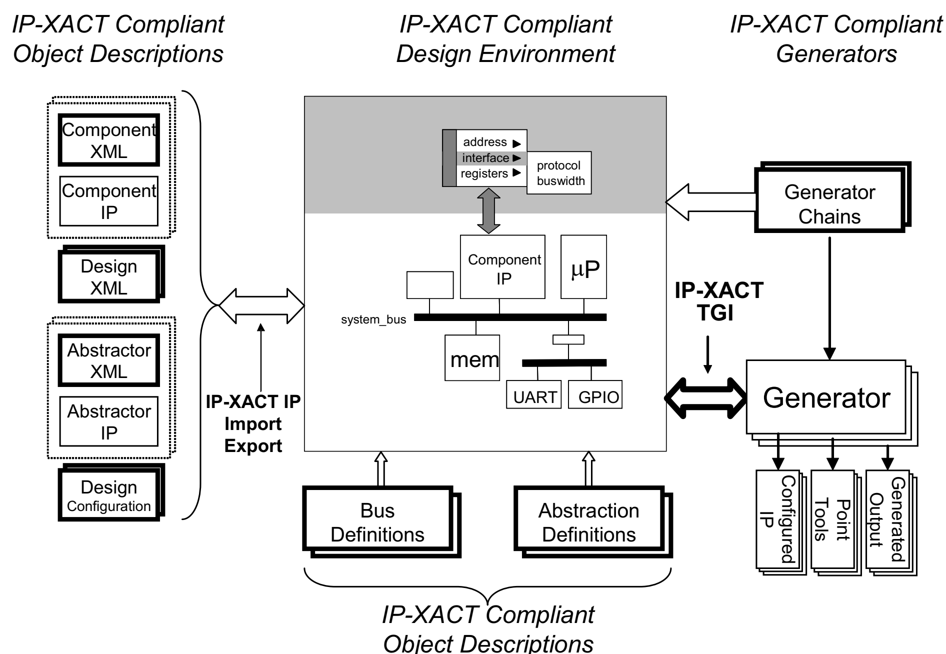


Figure 2.17: IP-XACT design environment (source: [48])

can be used for design capture, which is the documentation of design configuration and design intent. It can also be used to build a design, which is the creation of a design (or design model) [48].

The four central elements defined by the standard are bus definition, abstraction definition, component and design. This well-defined data format enables the creation of vendor-independent tools for automated IP- reuse and integration. Figure 2.18 depicts the basic architecture of an IP-XACT design.

Bus Definition and Abstraction Definition

The bus definition element is used to describe high level aspects of a bus while abstraction definitions describe low level aspects. The bus definition defines the protocol. It defines whether connections of a bus interface can be direct or not. Direct means that a master bus interface, for example, can be connected directly to a slave interface, and not direct means that an additional decoding or adaption is required. An abstraction definition defines the level of abstraction like for example RTL or SystemC.

IP-XACT Component

An IP-XACT component description can define one or more views of its implementation, or an interconnect infrastructure in the form of a bridge or a channel. RTL and TLM are the predefined model views but user-defined views can also be added. As depicted in Figure 2.18 the basic structure of an IP-XACT component includes a model, a bus interface and a portmap.

The model element is used to describe the views, the physical ports and the configuration parameters of a component. A model may have more than one view and each view element specifies a representation level of the component (e.g.: TLM).

A component may have multiple bus interfaces of the same or different types. Each busInterface element defines properties of this specific interface in a component. Among other attributes of the busInterface, the busType should be defined. It specifies the bus definition that this bus is referring to. It can also specify the abstraction definition where this bus interface is referenced.

The portmap element is actually part of the busInterface and is used to map the component's physical ports to the corresponding abstraction definition's logical ports.

IP-XACT Port

IP-XACT components might have an unbounded list of ports. A single IP-XACT port element can be specified either as wire or transactional. Wired ports are typically used for RTL model and transactional ports for TLM. IP-XACT provides a list of implementation constraints that can be specified for wired ports. These constraints can be used to document requirements to be fulfilled by an implementation of an RTL component. The constraints are grouped into so-called constraintSets.

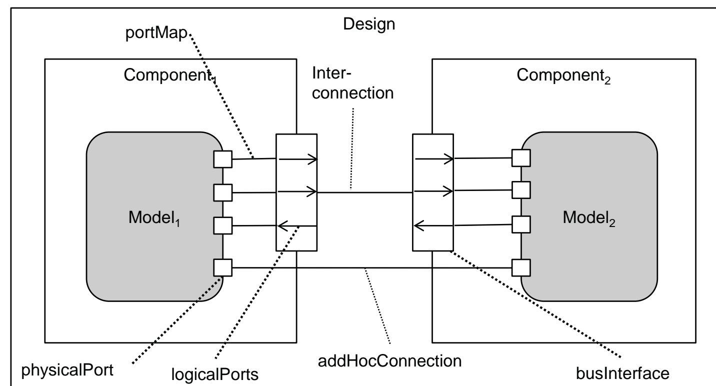


Figure 2.18: IP-XACT simplified Design structure.

Three different types of constraints can be optionally specified, namely driveConstraint, loadConstraint and timingConstraint. A timingConstraint specifies a delay constraint and is always relative to a certain clock. Each constraints can be further associated with different model views of the component (see [48]).

The timingConstraint specifies a delay type relative to a clock for the associated port. The delay type restricts the constraint to applying to only best-case (minimum) or worst-case (maximum) timing analysis. By default, the constraint can be applied to both. The delayType attribute may have two values min or max.

A component port can be specified as transactional if it uses or implements a service. A service can be implemented with functions or methods. In contrast to wired ports, transactional ports do not have timing constraints. Thus the current version of the IP-XACT schema does not provide means to define timing constraint at TLM.

Figure 2.19 depicts the structure of an IP-XACT Ports element. As it can be seen, the ports element is an unbounded list of port elements. Each port element defines the logical port information for the containing abstraction definition. It contains the following elements:

logicalName (mandatory) gives a name to the logical port that can be used later in component description when the mapping is done from a logical abstraction definition port to the components physical port. The logicalName shall be unique within the abstractionDefinition. The type of this element is Name.

displayName (optional) allows a short descriptive text to be associated with the port. The type of this element is string.

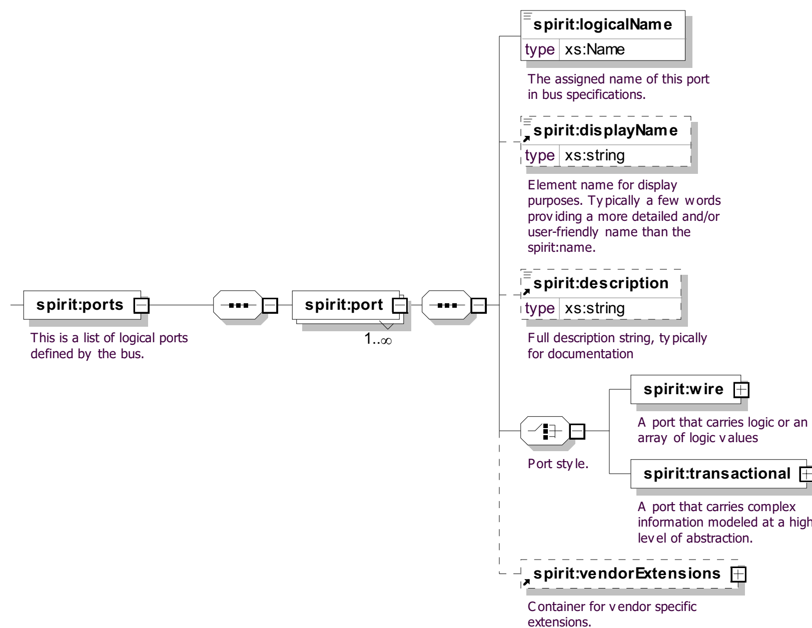


Figure 2.19: Tree view of the xml schema definition of an IP-XACT Ports

description (optional) allows a textual description of the port. The type of this element is string.

Each port also requires a **wire** element or a **transactional** element to further describe the details about this port. A wire style port is a port that carries logic values or an array of logic values. A transactional style port is a port that carries any other type of information, typically used for TLM.

vendorExtensions (optional) contains any extra vendor-specific data related to the port. The vendorExtensions element is a place in the description in which any vendor specific information can be stored. The vendorExtensions element allows any well-formed description.

IP-XACT Design

Figure 2.18 shows the basic structure of an IP-XACT design. The design element can be used as top level element for the assembly of component instances. A design contains a description of components including their configuration and interconnections. Hereby, three kinds of connection are available:

- Interconnections that describe connections between subcomponent's interfaces,

- AddHocConnections that describe connections between two components pins,
- HierConnections that describe connections between a busInterface of a sub component and the bus interface of the encompassing component.

For further details please refer to [48].

2.4.2 Design Modeling Language with SystemC

SystemC [53] was defined by the Open SystemC Initiative (OSCI), and is now promoted by the Accellera Systems Initiatives, and has been approved by the IEEE Standards Association as IEEE 1666-2005 [55].

SystemC is a system design and modeling language. This language was developed to meet the system designer's requirements for designing and integrating complex electronic systems very quickly while assuring that the final system will meet performance expectations [14].

The application areas are system-level modeling, architectural exploration, performance modeling, software development, functional verification, and high-level synthesis. Therefore, SystemC facilitates the co-development of hardware and software within a tight schedule. Thorough functional and architectural verification is required to avoid expensive and sometimes catastrophic failures in the device. Because of these features, the SystemC language is currently the de-facto industry standard for Electronic System-Level (ESL) design among the C++ based SLDLs.

Basically, SystemC builds on top of C++ and comes with a set of libraries like the SystemC Verification Library (SCV) and a discrete-event simulation kernel. Additionally, the SystemC language covers several levels of abstraction from Transaction Level Modeling (TLM) down to Register Transfer Level (RTL). Meanwhile, the interest of using SystemC has grown in the automotive industry. For instance, the affinities of some concepts of AUTOSAR and SystemC were outlined in [63].

Architecture of the SystemC language

As aforementioned, SystemC supports the modeling of both hardware and software. The primary application area for SystemC is the design of electronic systems. However, the language also provides generic modeling constructs that can be applied to non-electronic systems [15].

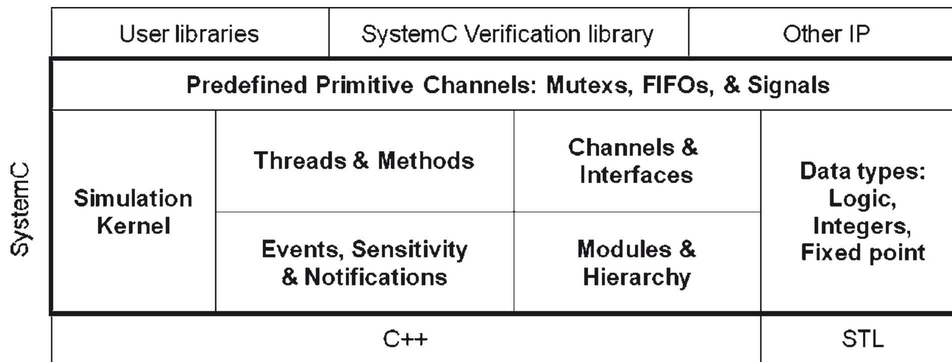


Figure 2.20: SystemC language architecture [15]

The overall architecture of the SystemC class library can be seen in Figure 2.20. The simulation kernel is the heart of the implementation. In fact, it's a lightweight scheduler responsible for the activation and suspension of SystemC processes (Threads and Methods). Furthermore, the architecture of SystemC incorporates an event mechanism, which forms the basis for synchronization. Both the simulation kernel and the event mechanism build the foundation for the communication elements: interfaces, channels, and ports.

SystemC provides means to separate functionality from communication. Functionality is implemented in Modules, whereas communication of implemented in channels. The communication mechanism is based on the Interface-Method-Call (IMC) scheme. Essentially, ports can only access channels through interfaces. An interface describes a collection of a fixed set of communication methods and a channel implements one or more interfaces. Moreover, hierarchical and other user-defined channels can be built on the top layer.

SystemC constructs

Figure 2.21 highlights the most important concepts provided by SystemC to model a component. A component can be represented in SystemC using a SystemC module (SC_MODULE). In practice, a typical SystemC module will not contain all of the illustrated concepts. The figure shows a SystemC module that may contain instances of other modules. Further, a SystemC method (SC_METHOD) and a SystemC thread (SC_THREAD) can also be defined within a SystemC module. Communication between modules and simulation processes is realized through various combinations of ports, interfaces, and channels. Coordination among simulation processes is also accomplished by means of events [15].

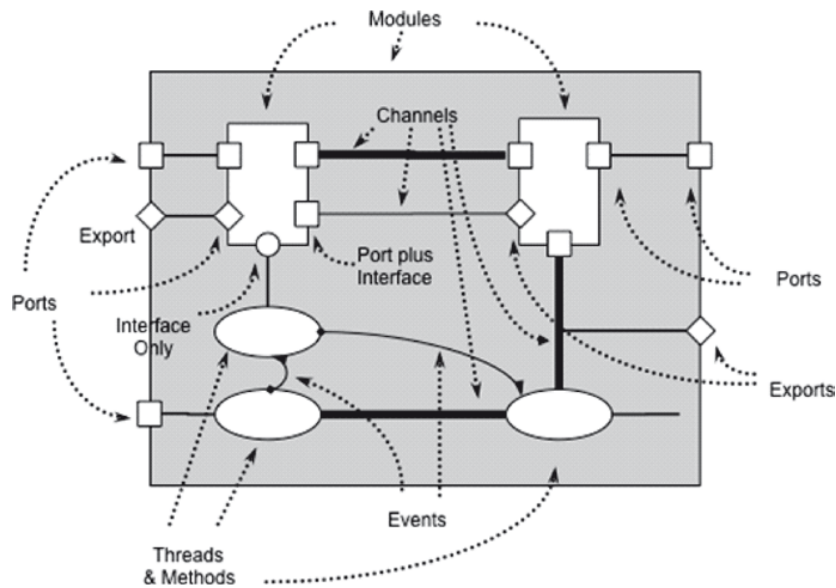


Figure 2.21: Artifacts of SystemC [15]

SystenC simulation kernel

SystemC uses a co-operative multitasking model where an executing process cannot be pre-empted by any other process. A SystemC process hands over control back to the scheduler either by explicitly calling a wait function in the case of a thread (SC_THREAD), or returning to the kernel in the case of a method (SC_METHOD).

Figure 2.22 gives a brief overview of the SystemC simulation kernel. It coordinates the communications among all components (see Figure 2.21). The SystemC simulator has two major phases of operation: elaboration and execution. A third, often minor, phase occurs at the end of the execution; this phase could be characterized as post-processing or cleanup. Execution of statements prior to the sc_start() function call are known as the elaboration phase. This phase is characterized by the initialization of data structures, the establishment of connectivity, and the preparation for the second phase, execution.

The execution phase hands control over to the SystemC simulation kernel, which orchestrates the execution of processes in order to create an illusion of concurrency. After the sc_start() function call, the simulation processes are invoked in a random order during initialization.

After initialization, simulation processes are run when events occur to which they are sensitive. The SystemC simulator implements a *cooperative multitasking* environment. Once started, a running process continues to run until

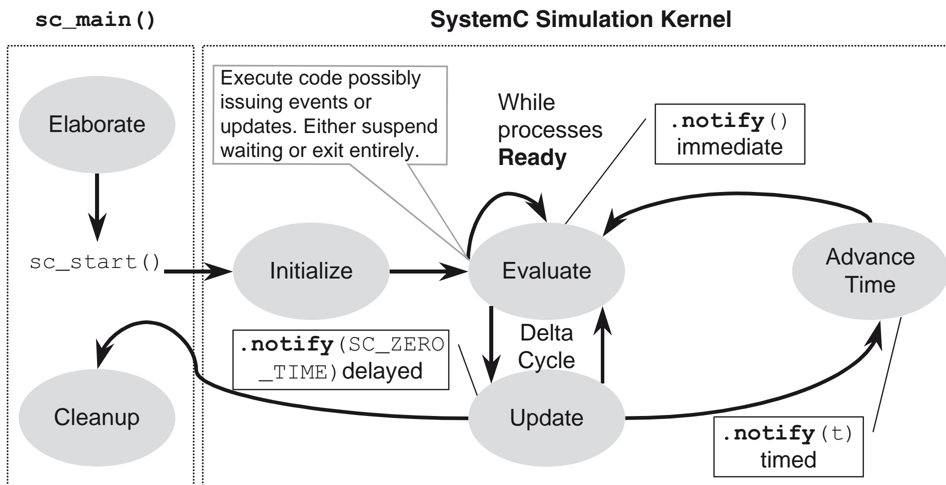


Figure 2.22: SystemC simulation kernel [15]

it yields control. Several simulation processes may begin at the same instant in simulator time.

The simulation kernel of SystemC follows the evaluate-update paradigm that is common in Hardware Description Languages (HDLs). In this case, all simulation processes are evaluated and then their outputs are updated. An evaluation followed by an update is referred to as a *delta cycle*. The concept of delta cycles, where multiple evaluate-update phases can occur at the same simulation time, is supported.

If no additional simulation processes need to be evaluated at that instant (as a result of the update), then simulation time is advanced. When no additional simulation processes need to run, the simulation ends [15]. For a full description of the SystemC simulation kernel please refer to [55, 15].

Transaction-level modeling: TLM-2.0

TLM is a high-level modeling approach widely used in the field of digital systems. The basic idea behind TLM is to separate details of communication among modules from the details of functional implementation.

Models can be categorized according to a range of criteria, including granularity of time, frequency of model evaluation, functional abstraction, communication abstraction and use cases.

As depicted in Figure 2.23, the TLM-2.0 standard defines a variety of use cases and coding styles. The coding styles are appropriate for, but not mandatory to, the various use cases. Additionally, the standard provides a set of programming mechanisms for the implementation of the coding styles.

The definitions of the standard TLM-2.0 interfaces are independent from the descriptions of the coding styles. The TLM-2.0 interfaces form the normative part of the standard and ensure interoperability.

Each coding style can support a range of abstraction across functionality, timing and communication. In principle, users can create their own coding styles.

An untimed functional model consisting of a single software thread can be written as a C function or as a single SystemC process, and is sometimes termed as algorithmic model. Such a model is not transaction-level per definition, because by definition a transaction is an abstraction of communication, and a single-threaded model has no inter-process communication.

A transaction-level model requires multiple SystemC processes to simulate concurrent execution and communication.

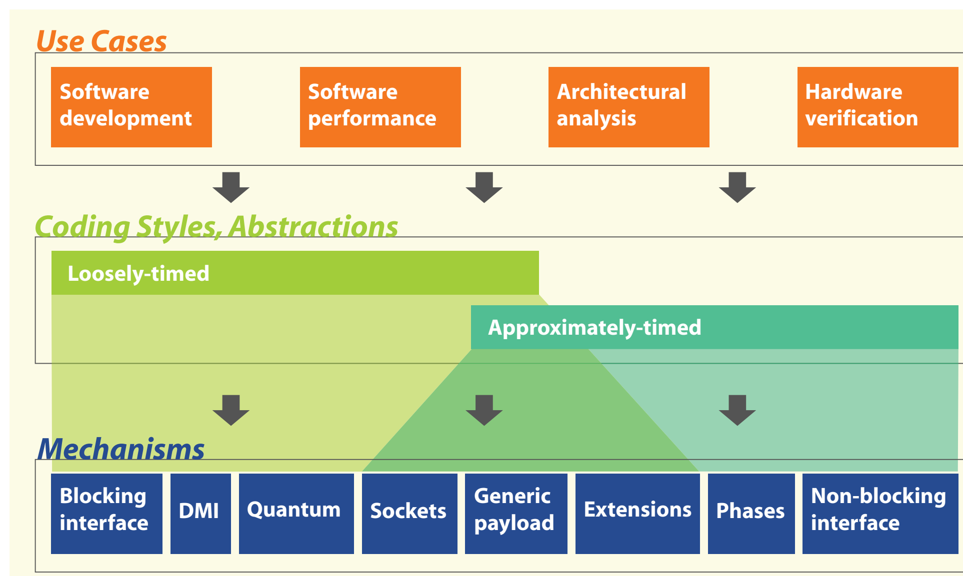


Figure 2.23: TLM2 Use Case, Codingstyles and Mechanisms [96]

Synchronization may be strong in the sense that the sequence of communication events is precisely determined in advance, or weak in the sense that the sequence of communication events is partially determined by the detailed timing of the individual processes. Strong synchronization is easily implemented in SystemC using FIFOs or semaphores, allowing a completely untimed modeling style where in principle simulation can run without advancing simulation time.

Untimed modeling in this sense is outside the scope of TLM-2.0. On the other hand, a fast virtual platform model allowing multiple embedded software threads to run in parallel may use either strong or weak synchronization. In

this standard, the appropriate coding style for such a model is termed *loosely-timed*.

A more detailed transaction-level model may need to associate multiple protocol-specific timing points with each transaction, such as timing points to mark the start and the end of each phase of the protocol. By choosing an appropriate number of timing points, it is possible to model communication to a high degree of timing accuracy without the need to execute the component models on every single clock cycle. In this standard, such a coding style is termed *approximately-timed* [96, 50].

2.4.3 Formal Property Specification Language with PSL

The PSL is an IEEE standard [52] that has been published in 2005. PSL is based on the Sugar language created at IBM Haifa Research Labs and published in 1994. It was developed to provide engineers a concise syntax and mathematically precise well-defined formal semantics for the specification of design properties. PSL is a declarative language for the formal specification of concurrent systems particularly suitable but not limited to the description of hardware designs. The standard is currently supported by the Accellera Systems Initiative [51].

Furthermore, it provides an interoperable specification language to exchange hardware specifications and develop seamless tool integration. It enables the developer to capture design intent in a verifiable form, while enabling the verification engineer to validate that the implementation satisfies its specification through static or dynamic verification [70].

PSL is a layered, multi-purpose, multi-level and multi-flavor assertion language. At its lowest-level, PSL uses references to signals, variables and values that exist in the design description. This ensures that each component's full range of behavior will be consistent, and apparent to various industry-standard verification tools, as the component moves through the design chain [70, 47].

PSL can express both properties that use linear semantics as well as those that use branching semantics. The first category of properties consists of properties of the PSL Foundation Language, while the second one belongs to the Optional Branching Extension. Properties with linear semantics reason about computation paths in a design and can be checked in simulation, as well as in formal verification. Properties with branching semantics reason about computation trees and can be only checked using formal verification. While the linear semantics of PSL are mostly used in properties, the branching semantic adds an important expressive power. For instance, branching semantics

are sometimes required to reason about deadlocks. In this thesis, we restrict ourselves to the simple subset of PSL, which is its linear-time temporal logic.

PSL layers and flavors

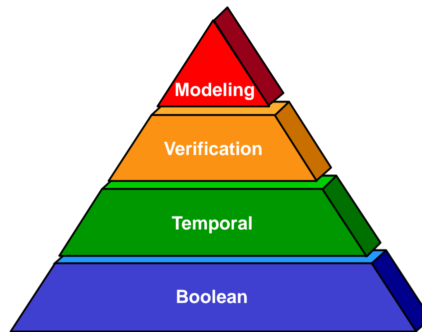


Figure 2.24: PSL is a Layered Language [58]

As shown in Figure 2.24, PSL is structured in four layers: the boolean layer, which contains the boolean expressions used in properties; the temporal layer, which contains the temporal properties and Sequential Extended Regular Expressions (SEREs); the verification layer for directing the use of PSL by a tool; and the modeling layer, for modeling behavior of inputs and auxiliary variables.

Based on this layered structure, several flavors of PSL have been defined. The most generic one is the gdl-flavor, which is based on the General Description Language (GDL). GDL was designed especially for the use in the PSL modeling layer and can be used for modeling systems in diverse problem domains, at various levels of abstraction.

Other PSL flavors are based on HDLs. The supported hardware description languages are SystemVerilog, Verilog, VHDL, SystemC. In each flavor, the Boolean and modeling layers conform to the syntax of the underlying HDL) (or GDL). The temporal and verification layers are not affected by flavors [44].

In the scope of this thesis we make use of the SystemC flavor. In the SystemC flavor, all expressions of the Boolean layer, as well as modeling layer code, are written in SystemC syntax. The SystemC flavor also has limited influence on the syntax of the temporal layer. For further details on the PSL syntax and its various layers, we recommend readers to read the IEEE Standard 1850 [52].

PSL operators

For a given flavor of PSL, the operators of the underlying HDL have the highest precedence. This includes logical, relational, and arithmetic operators of the HDL. Various PSL provide various operators. Each operator has a precedence relative to other flavors operators. In general, operators with a higher relative precedence are associated with their operands before operators with a lower relative precedence.

Operator class	Associativity	Operators
Union operator	left	union
Clocking operator	left	@

Table 2.5: Union and Clocking operators

An HDL name that is a PSL keyword cannot be referenced directly by its simple name in an HDL expression used in a PSL property. However, such a name can be referenced indirectly, using a hierarchical name or qualified name as allowed by the underlying HDL. An overview of the PSL operators is given in the following tables.

The foundation language operator with the next highest precedence after the HDL operators is the *union* operator. This operator is used to indicate a non-deterministic expression. It is followed by the clocking operator (see Table 2.5). The *clocking* operator, can be used to associate a clock expression with a property or sequence (see Section 2.4.3).

Operator class	Associativity	Operators
Consecutive operators	left	[*], [+]
Non-consecutive operator	left	[=]
Goto operator	left	[$\rightarrow$]

Table 2.6: SERE Repetition operators

Repetition operators (as listed in Table 2.6) describe another class of foundation language operators. These kind of operators allow to build more sophisticated SEREs using variations on the SERE repetition operators.

Consecutive repetition operators ([*], [+]) provide a shortcut to typing the same sub-SERE a certain number of times. The [*] operator stands for an arbitrary number of repetitions of the SERE it is applied to, including none. To specify any non-zero number of repetitions the operator [+] should be used. A specific number of repetitions can also be specified if needed. For instance, instead of typing $busy;busy;busy$, the following abbreviation can be used: $busy[*3]$ [36].

The nonconsecutive repetition operator ([=]) can be applied to Boolean expressions to describe repetitions that should happen on not necessarily consecutive cycles. The non-consecutive repetition operator [=n] will match any sequence of cycles in which there are n not necessarily consecutive repetitions of the Boolean expression being repeated. For example $busy[=3]$ will match any sequence of cycles in which the signal *busy* being repeated 3 times [36].

The Goto repetition operator ([$\rightarrow$]) is similar to the nonconsecutive repetition operator, except that the sequence of cycles being described end with an assertion of the Boolean expression being repeated. It will match any sequence of cycles starting at the current cycle and ending after you go to the nth occurrence of the Boolean expression [36].

Operator class	**Associativity**	**Operators**
Within operator	left	within
Non-length-matching conjunction operator	left	&
Length-matching conjunction operator	left	&&
Disjunction operator	left	$\mid$
Fusion operator	left	:
Concatenation operator	left	;
Implication operators	right	$\mapsto \mid =>$

Table 2.7: Sequence operators

Sequence operators (see Table 2.7) represent the next group of PSL operators. This group includes the so-called *within* operator, which is used to describe a behavior in which one sequence occurs during the course of another, or within a time-bounded interval. Furthermore, this group of classes includes *conjunction* operator, which are used to describe behaviors consisting of parallel paths. These operators are: *non-length-matching sequence conjunction* (&) and *length-matching sequence conjunction* (&&).

The sequence *disjunction* (|) operator, which is used to describe a behavior consisting of alternative paths. The *fusion* (:) operator can be used to describe a behavior in which a later sequence starts in the same cycle in which a previous sequence completes.

The sequence *concatenation* operator can be used to describe a behavior in which one sequence is followed by another one. The sequence *implication* operator can be used to describe behaviors consisting of a property that holds at the end of a given sequence. PSL distinguishes between overlapping suffix implication ($\mapsto$) and non-overlapping suffix implication ($\mid =>$).

Table 2.8 shows further foundation language operators. The table includes the *termination* operator, *occurrence* operator, *bounding* operators and *invariance* operators.The termination operators is used to describe a behavior

Operator class	Associativity	Operators
termination operators	left	abort, sync_abort, async_abort
occurrence operators	right	next*, eventually!
bounding operators	right	until*, before*
invariance operators	right	always, never

Table 2.8: Further foundation language operators

in which a condition causes both current and future obligations to be canceled. Regarding the *termination* operator, it can be either synchronous with respect to a clock event (sync_abort) or independent of the clock event (abort or async_abort).

A behavior in which an operand holds in the future can also be specified by means of the *eventually!* and the *next** operators. The *eventually!* operator states that the right operand holds at some time in the indefinite future, while the operator *next** operator states that the right operand holds at some specified future time or range of future times.

PSL also provide means to describe behaviors in which one property holds in some cycle or in all cycles before another property holds. The *until** operator states that the left operand holds at every time until the right operand holds, whereas the *before** operator states that the left operand holds at some time before the right operand holds.

Furthermore, the foundation language also include the *always* and *never* operators. Both operators can be used to describe behavior in which a property does or does not hold globally. With the *always* operator, the right operand always holds, whereas the *never* operator can be used to specify a behavior in which the right operand does never holds.

Operator class	Associativity	Operators
Implication operators	right	$\rightarrow$
Implication operators	right	$\leftrightarrow$

Table 2.9: Boolean operators

Table 2.9 shows implication operators that can be used to describe a behavior consisting of a boolean, a sequence, or a property that holds if another boolean, sequence, or property holds. Hereby, PSL distinguishes between the logical if implication ($\rightarrow$) and the logical if and only if implication ($\leftrightarrow$).

PSL Sequences and Properties

While the Boolean layer forms the foundation of PSL, the real power of PSL comes from its temporal layer. The term Temporal refers to the design behavior expressed as a series of Boolean expressions over multiple clock cycles. To support this, PSL has two major components in the temporal layer: Sequences and Properties.

Sequences are built from basic Boolean expressions and using sequence operators such as repetition operators.

A PSL Sequence is a sequential expression that may be used directly within a property or directive. A sequence declaration defines a sequence and gives it a name as shown in the following Listing.

```
1 sequence BusArb (boolean br, bg; const n) =
2 { br; (br && !bg)[*0:n]; br && bg };
```

As it can be seen in the listing, the named sequence *BusArb* represents a generic bus arbitration sequence involving formal parameters. *br* (bus request) and *bg* (bus grant), as well as a formal parameter *n* that specifies the maximum delay in receiving the bus grant.

As shown in the following Listing, the named sequence *ReadCycle* represents a generic read operation involving a bus arbitration sequence and Boolean conditions *bb* (bus busy), *ar* (address ready), and *dr* (data ready).

```
1 sequence ReadCycle (sequence BusArb; boolean bb, ar, dr) =
2 { BusArb; {bb[*]} && {ar[->]; dr[->]}; !bb };
```

There is no requirement to use formal parameters in a sequence declaration. A declared sequence may refer directly to signals in the design as well as to formal parameters.

On the other hand, Properties express temporal relationships among Boolean expressions, sequential expressions, and subordinate properties. They are built on top of sequences and can include Boolean expressions, sequences and other sub-ordinate properties. Various operators are defined to express various temporal relationships. A property declaration defines a property and gives it a name as shown in the following example.

```
1 property ResultAfterN
2 (boolean start; property result; const n; boolean stop) =
3 always ((start -> next[n] (result)) @ (posedge clk)
4 async_abort stop);
```

This property could also be declared as follows:

```
1 property ResultAfterN
2 (boolean start, stop; property result; const n) =
```

```
3 always ((start -> next[n] (result)) @ (posedge clk)
4 async_abort stop);
```

Both declaration types have slightly different interfaces (i.e., different formal parameter orders), but they both declare a property called *ResultAfterN*.

Simple PSL Examples

Let's consider a system that accepts requests of some sort and processes them. The assumption is that the system has some definition of time points, which may be points at which a system clock ticks (if the system is synchronous), or points at which certain chosen events occur. PSL only requires that we have a sequence (finite or infinite) of discrete time points. Our system has variables such as *req*, *ack*, *start*, *busy*, and *done*. Each variable is true at certain time points. We demonstrate how each of the following English statements, which describe system behavior, can be formulated in PSL.

- Whenever start is true at a time point, busy will be true at the following time point.

  $$always(start- > nextbusy)$$

- For every occurrence of req that is immediately followed by ack, processing of the acknowledged request begins at the next time point after the ack. The processing sequence begins with start, which is followed by busy for some number of time points, and ends with done.

  $$[*]; req; ack| => start; busy[*]; done$$

PSL is mathematically rigorous, therefore the properties in PSL are precise and unambiguous. However, they are also easy to read. Thus, a specification written in PSL can be used as input for automatic tools and may also serve as part of a human readable specification document.

A more detailed description of the PSL constructs and operators can be found in [52]. The PSL operators are based on Linear-time Temporal Logic (LTL) operators, Computational Tree Logic (CTL) operators. Other PSL constructs are based on SEREs. SEREs are a type of regular expression (see 5.1.2).

An assertion is a conditional statement that checks for specific behavior and displays a message if it does not occur. The assertion definition starts at System Level, when only requirements are defined. At this level only a few information about the final implementation are available, e.g., system interfaces, but the functional details provide the engineers with sufficient information for supporting the definition of temporal assertions.

Chapter 3

Related Work

3.1 IP-XACT

3.1.1 Extensions of the IP-XACT Schema

Many of today's embedded multiprocessor systems are heterogeneous systems, consisting of hardware and software components. The IP-XACT standard was defined to facilitate the automation of the composition and integration of multiprocessor systems. IP-XACT is a well-defined standard format for documenting IPs and is currently supported by many EDA tools.

However, there is no standard that covers all possible aspects in a design domain. For this purpose, the IP-XACT standard schema provides extension mechanisms to capture additional domain specific meta-information. This enables, for instance, the implementation of vendor specific tool features. Such extensions can then be submitted to the IP-XACT working group as candidate for future releases. For example, extensions in the design areas: analog-mixed signal, physical design planning, and power have recently been published by the Accellera Systems Initiatives [51].

To capture meta-information related to Hardware-dependent Software (HdS), IP-XACT extensions have been introduced in [79] and [72]. HdS basically describes the low level software layer that builds on top of the hardware platform in an embedded system. It closely interacts with the hardware infrastructure and provides the application software with an API (Application Programming Interface) that enables access to hardware devices [85].

In [79], extensions are defined to enable HdS integration in executable system models. The main goal of the extension was to enable the representation of driver specific information in order to facilitate the automatic generation of the corresponding simulation model of the system design. In addition to

the IP-XACT schema extensions, the authors defined guidelines for system level design integration and optimization. These guidelines include software, hardware platform and hardware-dependent-software optimization and integration, focusing in one of the fundamental stages of the design flow for complex MultiProcessor System-on-Chips (MPSoCs): the HW/SW simulation. Furthermore, these extensions enable system level simulation of the whole platform, whilst taking into account the influence of the design components that impact the performance, the power consumption, the functionality and the reliability of the system. The guidelines described in this document are used to define a system-level optimization methodology. The validity of the approach in achieving the reduction of power consumption, improvement of performance and development of scalable and reliable systems was demonstrated.

In [72] the authors elaborate on the expressiveness of IP-XACT for describing HdS meta-data to address the automation of HdS generation in the field of reconfigurable computing, where IPs and their HdS are generated on the fly, and therefore, are not fully predefined. Their approach combines IP-XACT-based design description with additional HdS meta-information to automatically integrate different architectural templates used in reconfigurable computing systems. Furthermore, the authors propose several IP-XACT extensions that enable the automatic generation and integration of the HdS. Moreover, they validate these specific extensions and demonstrate the interoperability of the approach based on an H.264 decoder application case study.

3.2 Modeling and simulation of embedded automotive software

3.2.1 Restbus Simulation

The number of ECUs in vehicles is constantly increasing, the software running on the ECUs has also become very complex. These facts make testing a central task within the development of automotive electronics. This task has a high degree of complexity.

In-vehicle-driving-tests are often time-consuming, expensive and often not reproducible, especially when some parts of the system are not available. Restbus Simulation (RBS) is a widespread technique typically applied in later phases of the development process before the complete ECU-network is available.

A typical application for RBS is the validation of a new ECU functionality. The Restbus simulation device simulates the remaining part of the bus net-

work by providing messages from the non-existing nodes to the rest of the network during the simulation. RBS can greatly reduce testing time and cost by allowing companies to simulate new ECU functionality under real-world conditions in order to investigate the response of an ECU without having to set up an entire vehicle network or perform expensive field tests [57, 98].

During simulation, the RBS device and the ECU under test are connected to a network bus. Therefore, the Restbus simulator must fully support the bus communication protocol used. Furthermore, monitoring capabilities of the ECU under test should be provided. Missing protocol-specific control data will cause the ECU to leave the functional state and go into an error state [57].

Up to now, only a few academic publications address the topic of RBS. However, it is a widely used method in the industrial sector. Several companies offer tools, tool chains and hardware equipment to support RBS for various network communication buses, mainly CAN and FlexRay [41]. But they all have one thing in common, which is the fact that a) the RBS simulation device is used to simulate the remaining part of the bus network, b) the system under test runs on a real ECU and c) the PC is only used to host the test automation software.

In contrast to existing RBS frameworks, we present in this thesis an RBS approach where the ECU functionality under test also runs on the simulation PC in addition to the test automation software. Our approach only requires the corresponding bus communication controller to communicate with to the remaining part of the bus network.

3.2.2 Modeling and simulation with SystemC

SystemC (Section 2.4.2) is widely used for the simulation of designs consisting of hardware and software components. In [99], an approach for generating executable SystemC models from software designs captured in a component-based modeling language (Component Language (COLA)) is presented. The approach follows the paradigm of synchronous data flow.

COLA has rigorous semantics and specification mechanisms. Due to its well-founded semantics, it is possible to establish an integrated development process. The resulting COLA models remain abstract and cannot be executed immediately. This allows an early validation and performance analysis of the design during the design process. The models can be formally processed using a model checker or a SystemC code generator.

In this thesis, we make use of IP-XACT for the description of the design components. The design components are provided together with their imple-

mentation. Based on their IP-XACT description a SystemC Top-Level file is generated in which the components are instantiated.

3.2.3 Design Framework for IP Reuse and Integration

In [9], we presented an IP-XACT based framework for automated design integration of mixed-level IPs. The functionality implemented in the framework realizes the key concepts behind IP auto-assembly. Two examples are presented to illustrate the application of the framework, one with the CoreConnectTM SoC [86] architecture and a second example with a self implemented FlexRay bus simulation library.

The investigations were based on the release version 1.4 of IP-XACT. Our modeling framework was created by applying the automatic framework generation process of the Eclipse Modeling Framework (EMF) [88]. EMF enables developers to rapidly construct robust applications based on simple models or meta-models. Thus, our graphical IP-XACT editor was generated with EMF using the XML schema of IP-XACT version 1.4. The functionality directly generated by EMF are the creation, manipulation, and validation of any type of IP-XACT XML files. Since Eclipse is widely used for framework generation, this supports the incorporation of further extensions.

The basic editor was extended by code generation capability. The code generator handles SystemC IPs and their integration, at both RTL and TLM. The interconnections of modules are mainly done at the TLM level, although RTL ports are also dealt with. On hand with such extension efforts and the accompanying experiments, we were able to identify some lacks of IP-XACT for a full automation of design integration, especially at the TLM level, which was rarely addressed at that time.

3.2.4 AUTOSAR Vs. SystemC

As introduced in Section 2.2.2, the AUTOSAR standard specifies a generic architecture and methodology for automotive applications. However, the engineering steps to be taken to move from a logical to a technical architecture or to a concrete implementation, are still not well supported by tools yet.

SystemC offers a comprehensive way to simulate, analyze, and verify software. It also enables the simulation of the timing behavior of underlying hardware and communication channels into account. The similarities between the SystemC and AUTOSAR concepts with respect to architecture modeling are presented in [63]. Additionally, the paper discusses approaches

on how to use SystemC during the design process of AUTOSAR-conform systems.

3.3 Verification of temporal properties

3.3.1 Verifying SystemC using an Intermediate Verification Language and Symbolic Simulation

SystemC has become a widely used standard for the development of embedded systems. The verification of SystemC designs is critical, since it can prevent error propagation down to the hardware. However, formal verification of SystemC models is challenging.

In [64], an approach is proposed consisting of verifying SystemC models using an Intermediate Verification Language (IVL) and symbolic simulation. Before dealing with symbolic inputs and the concurrency semantics, the authors propose an isolated approach by using an IVL. The approach decouples the development of the SystemC-to-IVL translator from the IVL verifier.

Additionally, an extensive benchmark set is presented and an efficient symbolic simulator integrating partial order reduction is proposed. The potential of the approach is validated by means of experimental comparison with other existing approaches.

As opposed to this approach, we propose a timing verification concept that does not require the translation of the SystemC model to an another verification language. In our approach the SystemC model under verification only requires a simple instrumentation consisting of making the Interface being monitored accessible to an external tool that co-simulates the model under test and the verification unit.

3.3.2 Verifying SystemC using a software model checking approach

Although, SystemC allows very efficient simulations, formal verification is still at a preliminary stage. Recent work translate SystemC into the input language of finite-state model checkers, but they abstract away relevant semantic aspects, and show limited scalability. In [22], another approach of formal verification of SystemC is presented. The approach is based on the reduction to software model checking. The authors explore two directions.

First, they rely on a translation from SystemC to a sequential C program, that contains both the mapping of SystemC threads in form of C functions, and the coding of relevant semantic aspects (e.g. the SystemC kernel). With regard to verification, this enables the off-the-shelf use of model checking techniques for sequential software, such as lazy abstraction.

The second approach exploits the intrinsic structure of SystemC. In particular, each SystemC thread is translated into a separate sequential program and explored with lazy abstraction, while the overall verification is orchestrated by the direct execution of the SystemC scheduler. The technique can be seen as generalized lazy abstraction applied to the case of multi-threaded software with exclusive threads and cooperative scheduling.

The two approaches were implemented in a software model checker. An experimental evaluation was carried out on several case studies taken from the SystemC distribution and from the literature to demonstrate the potential of the approach [22].

As opposed to this formal verification approach, we propose a simulation based verification approach.

3.3.3 Monitoring Temporal SystemC Properties

In [91] a temporal monitoring framework for the SystemC specification language is described. The framework uses a very minimal modification of the SystemC kernel, by exposing event notifications and simulation phases. Moreover, the user code is instrumented to allow observation of the relevant parts of the model state.

To validate the approach, the framework is used to specify and check properties of two case studies. The validation activities showed that monitoring SystemC properties using the framework had a reasonable overhead and a marginal cost. Finally, the authors demonstrate that monitoring at different levels of abstraction requires slight changes to the specification and the generated monitors.

3.3.4 Dynamic Assertion-Based Verification

Assertion-Based Verification (ABV) methodologies and tools do not apply to hardware and software components in the same way [33]. Regarding hardware components, both static ABV and dynamic ABV are widely used. Software components, on the other hand, are traditionally verified by means of

static ABV. Furthermore, these assumptions cannot be controlled by the assertion language.

In [33] the exploitation of model-driven design for guaranteeing such simulation assumptions is proposed. The paper describes an ABV framework for embedded software that automatically synthesizes assertion checkers to verify the embedded software according to the simulation assumptions.

3.3.5 Assertion-based Verification of temporal properties

The verification of temporal properties is one of the main challenges in embedded software. Formal property verification using model checking often suffers from the state space explosion problem when a large software design is considered. In [65], two new approaches to integrate assertions in the verification of embedded software using simulation-based verification are introduced.

The first approach consists of extending a SystemC hardware temporal checker with interfaces in order to monitor the embedded software variables and functions that are stored in a microprocessor memory model. Whereas the second approach consists of deriving a SystemC model from the original C program in order to integrate directly with the SystemC temporal checker.

Both approaches are validated using a case study on an embedded software from the automotive industry which is responsible for controlling read and write requests to a non-volatile memory.

Chapter 4

Methodology

This chapter presents our methodology and simulation framework. It is a SystemC-based Restbus simulation framework for early validation of automotive systems. The aim of our approach is to provide a Restbus simulation framework for functional verification of real-time critical automotive systems.

Before exercising the actual Restbus simulation process, we first perform some preliminary timing analysis of the SystemC model, in order to verify that our design under verification conforms to its timing requirements.

The focus during timing analysis lies on the synchronization between the components of the model with respect to data communication. To address these issues, we propose a methodology for assertion-based verification of the timing constraints on the SystemC simulation model and we make use IP-XACT to document the design components and model the platform of the design under verification.

Further, the discussion in this chapter mainly focuses on the specification of the generic architecture of the Restbus simulator, the extension of IP-XACT by timing constraints and the derivation of executable PSL assertions.

The derivation of executable PSL assertions is done according the the TADL2 semantic. TADL2 is a result of the European research project TIMMO2USE [49]. The main goal of the project was to address and propose practical solutions for relevant automotive system design use cases that require special consideration of timing aspects. Further, the language comes with a meta-model describing the attributes of the timing properties and provides the semantic behind the timing constraints.

Therefore, it is a logical step to define the PSL properties in compliance with the provided TADL2 semantic definition.

In fact both specification languages PSL and TADL2 are similar with respect to their expressiveness for formalizing timing constraints. However, in contrast to TADL2, PSL is supported by several SystemC simulation and verification tools, which better fits into our methodology.

4.1 Overall design flow

Our design flow provides modeling guidelines that define how to generate a Restbus simulation model running on a common off-the-shelf PC using a discrete event simulator. SystemC is our simulation language of choice. As discussed in Chapter 2 (Section 2.4.2), SystemC is a system level description language that comes with an event driven simulation kernel. Among other features, SystemC also introduces the notion of time which is crucial for modeling real-time critical systems. Furthermore, we use the IP-XACT standard for the documentation of design components and the platform. PSL is used as formal language for the specification of the timing requirements.

The Restbus simulator can be used in two ways. It can either be applied to simulate the Design Under Test (DUT) or to run a testbench. The usage as testbench is needed when the system under test is a real component. During this design integration phase, timing constraints can be introduced into the model using the provided IP-XACT timing extensions. A detailed discussion on the timing extensions will be made in Chapter 5.

Figure 4.1 illustrates an IP-XACT design example. To differentiate between real and simulated components, we introduce two additional IP-XACT-views namely HIL (for real components) and RBS (for Restbus components). This information is useful during code generation. As depicted in the figure, it is a generic structure that includes both real (Model view: HIL) and simulated components (Model view: RBS). Depending the phase of the design process two different versions of the simulation model are generated. For timing verification, a pure system model of the complete design is generated. This includes SystemC models needed for the simulation of the real components. In the Restbus simulation phase, the generated SystemC model will only include components that implement the functionality under test, since the remaining part of the overall system is physically available.

Furthermore, the example depicts a design composed of five components: four components and an interconnect component. The Interconnect is a generic component of our framework that acts as a middleware. A more detailed description of the Restbus simulator will be given in Section 4.2.

The overall design flow consists of five phases, which are depicted in Figure 4.2. As a starting point, the following inputs are assumed to be available: a) SystemC implementation of all design components, b) the corresponding

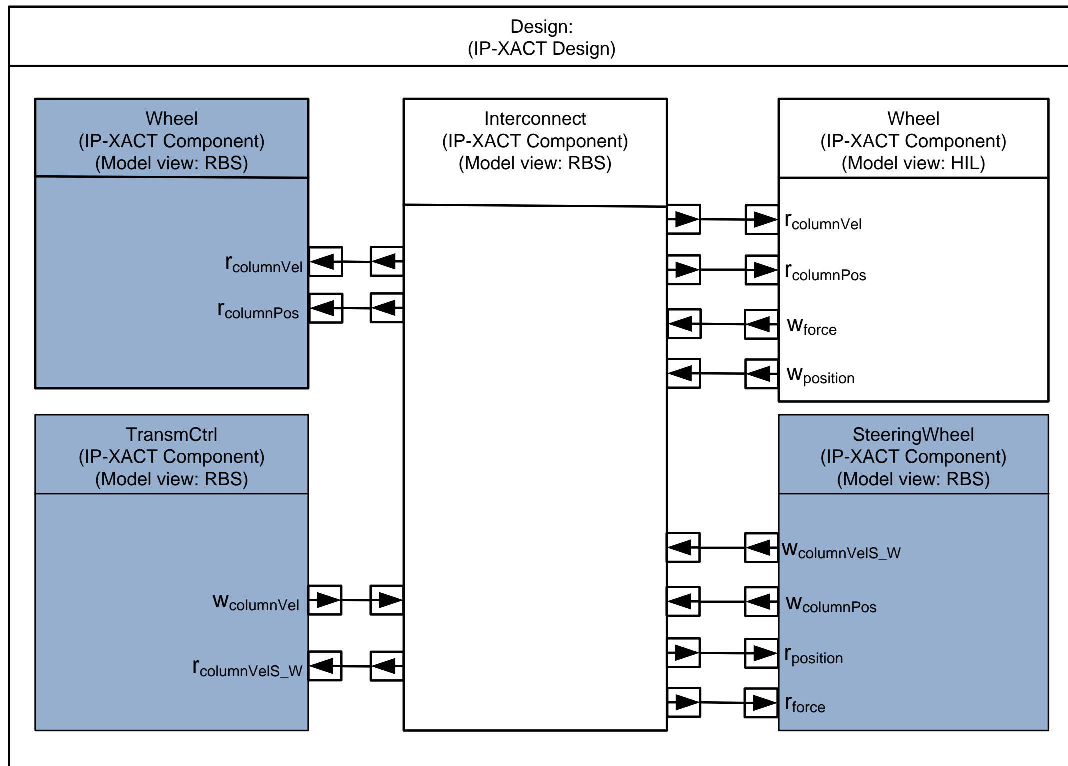


Figure 4.1: Graphical view of an IP-XACT design example

IP-XACT descriptions and finally c) timing requirements on the design under test. In the following, a more detailed description of the phases of the design flow will be given.

4.1.1 Phase 1: Component assembly

The first step consists of integrating all components into a design. As aforementioned, our modeling standard of choice is IP-XACT. The integration is based on the description of the individual components IPs provided in form of IP-XACT components. In the scope of this thesis, IP-XACT components are used to describe modules encapsulating functional models.

4.1.2 Phase 2: Timing requirements formalization and Code generation

In order to verify the timing properties of the design model, requirements need to be translated into a formal specification language. The formalized requirements can then be used to check if the design model is compliant to its specification. ABV is a well-known approach that provides verification

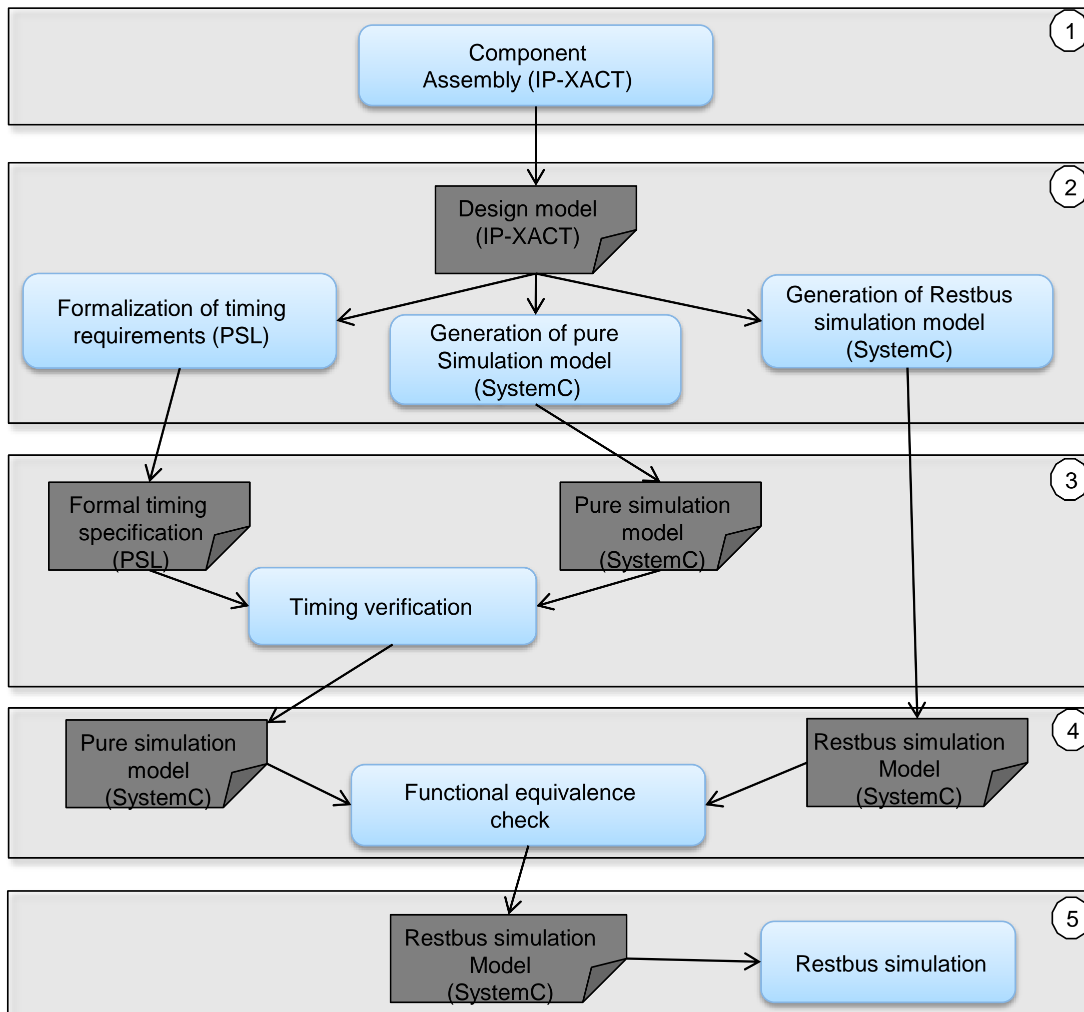


Figure 4.2: Restbus simulation design flow

engineers with means to formally capture the intended requirement specifications [42]. By means of assertions the model under investigation can then be checked in the following phase 3. In this phase of the design process PSL assertions are derived from the timing information contained within the IP-XACT descriptions. The transformation is based on the rules defined in Chapter 5.

Concerning code generation, it is important to first recall the behavior of the SystemC simulation kernel (see Section 2.4.2), which is a typical event-driven simulator. The scheduler of the SystemC simulation kernel is a cooperative non-preemptive scheduler that runs at most one SystemC process at a time. A running SystemC process runs until completion (SC_METHODS) or until it gives control back to the scheduler (SC_THREAD). All runnable processes are executed one at a time in a single delta cycle while postponing channel updates made by those running processes. After all runnable processes have been executed, the scheduler materializes the channel updates and wakes up sleeping processes that are sensitive to the updated channels. If there are runnable processes available, the scheduler moves to the next delta cycle. Otherwise, it accelerates the simulated time to the nearest time point in the future, where sleeping processes or events can be woken up [23].

In a Restbus simulation environment however, the SystemC simulator communicates with physical devices, and as a consequence, real-time here means the time measurements done at the physical environment. Figure 4.3 depicts the possible divergence between the time simulated in SystemC and the time measured on the physical system. As illustrated in Figure 4.3, SystemC can simulate faster or slower than real-time, depending on the complexity of the SystemC simulation model.

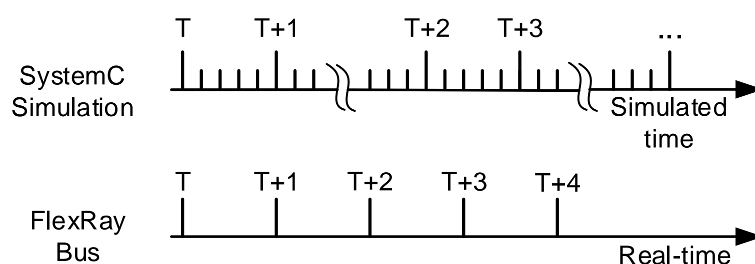
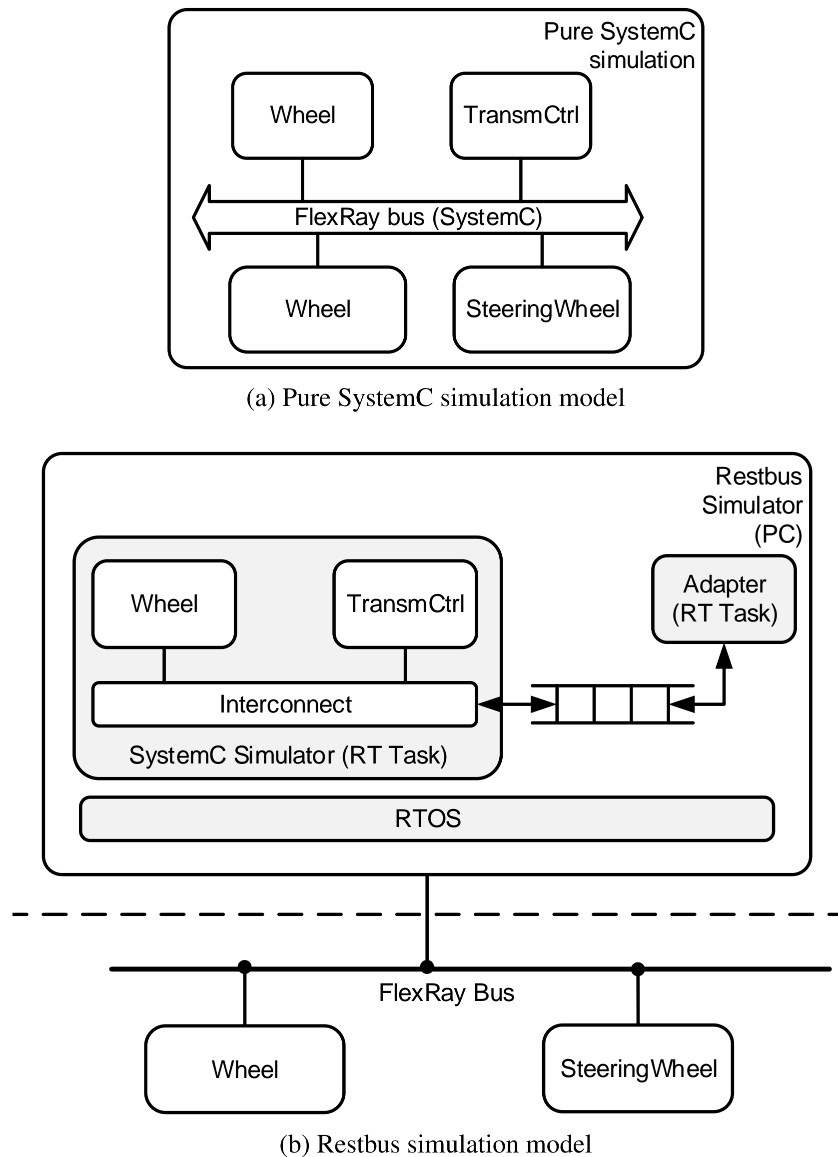


Figure 4.3: Divergence between simulated time and real-time

Since SystemC time and real time might diverge during simulation, it makes more sense to verify the timing properties in a homogeneous environment (with respect to time). Therefore, a pure SystemC simulation model is generated in addition to the Restbus simulation model. An example of the the two model variants can be seen in Figure 4.4. Obviously, the assumption made here is that SystemC models for all design components are available; how-

ever, both model variants should be functionally equivalent. This equivalence check analysis is performed in the next design phase. Moreover, the use of



(a) Pure SystemC simulation model

(b) Restbus simulation model

Figure 4.4: Simulation models used during the design process

a pure SystemC model gives the possibility to investigate and identify possible design flaws with respect to timing; provided the fact that both model variants are functionally equivalent, we can then make sure that the design model complies to the requirements before moving on to the actual Restbus simulation process.

The generic structure of an example of our IP-XACT design can be seen in Figure 4.1. The *Interconnect* is a logical component. For the pure SystemC simulation model, this component is substituted by a bus network model as depicted in Figure 4.4a. Whereas for Restbus simulation, a bus network inter-

face, the so-called *Adapter* will additionally be generated (see Figure 4.4b). The reason for having two different implementations of the generic IP-XACT design is the fact that Restbus simulation includes the communication with real hardware components. Therefore, we then face the challenge depicted in Figure 4.3. To cope with this challenge we introduce an additional component(Adapter) that will handle synchronization and hardware communication.

In [9] we presented an IP-XACT design integration framework for IP-XACT components and designs modeling. The front-end of the framework provides import and export functionality of IP-XACT descriptions. Furthermore, it includes a SystemC code generator. The code generator can be used for the generation of the SystemC top-level component in which the existing components are instantiated and connected. This code generator can be extended by additional features.

As already mentioned, we make use of IP-XACT Views to distinguish between real and virtual components. This information is useful for code generation. The View element in IP-XACT is an attribute that describes an implementation of a component. Components may have multiple views, each associated with its own function in the design flow (see Section 2.4.1). To differentiate between real and simulated ECUs, we introduce two additional views namely HIL (for real components) and RBS (for Restbus components).

4.1.3 Phase 3: Timing verification

As aforementioned, the timing requirements are formalized using PSL [52]. The PSL language is structured into four distinct layers: the Boolean, temporal, verification and modeling layers. The temporal layer of PSL constitutes the major part of the language. This layer includes expressions from the underlying boolean layer, temporal operators and Sequential Extended Regular Expressions or SEREs [34]. PSL is suitable for designs with synchronous timing, where temporal expressions are typically sampled using a clock (see Section 2.4.3 for further details).

Thus, PSL assertions can be used for formal verification by means of model checking (static ABV) or in simulation-based verification (dynamic ABV). Formal verification approaches require a formal model of the design under verification. Static ABV methods exhaustively check the assertions against the formal model, they provide verification engineers with high confidence in system reliability. However, they tend to suffer from the state-space explosion problem that limits their applicability to relatively small/medium size designs [60]. Therefore, dynamic ABV techniques are typically used for large

designs because they are scalable [47]. However, the drawback of dynamic ABV techniques is the fact that they cannot guaranty complete correctness.

In this stage or the design process, we perform dynamic ABV of the design model using the derived PSL assertions. Several verification tools can be used for this purpose [100, 45, 19].

4.1.4 Phase 4: Model equivalence check

Before proceeding to the timing verification stage, it should first be checked whether the aforementioned simulation model variants are functionally equivalent. There are several approaches for equivalence checking of SystemC models. These approaches are all either formal or simulation-based.

Grosse et al. [46], for instance, introduced a simulation based framework for equivalence checking between SystemC models. The models can be described at different levels of abstraction. The framework mainly focuses on TLM designs, and the solution provides mechanisms to easily compare variable accesses by co-simulating the models under investigation using a special client-server architecture.

Cimatty et al. [23] proposed a formal approach for the verification of SystemC models by reduction to software model checking. Their approach exploits the intrinsic structure of SystemC, where each SystemC thread is translated into a separate sequential program and explored by constructing an abstract reachability tree. Additionally, they claim to provide a precise formal model of the SystemC scheduler.

Formal models for the pure SystemC simulation and Restbus simulation models need to be extracted in order to perform a formal functional equivalence checking. This can be done using the formalization technique described in [23]. Having both formal models, the problem can be reduced to a standard equivalence checking problem. Several approached are described in the literature [24].

4.1.5 Phase 5: Restbus simulation

Once the pure SystemC simulation model has been checked and it has been proven that it fulfills its timing requirements, Restbus simulation can be performed using a model similar to the one depicted in Figure 4.4b. As shown in the figure, the Restbus simulator runs on top of a real-time operating system. Both the SystemC simulation process and the bus interface module (Adapter)

are executed as real-time tasks. The architecture of the Restbus simulator will be described in more detail in Section 4.2.

As already mentioned, data synchronization is a crucial issue during the Restbus simulation process. Therefore, Restbus simulation only starts after the pure SystemC simulation model has passed the previous analysis phase. Furthermore, we propose in this thesis a synchronization approach for early validation (see Chapter 7). During the Restbus simulation process, functional verification can be conducted. In this case the design under test can either be the components running on the host machine of the simulator, usually a PC or they can be the real network of ECUs. In the second case the simulator will be used to run the testbench.

4.2 Restbus simulator

4.2.1 Architecture of the Restbus Simulator

The architecture of the Restbus simulator is shown in Figure 4.5. In contrast to existing Restbus simulation infrastructures (see Chapter 3), our approach is based on the use of a standard PC for simulation. One advantage of this approach is the possibility to reuse (test-) components throughout the different levels of abstraction of the DUT. The second advantage is the fact that the bus network interface can easily be adapted to new network buses by simply changing the bus communication controller hardware on the simulation PC.

The framework basically consists of two main parts which are the actual simulator of the SystemC models (SystemC simulator) and an *Adapter* component. This architecture enables a clear separation between hardware specific functionality and the simulation of the DUT. Interaction with the communication controller hardware is made via hardware device drivers and is realized by the Adapter component.

4.2.2 The SystemC simulator

The SystemC simulator provides the runtime environment for the simulation of the DUT. As discussed in Section 2.4.2, SystemC models can be implemented at different levels of granularity/abstraction (see [14]). The levels of abstraction with respect to functionality and communication supported by SystemC can be seen in Figure 4.6.

Theoretically, there is no restriction on the kind of models that can be simulated by the SystemC simulator. Regarding functionality, a differentiation is

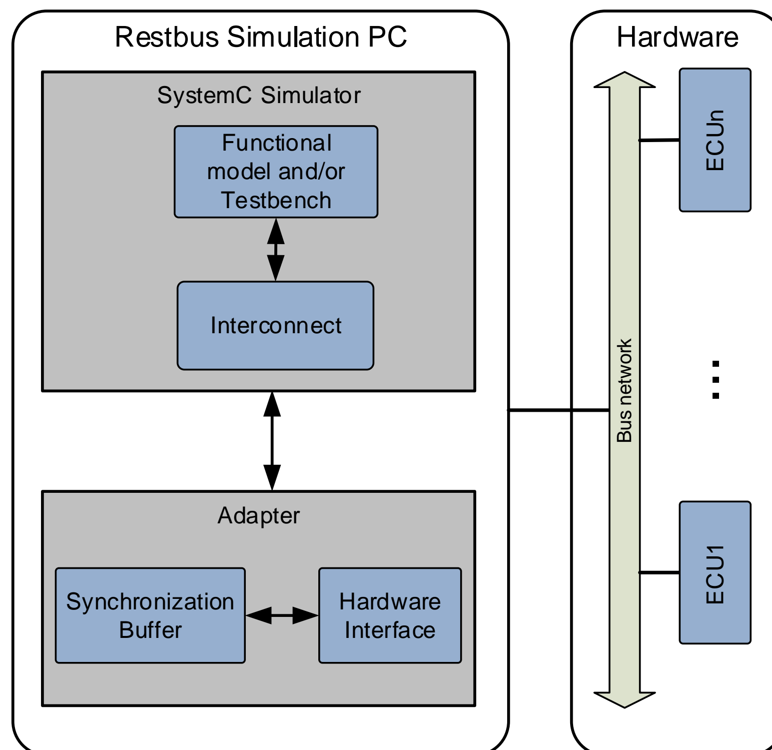


Figure 4.5: System Architecture of Restbus simulation framework

made between four accuracy levels, from RTL as the most accurate models to Un-Timed (UT) models as the most abstract level via Approximately Timed (AT) and Loosely Timed (LT) moving upwards. Concerning communication, models are also categorized into four groups spanning from UT via LT followed by AT and PCA (Pin and Cycle Accurate) as model accuracy increases. Moreover, there are two basic modeling abstraction techniques namely TLM and RTL. TLM can be refined into further categories as discussed in [21].

The choice of the design abstraction level depends on the current verification or test objectives. It is always a tradeoff between simulation speed and model accuracy. In early stages of the design process, typically TLM models are used, since they can be simulated much faster than RTL models [39]. As shown in Figure 4.6, the TLM modeling technique can be used at almost all levels of abstraction with respect to functionality and communication. Furthermore, TLM models can be used for architectural modeling, algorithmic modeling, virtual software development platform, functional verification or hardware refinement.

Additionally, System Architectural Model (SAM) and Bus Functional Model (BFM) models can also be used. SAMs are typically written in C, java or a similar language, and serve as communication vehicle between algorithm, hardware, and software groups. BFMs can be used to encapsulate the bus functionality of a processor for instance during architectural specification.

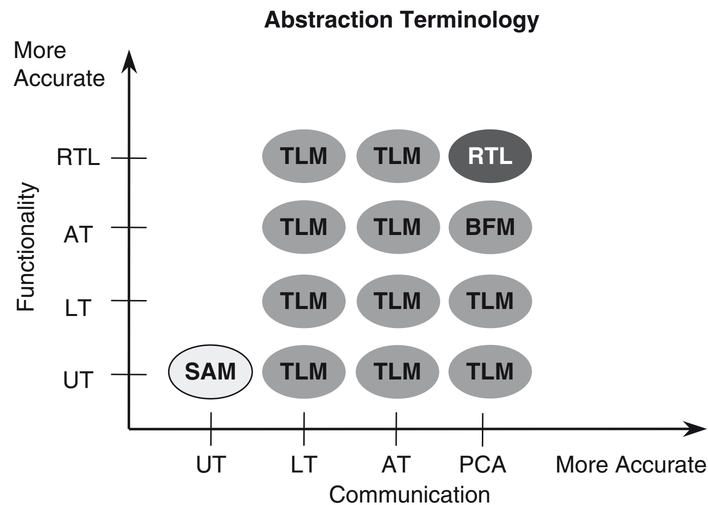


Figure 4.6: Abstraction refinement and TLM mapping source: [14]

The implementation of the DUT can either be hand-written or automatically generated using a Behavioral Modeling Tool (BMT) like Matlab/Simulink or Targetlink. Further, the component *Interconnect* acts as a middleware during Restbus simulation, by basically routing communication data from or to the components of the DUT.

4.2.3 Adapter

The Adapter is the hardware dependent part of the Restbus simulator, it acts as a middleware between the SystemC model and the bus network. In addition to hardware communication, this component synchronizes the SystemC simulator with the bus network. For this, it implements a synchronization mechanism that deals with potential oversampling and undersampling issues that might arise during bus communication. The synchronization approach will be described in more details in Chapter 7. As aforementioned, the separation between the simulation of the functional model and the execution of hardware dependent functionally brings more flexibility to the framework. Thus, it is much easier to adapt to new network communication buses. In contrast to existing Restbus simulation devices that have a predefined set of supported bus network interfaces, only the corresponding communication controller device together with its device driver would need to be plugged into and installed on the simulation PC.

Figure 4.7 displays the behavior of the main process of the Adapter component. As depicted in the figure, the process starts with the initialization and configuration of the communication controller. This is done via the device

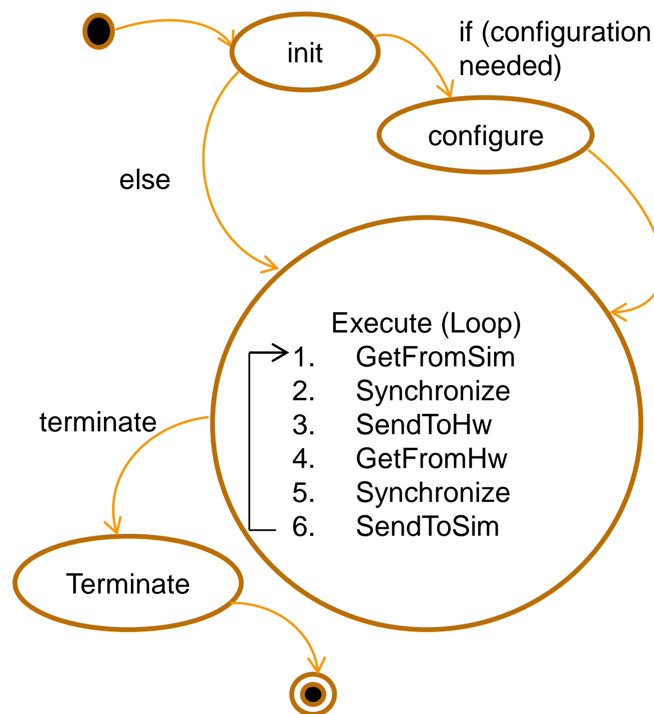


Figure 4.7: Generic behavior of the adapter component

driver that comes along with the communication controller device. Moreover, the routines to be executed are also specified in the Application Programming Interface (API) documentation of the device driver.

The input file format needed for the configuration of the Communication Controller (CC) depends on the type of network bus being used. Typical automotive buses are FlexRay, CAN, LIN, MOST and meanwhile Ethernet is getting more and more application. The bus configuration parameters are provided by means of a database file. Current standard formats are Field Bus EXchange Format (FIBEX) [12], CANdb and AUTOSAR also provides its own database configuration file format. The CANdb is the quasi-standard for the description of CAN communication data and is supported by Vector Informatik GmbH.

With respect to configuration, there are two different possibilities. The first one occurs when building up the whole test environment and the second and most challenging one, regarding network configuration is the integration of virtual network nodes into an already existing infrastructure. In the second case, the starting point is the existing network configuration.

The network configuration is influenced by the communication scenario. We distinguish between monitoring and control applications. For monitoring applications only the Restbus simulation node would need to be configured. However, for control applications the related physical nodes would also need

to be reconfigured since they have to be aware of the nodes simulated by the Restbus simulator. The configuration is done by a bus configuration tool (e.g.: [38]).

Chapter 5

Assertion-Based Timing Verification

Mastering the design challenges of today's automotive systems has become extremely complex. The verification of timing properties is of great importance. In today's cars, functions are distributed over several ECUs. For instance, an Adaptive Cruise Control (ACC) system requires at least 5 ECUs to control and operate engine, gearbox, braking, MMI-Interface and the radar system [87].

In this chapter, we will introduce timing extensions for IP-XACT. By means of these extensions, timing constraints on the design can be captured in a formal way to specify temporal correctness. This can be useful for instance for the specification of end-to-end delays during component integration. A design is said to be temporally feasible, if it meets all specified timing constraints. The constraints themselves are identified during the design process.

In conjunction with the introduction of timing extensions, we will provide transformation pattern from each timing constraint into executable PSL formulas[1]. The timing extensions were inspired by TADL2 [80]. TADL2 is an outcome of the ITEA2 project TIMMO2USE [76].

Our transformation pattern were defined according to the semantics specified by TADL2 [80]. Thus, based on these transformation patterns, timing requirements on a Design Under Verification (DUV) can be expressed in PSL, which is supported by several verification tools. As described in Chapter 4, the DUV considered in this thesis is a pure SystemC simulation model.

The main contributions of this chapter are the definition of timing constraints for IP-XACT and the corresponding executable PSL assertions for dynamic ABV.

[1]PSL is a formal language developed by the Accellera Systems Initiative [51] that can be used to specify properties or assertions for designs (see Section 2.4.3).

5.1 Background

5.1.1 Motivation

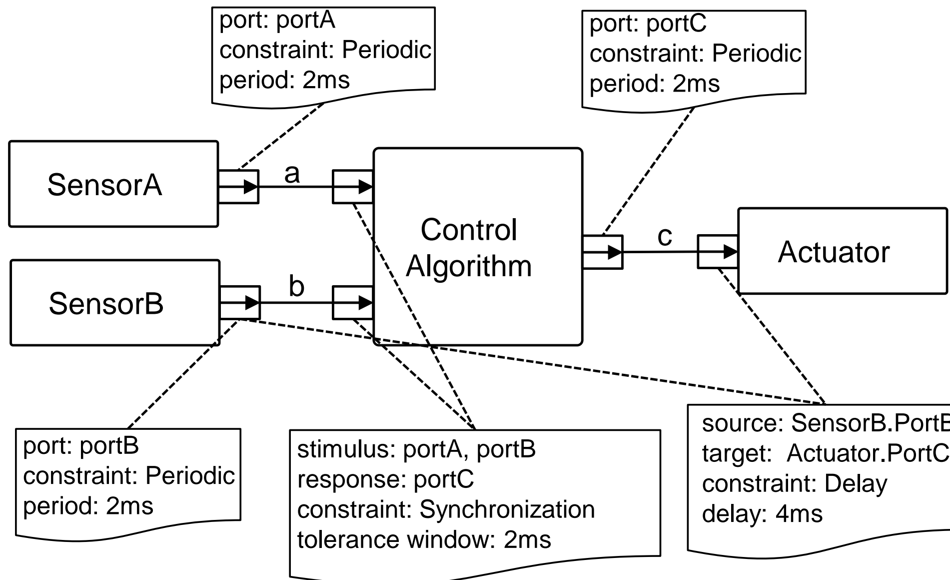


Figure 5.1: Simple example illustrating the need for the specification of timing constraints

Figure 5.1 depicts a basic example that motivates the need to augment the design description with timing requirements, in order to facilitate the design verification process. This rather simple example depicts the logical view of a control system, where a control algorithm receives sensor data (a and b) from two distinct sensor components (SensorA and SensorB). The sensor data are then further processed by the algorithm in order to compute the output data (c) needed by an actuator component (Actuator).

In a real-time critical system the reaction time of such a control system is crucial. Therefore, the design has to be thoroughly investigated, which means tested and verified. As discussed in Chapter 2, one approach is to attach timing requirements to observable points in the design, using our introduced timing constraints.

Let's consider a situation where the sensor values a and b are provided with a transmission period of $2ms$. Further, let the data acquisition period of the control algorithm also be $2ms$ and the data acquisition period of the Actuator component be $4ms$. Then, a maximum duration of $4ms$ from the transmission of sensor data a and b to the reception of the value c by the actuator component can for instance be derived. Additionally, a periodic timing constraint of $2ms$ can also be specified for each sensor component to later verify if it is really the case during the simulation process.

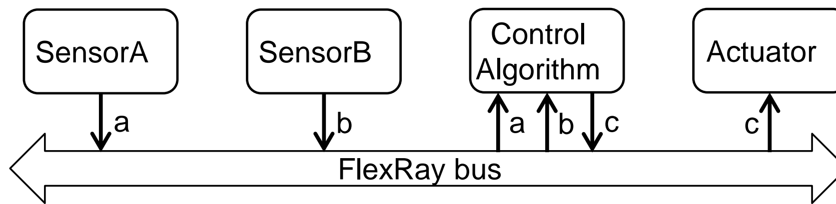


Figure 5.2: Resulting network topology after deployment

Figure 5.2 depicts the network topology of the control system resulting from the deployment of the system's components over a network of four ECUs. The configured FlexRay communication schedule can be seen in Figure 5.3. As discussed in Chapter 2, FlexRay communication is based on the TDMA and FTDMA protocols. The schedule depicted in this figure only uses the static segment of the communication cycle. The static segment is subdivided into four static slots, whereas the sensor components are assigned the first and second slots for data transmission. The control algorithm uses the fourth slot to transmit the computed actuator value c. Further, the components Control algorithm and Actuator are also assigned the respective reception slots.

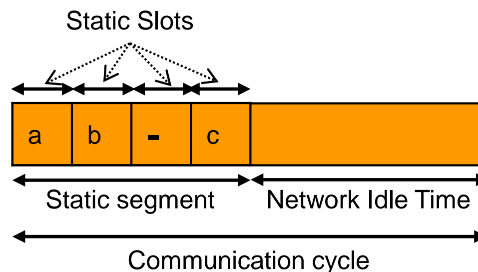


Figure 5.3: FlexRay communication schedule configuration

Obviously, due to the introduction of the bus network into the design, further timing requirements can be derived. Figure 5.4 depicts an analysis window showing event occurrences related to data transmission and reception of the ECUs in the network. In the figure, event occurrences are depicted horizontally for each component using small stripes. Furthermore, t_{xi} denotes the i-th event occurrence of transmission of data x, while r_{xi} denotes the i-th event occurrence of the reception of data x. The age of the data being processed by the controller ECU is identified in the respective cycle, thus the notation x_i is used.

Moreover, a simplified view of the bus communication schedule can be seen at the bottom of the figure.

Figure 5.4 basically outlines further timing constraints that can be derived. The verification process could also include a check if sensor data are provided on time to the ECU hosting the control algorithm. This can be done by defining a tolerance window that specifies the time frame within which the

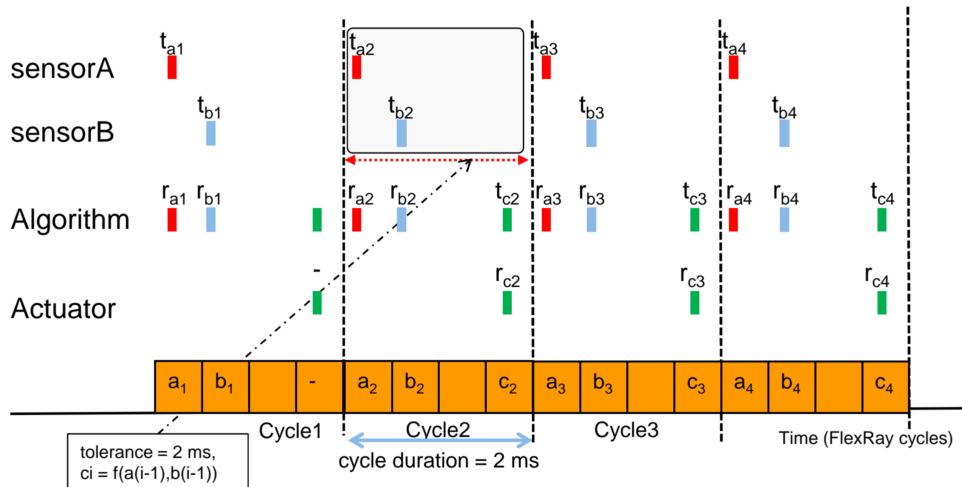


Figure 5.4: Analysis window of the distributed system events

data needed by the control algorithm should be provided (e.g.: $c_3 = f(a_2, b_2)$ at communication cycle 3).

5.1.2 PSL, Sequential Extended Regular Expression (SERE)

As discussed in Section 2.4.3, PSL can be used as a formal language to specify temporal properties of synchronous systems using the temporal layer. PSL includes a special type of regular expressions called Sequential Extended Regular Expression (SERE). SEREs provide means to describe the relation between Boolean layer expressions over time.

The simplest type of SERE is a sequence of Boolean expressions separated by semicolons. For instance, the SERE $\{req; !ack; ack\}$ describes a scenario occurring over three timing points (or cycles), in which a req signal holds at the first timing point, the acknowledgment signal (ack) does not hold at the second, but holds at the third timing point.

SEREs may also be used to describe more complex event sequences. For example, the operator [*] indicates an interval from zero to more timing points, at which any event may occur. Thus, the SERE $\{start; [*]; done\}$ describes any scenario that begins with a start signal and ends with a done signal. The $[*]$ operator may also be attached to a Boolean expression. The expression $busy[*]$ describes for instance an interval from zero to more timing points at which the signal busy is true.

Further operators serve as shorthands for more complex constructions. For any natural number n, the expression $busy[*n]$ describes a sequence of exactly

n timing points. For example, $busy[*4]$ is equivalent to $\{busy;busy;busy;busy\}$. SEREs may be used as building blocks of PSL properties.

Typically, a property may be composed of SEREs using the temporal suffix implication operator $|=>$. For instance:

$$\{[*];req;ack\}|=>\{start;busy[*];done\}.$$

This property specifies that any occurrence of the left-hand side scenario must be followed by an occurrence of the right-hand side scenario. In this particular case, $\{[*];req;ack\}$ describes a sequence where *req* is immediately followed by *ack*, which may occur at any point in time (due to the $[*]$ at the beginning of the SERE). Additionally, the property specifies that such a sequence must immediately be followed (i.e. starting at the next timing point) by a sequence matching $\{start;busy[*];done\}$.

Moreover, PSL offers temporal operators such as *always* and *eventually*, which specify when a Boolean expression must always hold. The combination of these temporal operators with SEREs thus enables PSL to express and specify timing related properties. In order to specify appropriate timing constraints, it is necessary to provide a timing specification, that clearly defines a time base for the simulation clock.

As a summary, PSL is a language that can be used to formulate standard temporal logics formulas LTL and CTL. PSL formula can be compiled down to a formula of pure LTL (respectively, CTL).

Specified PSL formulas can then be used to automatically generate checks of simulations. This can be done, for example, by directly integrating the checks in the simulation tool; by interpreting PSL properties in a testbench automation tool that drives the simulator; by generating HDL monitors that are simulated alongside the design, or by analyzing the traces produced at the end of the simulation [52].

5.1.3 IP-XACT

As discussed in Section 2.4.1, IP-XACT is an IEEE standard that provides standardized structures for packaging, integrating, and reusing IPs within EDA tool chains. IP-XACT is a well established standard in the EDA industry. The intent of IP-XACT is not to provide means for behavioral modeling, but rather to provide the description of component structures and interfaces.

In the context of this thesis we make use of the powerful capabilities of IP-XACT to model the hardware baseline of the DUV. Figure 5.5 illustrates the basic structure of an IP-XACT design as specified by its schema.

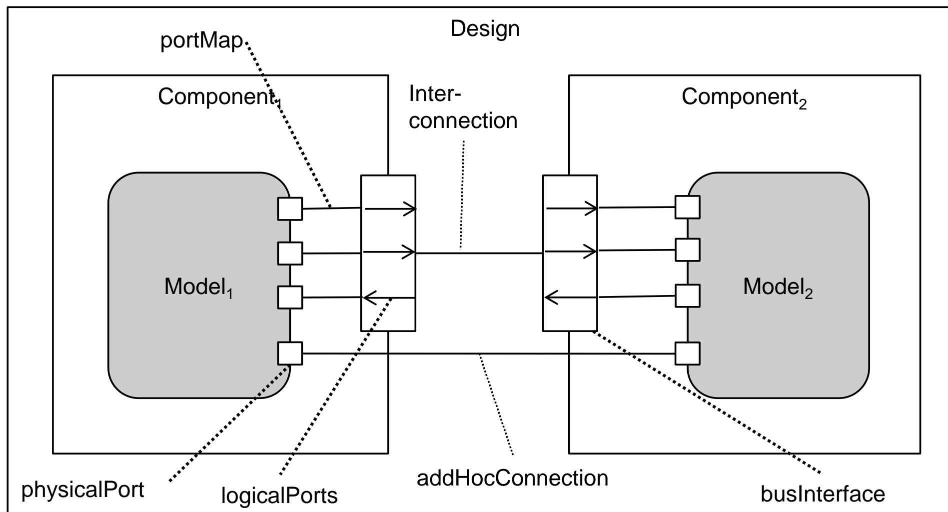


Figure 5.5: A simplified design structure of IP-XACT

IP-XACT supports design modeling at different abstraction levels, mainly RTL and TLM. Furthermore, IP-XACT distinguishes between wired and transactional ports depending of the abstraction level of the model (see Chapter 2, Section 2.4). The current standard (IEEE 1685-2009 [48]) provides means to specify delay constraints on *wired ports*, which are intended for use in RTL models (see [48], page 38). However, besides the fact that only one type of timing constraint is supported, there are still, to our knowledge, no means to specify timing constraints for transactional level models.

As depicted in Figure 5.5, the BusInterface element allows for grouping together ports that collaborate to a single protocol. Components communicate with each other through their bus interfaces. Bus interfaces map the physical ports of the component to the logical ports of the abstraction definition.

The IP-XACT port on a component (logicalPort), can be specified as either of type wire or transactional. A wire type port can be used to specifiy RTL information, whereas the transactional type of port is the type of port that can be used to specify the TLM and SystemC information for ports that have user defined interfaces.

The IP-XACT standard provides means to integrate user defined information by means of the so-called *Vendor-extensions*[2]. Our proposed timing extensions can be applied on both types of IP-XACT ports, this means not only on wired ports but also on transactional ports. As aforementioned, the theoretical background for the proposed timing extensions can be found in TADL2 [80]. TADL2 is an outcome of the ITEA2 project TIMMO2USE [82].

[2]The IP-XACT extension mechanism supports the definition of vendor specific features to implement specific tool or flow features, such as the storage of vendor-specific IP meta-data. [51]

The language defines not only a meta-model but also provides a semantic definition for the specified timing constraints.

The diagrams used throughout this chapter for the introduction of the timing extensions give a graphical view on the organization of their elements and attributes. They were all generated using the Extended Markup Language (XML) Schema Editor XMLSpy [11]. A description to the semantic behind the pictorial representation of the tool XMLSpy ist presented in Appendix B.

5.1.4 DataEvents and Event chains

Before going into the details of the IP-XACT timing extensions, let's first introduce the XML schema elements *DataEvent* and *EventChain*, that are two basic elements that build the foundation our timing constraint schema extension.

As mentioned in the previous section, the structure of the timing constraints is depicted using a pictorial representation of the new XML Schema elements (see Appendix B). This kind of graphical representation shows a tree-like view of the structure of the XML schema elements. The hierarchy of the structure can be read from left to right. Each *subElement* is represented by a branch and can be mandatory or optional. A branch element may contain further elements inside and is indicated by the small plus sign (+) in the small box on the right.

Moreover, we focused on the subset of the TADL2 timing constraints (Section 2.2.3) related to data transmission (send/write) and reception events (receive/read). Generally speaking, a *timing event* denotes any form of identifiable state change of a system at run-time, that can be constrained with respect to timing. These changes can occur at distinct points in time; referred to as occurrences of the event.

In this thesis, we focus on the timing constraint subset related to data transmission. We refer to data transaction primitives as *DataEvent*. As depicted in Figure 5.6, a *DataEvent* basically points to the Port of a Component through which the data is sent or received. This is realized via the schema element "PortRef".

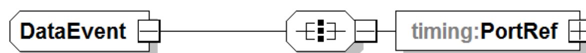


Figure 5.6: DataEvent contains a reference to the schema element PortRef

Therefore, simulation traces can be captured as specified in the following Formula 5.1.

$$DataEvent e = e_i, \ldots, e_n, \ldots \tag{5.1}$$

where e_i is a timing point, that denotes the i-th occurrence of the *DataEvent* e. Event occurrences are expressed using timestamps.

Furthermore, causally related *DataEvents* can be expressed by means of the so-called *EventChain* element (highlighted by the yellow boxes in Figure 5.7).

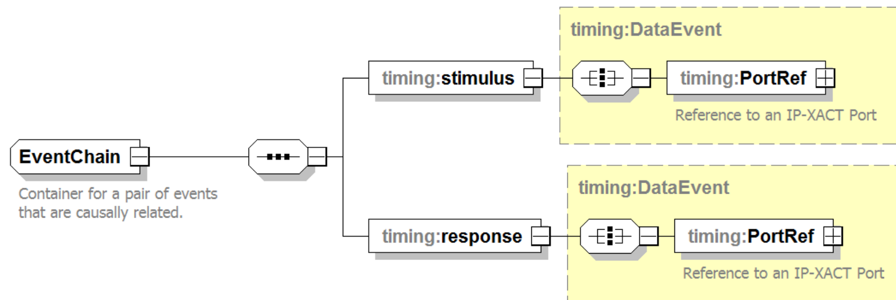


Figure 5.7: Event chain

5.1.5 TADL2: Notation

The timing augmented description language TADL2 uses First-Order logic formulas to express the logic equivalence of the timing constraints. The history of First-Order Logic (FOL) has been described by Ferreiros in [40]. FOL is widely used in areas such as mathematics, and computer science. It is also referred to in the literature as First-Order Predicate Calculus, Lower Predicate Calculus, or Predicate Logic [95].

First-Order logic assumes that the world contains objects, relations and functions. The basic elements are Symbols, Atomic sentences and Terms. The alphabet consists of the following symbols:

- Constants: e.g.: d, c, c_1
- Predicates: e.g.: P, Q, $ItIsRaining$
- Functions: e.g.: f, g
- Variables: e.g.: x, y, a, b, ...
- Connectives: e.g.: $\wedge$, $\Rightarrow$, $\forall$, $\exists$, ...
- Auxiliary symbols: e.g.: (,) ;

Atomic sentences can be built as follows: $Predicate(term_1, ..., term_n)$ or $term_1 = term_2$. A term can be a function ($function(term_1, ..., term_n)$), a constant or a variable [95].

TADL2 Notation	Description
$a \wedge b$	constraint a and constraint b are both true
$a \Rightarrow b$	a is false or a and b are both true
$a \Leftrightarrow b$	a and b have the same truth value
$\forall x : c$	c is true for all possible values of x
$\exists x : c$	c is true for at least one value of x
$\|X\|$	number of elements in set X
$X \subseteq Y$	all elements in X are also in Y
$\forall x \in Y : c$	for each x in Y, c is true
$\exists x \in Y : c$	there is an x in Y such that c is true
$X \leq Y$	X is a subsequence of Y
$x \leq y$	x occurred earlier than y
$x = Y(i)$	x is the element number i in Y counting from zero
$[X]$	set of all occurrences between smallest and greatest occurrences in X
$\lambda([X])$	the length of the continuous intervals in X
$\lambda(X)$	the total length of all continuous intervals in X

Table 5.1: Notation used in the definition of the TADL2 semantics

Table 5.1 gives a list of the operators used by TADL2. Syntactic and semantic objects like events, constraints and time, are referenced by simple variable names such as ”c” for constraint, ”x” for an event occurrence, or ”Y” for a set or sequence of event occurrences. To denote attributes of such objects TADL2 makes use of an object-oriented notation, where for instance c.jitter denotes the attribute jitter of constraint c. Variable x thus stands for a particular occurrence, which in all arithmetic contexts simply means its timestamp value [80]. Since event occurrences are ordered according to their timestamps, a set of occurrences can be seen as a sequence. Thus, sub-sequence relations between such sets, as well as the notion of indexing are also used.

5.2 Formalizing Timing Requirements

As discussed in Chapter 4, the first step of the design process consists of assembling all components needed for the design. The architecture design is based on the IP-XACT component description. One of the activities of the second stage of the design process consists of formalizing timing requirements, that will guide through the timing verification process. Timing requirements can typically be derived from the design requirement specification.

To facilitate the automation of the design verification process, it is more convenient to have all required timing information included in the design model.

The current IP-XACT standard (IEEE 1685) [48] does not provide means to define timing constrains on IP-XACT components. Therefore, we propose the introduction of timing extensions for the standard consisting of a set of timing properties.

5.2.1 Reason for using both TADL2 and PSL

As already mentioned, the proposed timing properties represent a subset of the Timing Augmented Description Language (TADL2). TADL2 is a result of the European research project TIMMO2USE [49]. The main goal of the project was to address and propose practical solutions for a group of automotive system design use-cases that require special consideration of timing aspects. Therefore, it is a logical step to use TADL2 as a starting point for our automotive design and verification process and to consider not only the timing properties defined in TADL2, but also the semantic behind the timing constraint modeling elements.

One of the core contribution of this thesis is the development of a concept for automated timing verification of SystemC design models. The reason for using PSL is the fact that, there is to our knowledge no existing tool support for the verification of TADL2 timing constraint during SystemC simulation. Thus, we opted in the context of this thesis for PSL.

Moreover, our verification approach is realized using a commercial verification tool that supports PSL as formal property specification language. As a consequence, the timing properties contained within the IP-XACT model are formalized using PSL. However, the transformation rules from the timing properties to PSL specifications are compliant to the semantic defined by TADL2 . Furthermore, this transformation is more or less straightforward, since both languages are based on the same formalism.

Since our timing constraints are related to data communication, the extensions are attached to the *Port-description* of the IP-XACT component element. Figure 5.8 gives an overview of the type of timing constraints supported by our extension. As depicted in the figure, a timing should constrain one of the following: Delay, Order, Periodic, Sporadic, Input- or Output synchronization constraint. This is indicated by the symbol being drawn with a dashed line and the choice box symbolB.1.2. The element *typeOfConstraint* is a mandatory element, that specifies the kind of timing constraint that has been defined.

As part of the contribution of this thesis, we define transformation pattern from the timing properties contained in the IP-XACT model to executable PSL assertions. These transformation rules could then be used as a basis for the implementation of a pattern based PSL code generator.

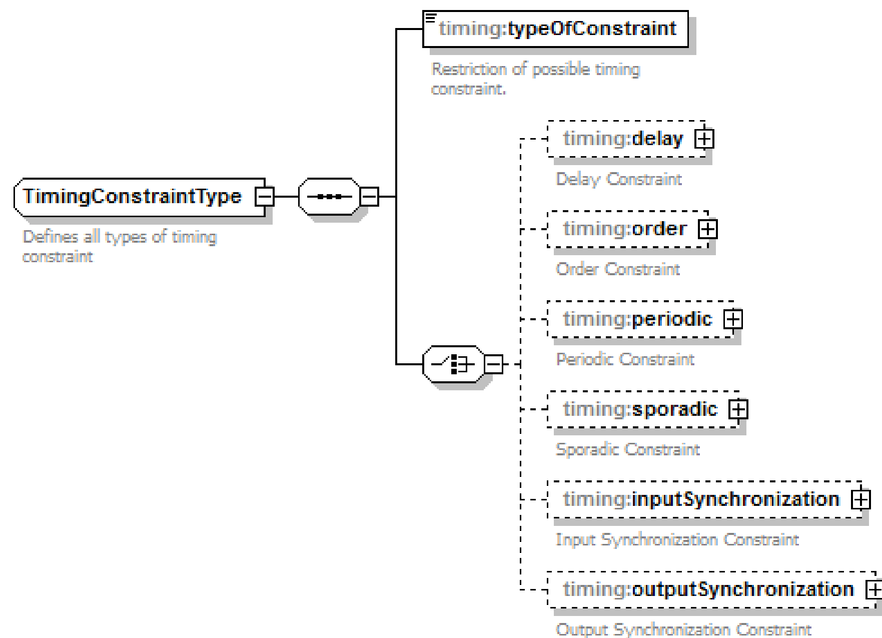


Figure 5.8: Proposed timing extensions for IP-XACT

TADL2 distinguishes between basic and derived timing constraints together with corresponding semantic definition (see [80]). Derived timing constraints are expressed by means of the basic ones. The chosen subset of basic timing constraints used in the scope of this thesis are: *RepeatConstraint*, *DelayConstraint* and *RepetitionConstraint*.

As introduced in Section 2.4.3, PSL has two major components in the temporal layer: Sequences and Properties. Sequences are built from basic Boolean expressions and using sequence operators such as repetition operators.

A PSL Sequence is a sequential expression that may be used directly within a property or a directive. A sequence declaration defines a sequence and gives it a name. A declared sequence may refer directly to signals in the design as well as to formal parameters.

Properties, on the other hand, can be used to express various temporal relationships among Boolean expressions, sequential expressions, and subordinate properties. A property declaration defines a property and gives it a name. Analogously to PSL Sequences, PSL Properties may also refer to formal parameter. The advantage of using formal parameter is the fact that it promotes reuse.

An instantiation of a PSL declaration creates an instance of the named declaration and provides an actual parameter for each formal parameter. In the instance created by the instantiation, each actual parameter expression in the actual parameter list of the instantiation replaces all references to the formal

parameter in the corresponding position of the formal parameter list of the named declaration [52].

In the remaining part of this chapter, each timing constraint will first be introduced using a first order logic formula as specified by the TADL2 semantic definition, followed by our proposed PSL formula.

Furthermore, the named sequences and properties are generic and involve all formal parameters needed to express the specific timing constraints. In the scope of this thesis we focus only on the definition of timing constraints for $span = 1$. Consequently, this simplifies their formalization in PSL.

To be able to specify the timing constraints in PSL, we make use of logical signals to notify the occurrence of the data related events (*dataEvent*) during the SystemC simulation.

5.2.2 RepeatConstraint

The *RepeatConstraint* is a basic timing constraint that describes a repeated distribution of occurrences of a single event. It is characterized by four attributes, namely, a reference to the component port through which the event under observation occurs, lower- and upper-bounds and a span attribute. The attributes lower and upper are timing points, that specify the range of accepted durations between the occurrences of the referenced *DataEvent*.

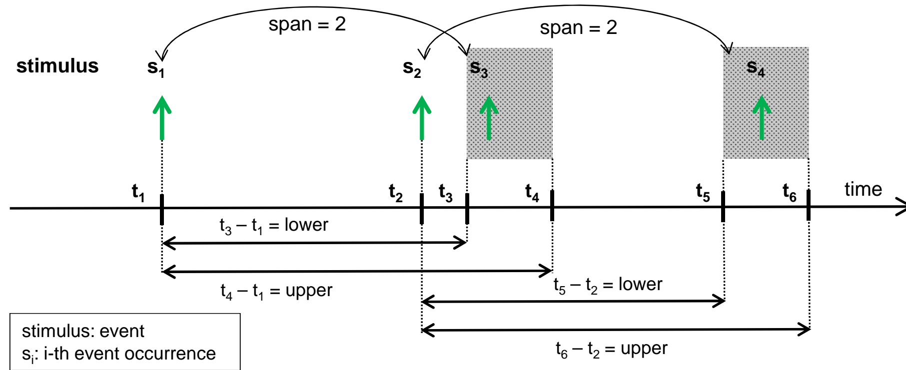


Figure 5.9: A set of event occurrences satisfying a RepeatConstraint for span=2.

As illustrated in Figure 5.9, data event occurrences are depicted by upward oriented arrows, and the span attribute is used to define the index of the correlated event occurrences. Furthermore, the figure depicts a set of event occurrences satisfying a RepeatConstraint with $span = 2$. The corresponding TADL2 expression can be specified using Formula 5.2,

$$\forall X = \{s_i, ..., s_{i+2}\} \leq S : |X| = 3 \Rightarrow lower \leq s_{i+2} \leq upper \qquad (5.2)$$

where s_i denotes the i-th event occurrence of the stimulus event. For $span = 2$ the RepeatConstraint constrains the event occurrences s_i and s_{i+2}.

The usage of the span attribute can be important for the design of a FlexRay bus network for instance, where a specific transmission slot can be used by a network node for the transmission of different kind of data. This notion is usually referred to as Cycle Multiplexing. Cycle Multiplexing is a concept provided by the FlexRay communication protocol to increase the number of different messages that can be transmitted within the same dedicated transmission slot (see Section 2.3.2).

As specified in [80], the *RepeatConstraint* is satisfied if and only if, for each sub-sequence of event occurrences X containing $span+1$ event occurrences, the distance between the first and last event occurrences in X lies between the lower and upper bounds. This constraint is generalized in TADL2 using Formula 5.3.

$$\forall X \leq dataEvent : |X| = span + 1 \Rightarrow lower \leq \lambda([X]) \leq upper \tag{5.3}$$

In the formula, $\lambda([X])$ (see Table5.1) basically denotes the timing point of the last event occurrences in the sub-sequence X. *dataEvent* denotes the complete set of observed event occurrences.

As previously mentioned, we focus in this chapter on the timing constraint specification with $span = 1$, where the span attribute is relevant. Therefore, $\lambda([X])$ will always denote the next occurrence of given *dataEvent*. Thus, we propose the following PSL declaration, to express the RepeatConstraint:

```
1 sequence repeatSequence (boolean dataEvent; const lower, upper) =
     {[*lower:upper]; rose(dataEvent)};
2 property repeatConstraint(boolean timingEvent; const lower,
     upper, span) = always(repeatSequence (timingEvent, lower,
     upper))@(posedge clock);
```

The PSL sequence declaration above states that the *dataEvent* should repeatedly occur within a time window specified by the timing points lower and upper. Further, the property declaration states that the sequence should always hold in every evaluation cycle of the PSL assertion, which is compliant to the TADL2 semantic specified by Formula 5.2. Furthermore, *(posedge clock)* denotes the default PSL clock.

5.2.3 StrongDelayConstraint

The *StrongDelayConstraint* is used to specify that a response has to happen within some time interval after its respective stimulus has been raised.

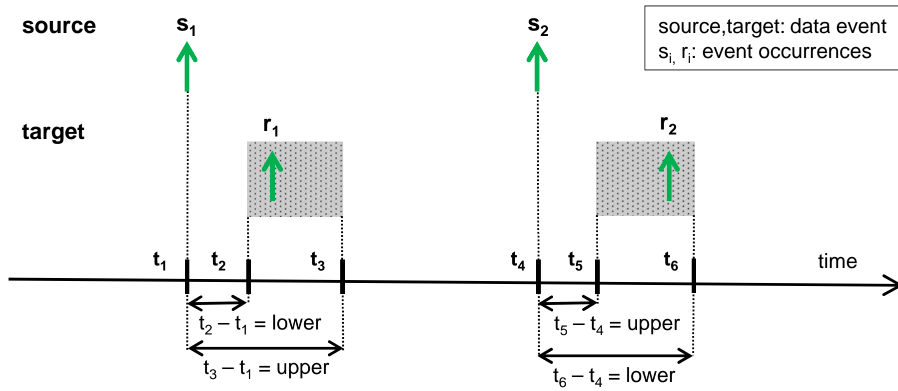


Figure 5.10: A set of event occurrences satisfying a StrongDelayConstraint [80]

Figure 5.10 depicts a system behavior that satisfies a *StrongDelayConstraint*. The picture shows two communicating nodes (source and target); the stripes on the time line indicate the timing points of the communication events of the nodes.

A *StrongDelayConstraint* can be used to constrain the occurrence of stimulus event (generated by the source node) and its corresponding response occurrence (generated by the target node) to occur in lock-steps. Stray response occurrences are not allowed, this means, only one response occurrence for each stimulus occurrence is allowed.

The StrongDelayConstraint is characterized by four attributes, which are: the references to the stimulus, response ports and lower- and upper-bounds to constrain the occurrences of the response. Formula 5.4 gives the equivalent logic formula as specified in TADL2.

$$\forall i : \forall x = stimulus(i) \Rightarrow \exists y : y = response(i) \wedge lower \leq y - x \leq upper \quad (5.4)$$

In the formula, stimulus and response denote the events at the source and target port respectively, whereas the index i is used to match the i-th stimulus event occurrence with its corresponding response event occurrence. The TADL2 expression for the example illustrated in Figure 5.10 can be defined using Formula 5.5.

$$\forall i : \forall t_j = s_i \Rightarrow \exists t_k : t_k = r_i \wedge lower \leq t_k - t_j \leq upper \quad (5.5)$$

Where t_j denotes the timing point of the event-occurrence s_i and t_k denotes the timing point of event-occurrence r_i.

The following PSL property declaration can used to express Formula 5.4:

```
1 property StrongDelayConstraint (boolean stimulus, response; const
    lower, upper) = always ({rose(stimulus)} |=> {[*lower:upper];
    rose(response)}) @(posedge clock);
```

The proposed PSL property declaration states that an occurrence of the dataEvent is always expected within a time frame defined by the timing points lower and upper after the occurrence of the stimulus event. Furthermore, the property holds in a given cycle if and only if either the Sequence that is the left operand of the implication operator (| =>) ($\{rose(stimulus)\}$) does not hold at the given cycle, or the sequence that is the right operand ({[*lower:upper]; rose(response)}) immediately holds in a cycle following any cycle in which the sequence that is the left operand holds.

5.2.4 RepetitionConstraint

So far, we have introduced two basic timing constraints needed to formally express our timing extensions. The *RepetitionConstraint* describes the third basic timing constraint. In fact, this constraint extends the notion of repeated event occurrences by allowing local deviations from the ideal repetitive pattern specified by the RepeatConstraint.

Strictly speaking, the RepetitionConstraint is a timing constraint, that can be expressed using the RepeatConstraint and the StrongDelayConstraint [80]. The RepetitionConstraint is expressed as follows:

```
RepetitionConstraint (dataEvent, lower, upper, span, jitter) =
RepeatConstraint (X, lower, upper) and
StrongDelayConstraint (X, dataEvent, 0, jitter)
```

Where $jitter$, $lower$ and $upper$ are formal parameters needed to specify the allowed time window for each $dataEvent$-occurrence. $dataEvent$ refers to the event occurrences at a specific port of the component under investigation. Furthermore, X denotes reference timing points that are needed to constraint the event under observation ($dataEvent$). Moreover, X must satisfy the RepeatConstraint, and both X and $dataEvent$ must satisfy the StrongDelayConstraint.

Given the definitions of the RepeatConstraint in Formula 5.3 and the StrongDelayConstraint in Formula 5.4, the following logic formula can be derived for the RepetitionConstraint:

$$\begin{cases} (\forall Y \leq X : |Y| = span + 1 \Rightarrow lower \leq \lambda([Y]) \leq upper) \wedge \\ (\forall i : \forall x = X(i) \Rightarrow \exists y : y = dataEvent(i) \wedge 0 \leq y - x \leq jitter) \end{cases} \quad (5.6)$$

Figure 5.11 illustrates an example showing a system behavior that satisfies a RepetitionConstraint. The event-occurrences of the stimulus are the actual timing events under observation. These event occurrences are correlated

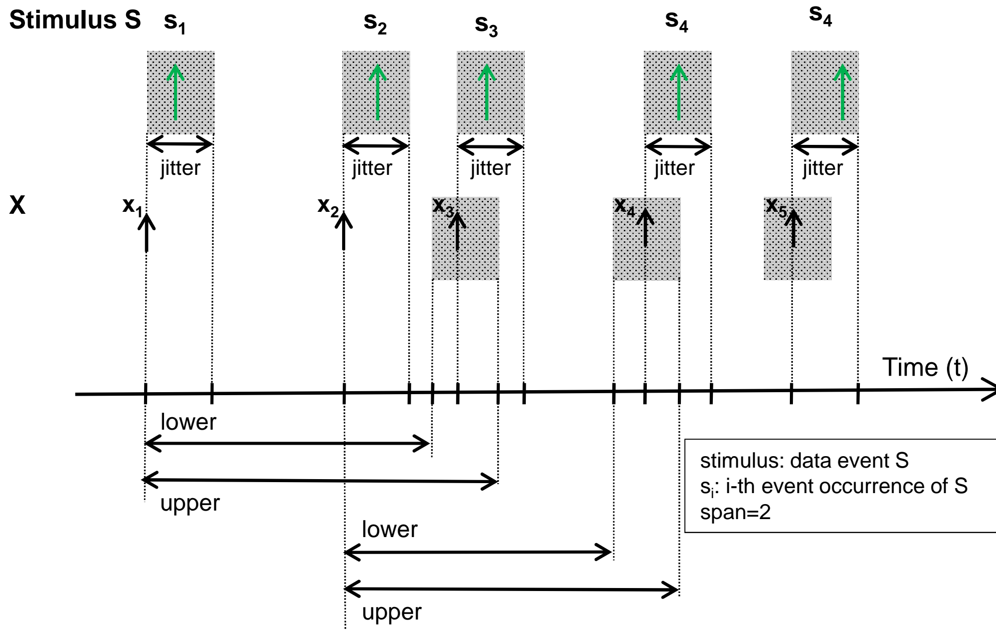


Figure 5.11: A set of event occurrences satisfying a RepetitionConstraint with $span = 2$

with occurrences of a reference event called X. After the occurrence of the reference event X, the stimulus should occur within a timing window specified by the parameter jitter. Furthermore, the reference event must fulfill the RepeatConstraint. The TADL2 expression satisfied by this example can be expressed using Formula 5.7.

$$\begin{cases} (\forall Y = \{x_i, ..., x_{i+2}\} \leq X : |Y| = 3 \Rightarrow lower \leq x_{i+2} \leq upper) \wedge \\ (\forall i : \forall t_j = x_i \Rightarrow \exists t_k : t_k = s_i \wedge 0 \leq t_k - t_j \leq jitter) \end{cases} \quad (5.7)$$

For $span = 1$, we propose the following PSL sequence and property declaration to express the RepetitionConstraint:

```
1 property RepetitionConstraint (boolean refEvent, dataEvent; const
    lower, upper, jitter) = always (({[*lower:upper];
    rose(refEvent)}) && ({rose(refEvent)} |=> {[*0:jitter];
    rose(dataEvent)}))@(posedge clock);
```

The SERE length-matching and operator (&&), constructs a so-called *compound* SERE in which the two compound SEREs $([*lower : upper]; rose(refEvent))$and $(rose(refEvent)| => [*0 : jitter]; rose(dataEvent))$ both hold and complete in the given cycle.

The following sections introduce our IP-XACT extensions. The figures used for the description of these extensions were all generated from the extended IP-XACT XML-schema using the XML schema editor tool XMLSpy[11].

Furthermore, the figures are used to depict the attributes of our timing constraints.

5.2.5 DelayConstraint

The *DelayConstraint* is one of the most important timing constraints. It is basically needed to constrain delays between the occurrence of a stimulus and its corresponding response. An ACC system is a typical example, where the duration could be specified between the reception of the desired vehicle speed requested by the driver and the observation of the current vehicle speed being equal to that desired speed.

Figure 5.12 shows the structure of the *DelayConstraint* element. This timing constraint always associates two causally related data events namely a stimulus with a response. Furthermore, the attributes *maximum* and *minimum* as shown in the uppermost box symbol need to be defined to be able to specify the range of acceptable delays between the occurrences of an event at the source port and the corresponding response at the target port. This information is crucial for time budgeting for instance during the integration of new components into existing designs. (see Section B.1.3 for further details about the notation)

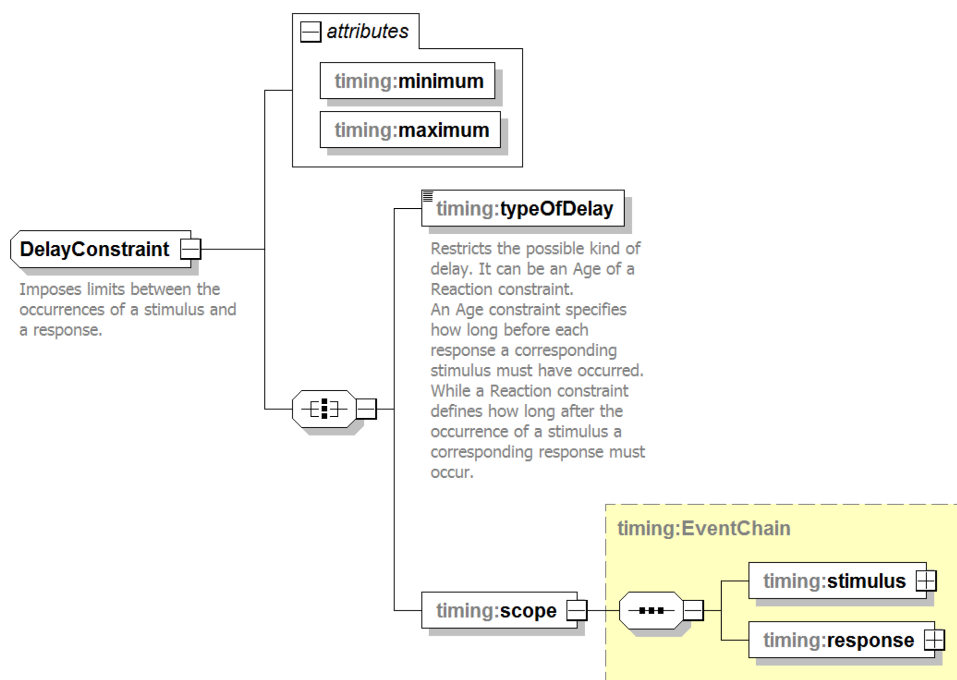


Figure 5.12: Graphical representation of the XML schema of the DelayConstraint element

Furthermore, we distinguish between two types of delay constraints, namely the Age- and the Reaction-Constraint. Both delay types have different semantics. This information can be specified using the attribute *typeOfDelay*. The AgeConstraint requires the latest occurrence of the stimulus to lie within the prescribed time bounds, whereas a ReactionConstraint restricts the earliest occurrence of the response to lie within the prescribed time bounds. In other words, a ReactionConstraint specifies a time window for the occurrence of each response.

The AgeConstraint is logically formulated TADL2 using Formula 5.8.

$$\forall y \in scope.response, \exists x \in scope.stimulus \wedge minimum \leq y - x \leq maximum \tag{5.8}$$

where *scope* is an eventChain containing the reference to the stimulus and response ports, as depicted in Figure 5.7. x and y denote the occurrence of the stimulus event and response event, respectively. In an airbag control system where various sensors values such as accelerometers need to be updated at a specific rate, an AgeConstraint can be specified using the following TADL2 expression:

$$\forall y \in collisionDetected, \exists x \in AccelValueRec \wedge 0 \leq y - x \leq 5ms \tag{5.9}$$

The ReactionConstraint is formulated in TADL2 using Formula 5.10.

$$\forall x \in scope.stimulus, \exists y \in scope.response \wedge minimum \leq y - x \leq maximum \tag{5.10}$$

In the formula, y denotes the first occurrence of the response event. A typical example is the activation time of an airbag control system after a collision has been detected. This constraint can be expressed in TADL2 as follows:

$$\forall x \in collisionDetected, \exists y \in gasGenPropellantIgnited \wedge 10ms \leq y - x \leq 20ms \tag{5.11}$$

Age- and Reaction-Constraints can be formalized and expressed with PSL as follows:

```
1 property ageConstraint(boolean stimulus, response; const minimum,
     maximum) = always ({stimulus[*]; [*minimum:maximum];
     response})@(posedge clock);
2 property reactionConstraint(boolean stimulus, response; const
     minimum, maximum) = always ({stimulus; [*minimum:maximum]
     response[*]})@(posedge clock);
```

The named properties *ageConstraint* and *reactionConstraint* involve the formal parameters stimulus and response. Additionally, the formal parameters minimum and maximum specify the minimum and maximum delay allowed respectively. Furthermore, these property-declarations describe behaviors in

which the response must occur within a certain number of clock cycles after the occurrence of the stimulus which is the enabling condition. As specified in TADL2, it is the last occurrence of the stimulus that is used as the enabling condition for the ageConstraint ($stimulus[*]$), whereas in the case of the reactionConstraint, it is the first occurrence of the stimulus that is relevant ($stimulus$).

5.2.6 SporadicConstraints

Figure 5.13 shows the structure of the SporadicConstraint element. As it can be seen in the attribute block, the attributes *lower*, *upper*, *jitter* and *minimum* are required in order to specify a SporadicConstraint. A SporadicConstraint can be expressed using both the Repeat- and RepetitionConstraint [80].

The main difference here is the fact that, the RepetitionConstraint is applied here with a default span attribute of $span = 1$. Moreover, the effective minimum distance $\lambda([Y])$ between two event occurrences must be greater or equal than the value given by the attribute *minimum* of the constraint, even if the constraint attributes $lower - jitter$ would suggest a smaller value. SporadicConstraints are useful for event-triggered systems or bus networks

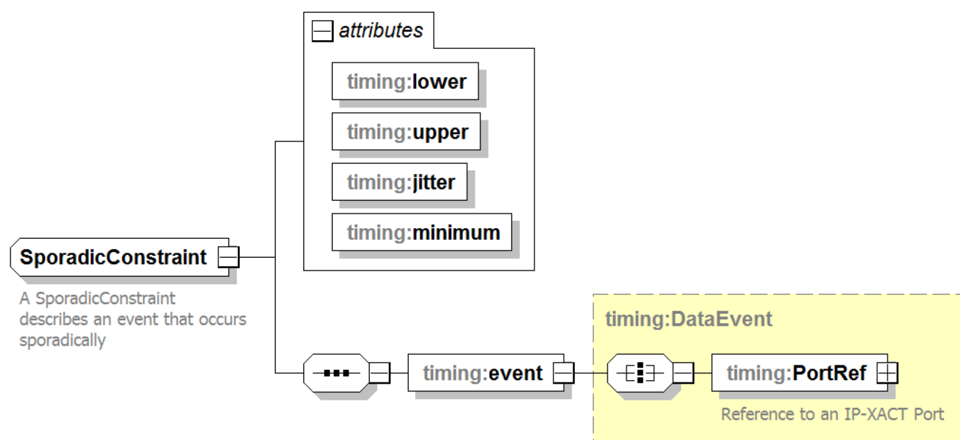


Figure 5.13: Sporadic constraint

such as CAN. Events that occur sporadically can be constrained using the SporadicConstraint.

In TADL2, the SporadicConstraint can be expressed as follows:

```
SporadicConstraint (dataEvent, lower, upper, jitter, minimum) =
RepetitionConstraint(dataEvent, lower, upper, jitter) and
RepeatConstraint(dataEvent, minimum, infinity)
```

Therefore, the following logic formula for the SporadicConstraint can be derived by combining both Formula 5.6 and Formula 5.3:

$$\begin{cases} (\forall Y \leq X : |Y| = 2 \Rightarrow lower \leq \lambda([Y]) \leq upper) \wedge \\ (\forall i : \forall x = X(i) \Rightarrow \exists y : y = dataEvent(i) \wedge 0 \leq y - x \leq jitter) \wedge \\ (\forall Y \leq dataEvent : |Y| = 2 \Rightarrow \lambda([Y]) \geq minimum) \end{cases} \tag{5.12}$$

Thus, we propose the following PSL property declaration to formalize and express a SporadicConstraints:

```
1 property sporadicConstraint (boolean refEvent, dataEvent; const
    lower, upper, jitter, minimum) = always (({[*lower:upper];
    rose(refEvent)}) && ({rose(refEvent)} |=> {[*0:jitter];
    rose(dataEvent)}) && ({[*lower:inf];
    rose(dataEvent)}))@(posedge clock);
```

In the property declaration of the SporadicConstraint, the formal parameter *refEvent* specifies a reference signal that satisfies a RepeatConstraint. Furthermore, both *refEvent* and *dataEvent* must satisfy the RepetitionConstraint. The transformation of the SporadicConstraint into PSL is correct, since it is basically a combination of Formulas 5.6 and 5.3.

5.2.7 Periodic constraints

PeriodicConstraints can be very useful in time-triggered bus networks such as FlexRay, where communication data is sent periodically based on the TDMA schedule. Loosely speaking, a PeriodicConstraint describes an activity that occurs periodically. This type of timing constraint can be applied in monitoring applications for example, where environment data such as temperature need to be periodically checked.

In fact, a PeriodicConstraint is a variant of SporadicConstraint where the attributes lower and upper are equal and denoted by the period attribute [80]. The attributes of the PeriodicConstraint can be seen in Figure 5.14. Thus, a PeriodicConstraint can be expressed using a SporadicConstraint as follows:

```
PeriodicConstraint (dataEvent, period, jitter, minimum) =
SporadicConstraint (dataEvent, period, period, jitter, minimum)
```

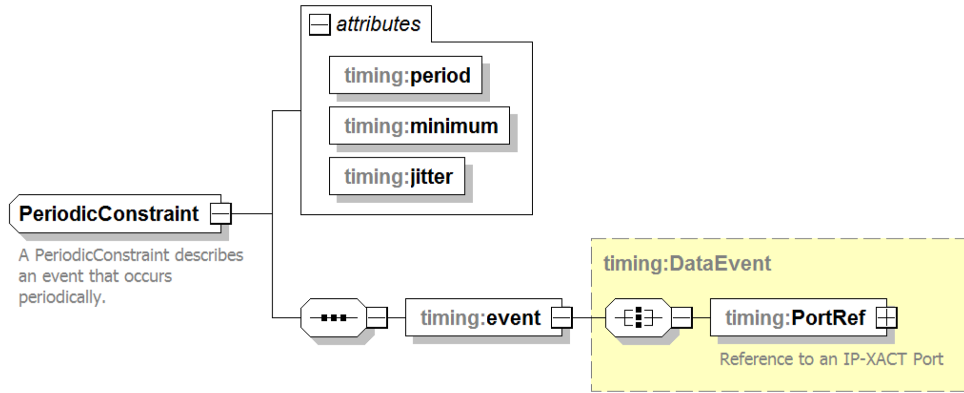


Figure 5.14: Periodic constraint

Formula 5.13 can therefore be derived from Formula 5.12

$$\begin{cases} (\forall Y \leq X : |Y| = span + 1 \Rightarrow \lambda([Y]) = period) \wedge \\ (\forall i : \forall x = X(i) \Rightarrow \exists y : y = dataEvent(i) \wedge 0 \leq y - x \leq jitter) \wedge \\ (\forall Y \leq dataEvent : |Y| = 2 \Rightarrow \lambda([Y]) \geq minimum) \end{cases} \tag{5.13}$$

Let's consider a brake-by-wire system where a specific wheel speed value is periodically sent on the FlexRay bus. The sampled wheel speed sensors values have to be written in the message buffer of the communication controller within a specified time window. The corresponding TADL2 expression can be specified as follows:

$$\begin{cases} (\forall Y = \{cycleStart_i, cycleStart_{i+1}\} \leq cycleStart : |Y| = 2 \Rightarrow cycleStart_{i+1} = period) \wedge \\ (\forall i : \forall t_j = cycleStart_i \Rightarrow \exists k : t_k = wheelSpeed_i \wedge 0 \leq t_k - t_j \leq jitter) \wedge \\ \forall X = \{wheelSpeed_i, wheelSpeed_{i+1}\} \leq wheelSpeed : wheelSpeed_{i+1} \geq minimum \end{cases} \tag{5.14}$$

Thus, we propose the following PSL property declaration:

```
1 property periodicConstraint (boolean refEvent, dataEvent; const
    period, jitter, minimum) = always (({[*period];
    rose(refEvent)}) && ({rose(refEvent)} |=> {[*0:jitter];
    rose(dataEvent)}) && ({[*period:inf];
    rose(dataEvent)}))@(posedge clock);
```

Since the named property-declaration periodicConstraint is just a variant of the sporadicConstraint, the derived PSL formula also semantically equivalent to the corresponding TADL2 expressions specified by Formula 5.13.

5.2.8 Synchronization Constraint

Basically, a synchronization constraint refers to an unbounded list of ports and specifies how tight the occurrences of a group of events should follow each other. Generally speaking, a system behavior satisfies a Synchronization constraint if and only if all associated events occur within a specified tolerance window, from a reference point in time. Multiple occurrences of the event within the tolerance window are allowed. Moreover, the aim of the synchronization constraint is to keep a consistent interaction between different components of a system. The importance of a Synchronization constraint has been described in Section 5.1.1 by means of the scenario depicted in Figure 5.1.

We distinguish between input- and output-synchronization constraints. Figure 5.1 illustrates an application example of the InputSynchronizationConstraint, that associates the stimuli a and b with the response c. As shown in the figure, the computation of the actuator value c requires the input values a and b. Therefore the events associated to a and b must occur before the event associated to c.

For meta-modeling reasons, we introduce two variations of the EventChain element, namely *EventChainIn* and *EventChainOut*. As depicted in Figure 5.15, an EventChainIn expresses the correlation between two or more stimuli and a response, whereas an EventChainOut correlates one stimulus with two or more responses (see Figure 5.16).

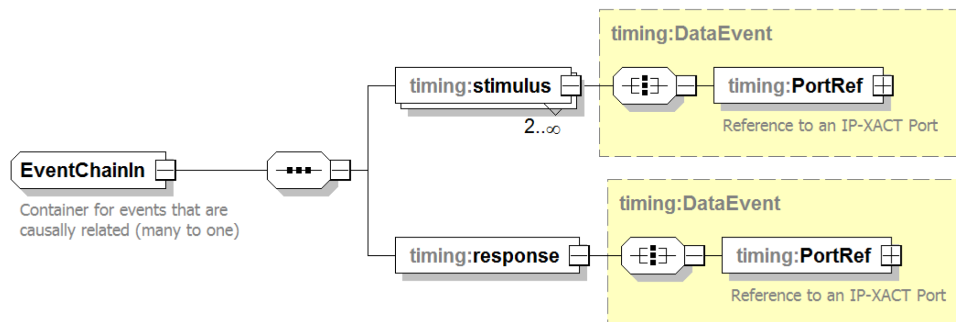


Figure 5.15: EventChainIn: constrains two or more stimuli with one response

Input Synchronization Constraint

The *InputSynchronizationConstraint* specifies how far apart correlated stimuli should occur. With this timing constraint, it is the latest of the event occurrences for each stimulus that is required to lie within the tolerance win-

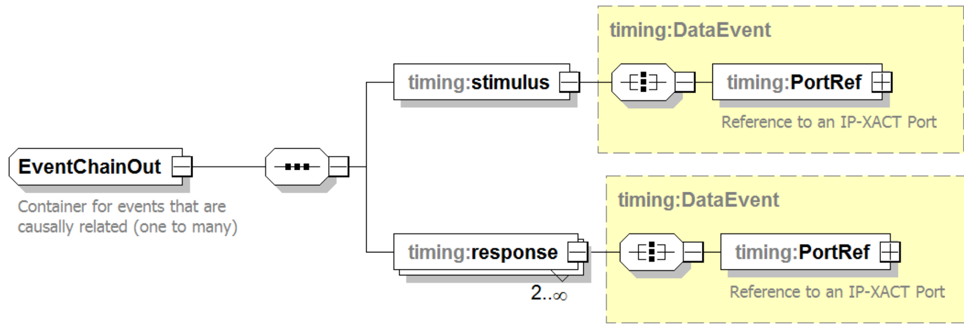


Figure 5.16: EventChainOut: constrains two or more responses with one stimulus

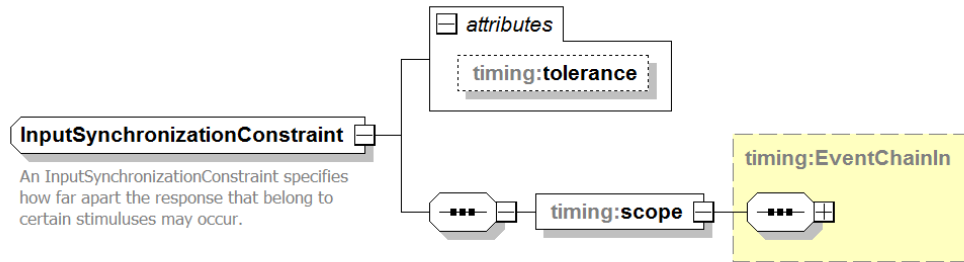


Figure 5.17: Input synchronization constraint

dow. The structure of the InputSynchronizationConstraint is depicted in Figure 5.17. The corresponding logic formula is defined in Formula 5.15.

$$\begin{cases} (\forall y \in scope.response : \exists t : \forall i : \exists x \in scope.stimulus_i) \wedge \\ (\forall i ; \forall x' \in scope.stimulus_i : x' \leq x) \wedge \\ (0 \leq x - t \leq tolerance) \end{cases} \tag{5.15}$$

where t is a reference point in time and $scope$ is from type eventChainIn (see Figure 5.15). Formula 5.15 states that a system behavior satisfies an InputSynchronizationConstraint c if and only if for each occurrence of a response y ($c.scope.response$), there is a time t such that for all stimuli ($c.scope.stimulus_i$), there is an occurrence of $stimulus_i$ (x) such that for all other occurrences x' of $stimulus_i$ x is maximal ($x' \leq x$) and the occurrence of x lies within the interval window ($0 \leq x - t \leq tolerance$).

Let's consider a BBW system where rotational wheel speed data must be simultaneous acquired by all ABS controllers within a time window of $360\mu s$ after each bus communication cycle start. The corresponding TADL2 expression can be formulated as follows:

$$\begin{cases} (\forall y \in wheelSpeedProcessed : \exists t : \forall i \in \{fr, fl, rr, rl\} : \exists x \in wheelSpeed_i) \wedge \\ (\forall i \in \{fr, fl, rr, rl\} ; \forall x' \in wheelSpeed_i : x' \leq x) \wedge \\ (0 \leq x - t \leq tolerance) \end{cases} \tag{5.16}$$

Given an EventChainIn with $n = 2$ stimuli and a reference event occurrence, that can be used as a reference timing point t, the InputSynchronizationConstraint can be expressed in PSL as follows:

```
1 property InputSynchronizationConstraint (boolean refEvent,
    stimulus_1, stimulus_2; const offset, tolerance) = always
    (({rose(refEvent)} |=> {[*offset:tolerance];
    rose(stimulus_1)}) && ({rose(refEvent)} |=>
    {[*offset:tolerance]; rose(stimulus_2)}))@(posedge clock);
```

The proposed InputSynchronizationConstraint is semantically equivalent to the TADL2 specification defined by Formula 5.15, since it also specifies a behavior, where all stimulus events should concurrently occur within the same tolerance window, from a reference point in time denoted by t. In the PSL formula, the dataEvent *refEvent* is used as reference event occurrence.

Output Synchronization Constraint

An *OutputSynchronizationConstraint* specifies how far apart the responses that belong to a specific stimulus must occur. All responses must refer to the same stimulus event. It is the earliest of the event occurrences for each response that is required to lie within the tolerance window.

A typical application example of this constraint is the ABS in a car, where the individual brake activations should be applied simultaneously on the four wheels. Figure 5.18 and Formula 5.17 show the attributes and the logic equivalence of the OutputSynchronizationConstraint respectively.

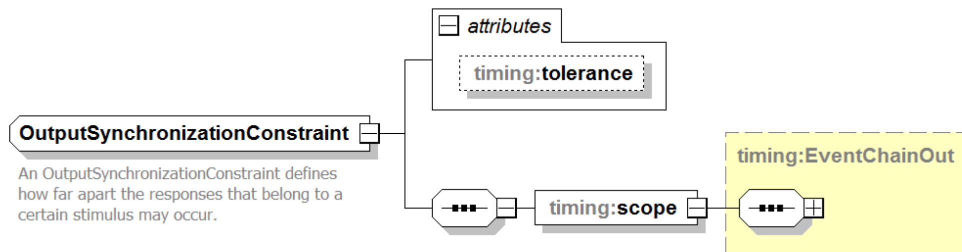


Figure 5.18: Synchronization Constraint

$$\begin{cases} (\forall y \in scope.stimulus : \exists t : \forall i; \exists y \in scope.response_i)\wedge \\ (\forall i; \forall y' \in scope.response_i : y \leq y')\wedge \\ (0 \leq y - t \leq tolerance) \end{cases} \tag{5.17}$$

where t denotes a reference point in time and scope is of type EventChainOut.

The OutputSynchronizationConstraint can be use to specify a system behavior, where the ABS torque values must be sent by all ABS controllers within

a time frame of $360\mu s$ after each bus communication cycle start. This can be done using the following TADL2 expression.

$$\begin{cases} (\forall y \in cycleStart : \exists t : \forall i \in \{fr, fl, rr, rl\}; \exists y \in absCtrl_i) \wedge \\ (\forall i \in \{fr, fl, rr, rl\}; \forall y' \in absCtrl_i : y \leq y') \wedge \\ (0 \leq y - t \leq tolerance) \end{cases} \tag{5.18}$$

Given an EventChainOut that composed of one stimulus and $n = 2$ responses, the OutputSynchronizationConstraint can be formalized and expressed with PSL as follows:

```
1 property outputSynchronizationConstraint (boolean stimulus,
    response_1, response_2; const offset, tolerance) = always
    (({rose(stimulus)} |=> {[*offset:tolerance];
    rose(response_1)}) && ({rose(stimulus)} |=>
    {[*offset:tolerance]; rose(response_2)}))@(posedge clock);
```

The proposed outputSynchronizationConstraint constraint is semantically equivalent to the TADL2 specification defined in Formula 5.17, since it also specifies a behavior, where the response-events should concurrently occur within the same tolerance window after each occurrence of the stimulus.

5.2.9 Order Constraint

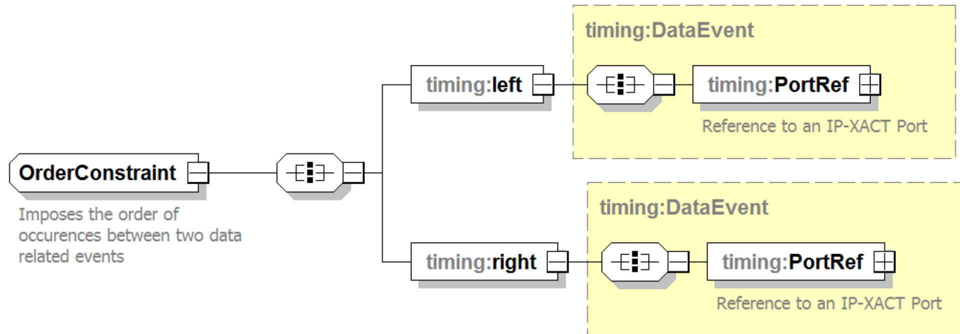


Figure 5.19: Order constraint

The *OrderConstraint* can basically be seen as a precedence constraint (see Figure 5.19). It imposes an order between the occurrence of two events. This timing constraint is particularly important for data synchronization, where the order of occurrence of read and write transactions should be maintained. Formula 5.19 shows the logical expression of the order constraint.

$$\forall i : (\exists x : x = left(i) \Rightarrow \exists y : y = right(i)) \wedge (x < y) \tag{5.19}$$

Formula 5.20 gives an example for the application of the OrderConstraint, where x and y denote the timing point of the occurrences of the events $wheelSpeed_i$ and $absCtrl_i$ respectively. The formula states that the wheelSpeed value has to be read before computing the ABS torque value.

$$\forall i : (\exists x : x = wheelSpeed_i \Rightarrow \exists y : y = absCtrl_i) \wedge (x < y) \tag{5.20}$$

An OrderConstraint can be formalized and expressed with PSL using the before directive as follows:

```
1 property orderProperty (boolean left; boolean right) = always
    ({left before! right})@(posedge clock);
```

The orderConstraint can be expressed in PSL using the before operator. In PSL, a *before* property holds in the current cycle of a given path if and only if either the left operand holds at the current cycle or at some future cycle, or the left operand holds strictly before the right operand holds, or the right operand never holds. Therefore, this directive is suitable to express a precedence constraint.

5.3 Verification of the timing properties

In a concrete simulation model, each PSL declaration will directly refer to signals of the DUV as well as to formal parameters. This will be realized by instantiating the PSL declaration. The instantiation of a PSL declaration creates an instance of the named declaration and provides an actual parameter for each formal parameter. In the instance created by the instantiation, each actual parameter expression in the actual parameter list of the instantiation replaces all references to the formal parameter in the corresponding position of the formal parameter list of the named declaration (please refer to [52], page 104 for further details).

The verification of the timing properties takes place in the third phase of our design flow (see Chapter 4). In this phase, the design model is verified using the specified timing properties.

The timing verification step is a tool dependent process step. Therefore, this process step will be explained in Chapter 6 using a case study. The case study discusses the verification of the SystemC model of a brake by wire system.

5.4 Summary

The main contributions in this chapter are a) the extension of the IP-XACT standard with timing extensions, b) the elaboration and definition of transformation rules from defined IP-XACT-timing-constraints to executable PSL formulas. Hereby, the transformation rules are compliant to the TADL2 semantic. The provided transformation rules could then be implemented by a PSL code generator, for instance.

The chosen timing constraints are originated from the Timing Augmented Description Language 2 (TADL2)(see Section 2.2.3). TADL2 is an output of the European funded research project TIMMO2USE. I was part of the project team that worked out the constraints definition and validation [82, 80].

Moreover, the focus lied here on the subset of the TADL2 timing constraints related to data transmission (send/write) and reception (receive/read). Generally speaking, a *timing event* denotes any form of identifiable state change of a system at run-time, that can be constrained with respect to timing. These changes can occur at distinct points in time; referred to as occurrences of the event.

Since the main focus of this thesis is the data synchronization, the presented set of timing constraints are typical for the application area considered in this thesis which the design of automotive bus network.

The semantic behind the timing constraint elements defined by TADL2 is specified using a first order logic notation, which make the transformation to PSL a lot easier since PSL is a language that can be used to formulate standard temporal logics formulas [52].

As mentioned in the previous section, the verification of the timing properties takes place in the third phase of our design flow (see Chapter 4). Timing verification is a tool dependent process step. Therefore, this process step will be demonstrated in Chapter 6 using a case study. The case study discusses the verification of the SystemC model of a brake by wire system.

In the chapter we do not provide a proof of correctness of our transformation rules. Therefore the example used in Chapter 6 will also help validate our transformation rules.

Chapter 6

Verification of timing properties: case study Brake-By-Wire

This chapter describes the timing verification approach by means of an automotive Brake-By-Wire example. This helps evaluate the third phase of our Restbus simulation design flow.

In order to evaluate this phase of the design flow, the model under verification will first be described. Afterward, timing requirements will be specified. Based on the specification of the timing requirements to be verified, corresponding PSL formulas will be derived. Finally, the results obtained after the verification process will be presented.

The system model used in this case study is a SystemC model of an automotive BBW application. The model was developed in C-LAB for research purposes. The application is distributed over a set of ECUs and includes ABS functionality. Using this simulation model, the following timing constraints could be specified:

- RepeatConstraint
- StrongDelayConstraint
- RepetitionConstraint
- InputSynchronizationConstraint
- OutputSynchronizationConstraint
- DelayConstraint

6.1 Functional decomposition of the BBW model

The control algorithms of the BBW model was modeled using MATLAB/Simulink [68]. The block diagram of the Simulink model is depicted in Figure 6.1.

Generally speaking, x-By-Wire characterizes the replacement of traditional components such as pumps, hoses, fluids, belts, vacuum servos and master cylinders with electronic sensors and actuators [16]. All brake components are electronically controlled.

A typical BBW system is composed of two main functions that are implemented by the Brake- and ABS-Controller. The Brake controller reads the wheel speed and brake pedal sensor data to compute the desired brake torque to be applied at the four wheels. The second functionality realized by the ABS controller is needed to adapt the brake force applied on each wheel if the speed of one wheel is significantly smaller than the estimated vehicle speed. In this case, the brake force is reduced on that wheel until it regains a speed that corresponds to the estimated vehicle speed. The ABS controller acquires wheel sensor data from each wheel and determines the estimated vehicle speed [81].

The SystemC simulation model was built from C code generated using the MATLAB tool Real-Time Workshop [69] from a Simulink model developed within the department. Basically the C code of the BBW model was wrapped into SystemC modules.

Afterward, a simulation model of the FlexRay bus communication network was added to the overall simulation model. The topology of the simulation model can is depicted in Figure 6.2.

As shown in the figure, the application is distributed over a network of five controller nodes: four ABS controllers and a so-called *CarEnvModel* node. Each ABS node implements the aforementioned functionality for a specific wheel. In our simulation model, the CarEnvModel node implements the Brake controller functionality for all wheels. Moreover, its functionality incorporates both the simulation of the vehicle dynamics and a stimulus generator. The Stimulus generator basically generates data representing the pressure applied on the brake and acceleration pedals.

The verification of the timing properties is an activity performed in phase 3 of the design methodology (see Chapter 4). However, the timing constraints to be verified first need to be defined, which is part of phase 2 of the design process. Since our focus lies on data synchronization, detailed knowledge about the communication matrix specifying which nodes communicate with each other and the underlying communication schedule is necessary (see Section 2.3.2).

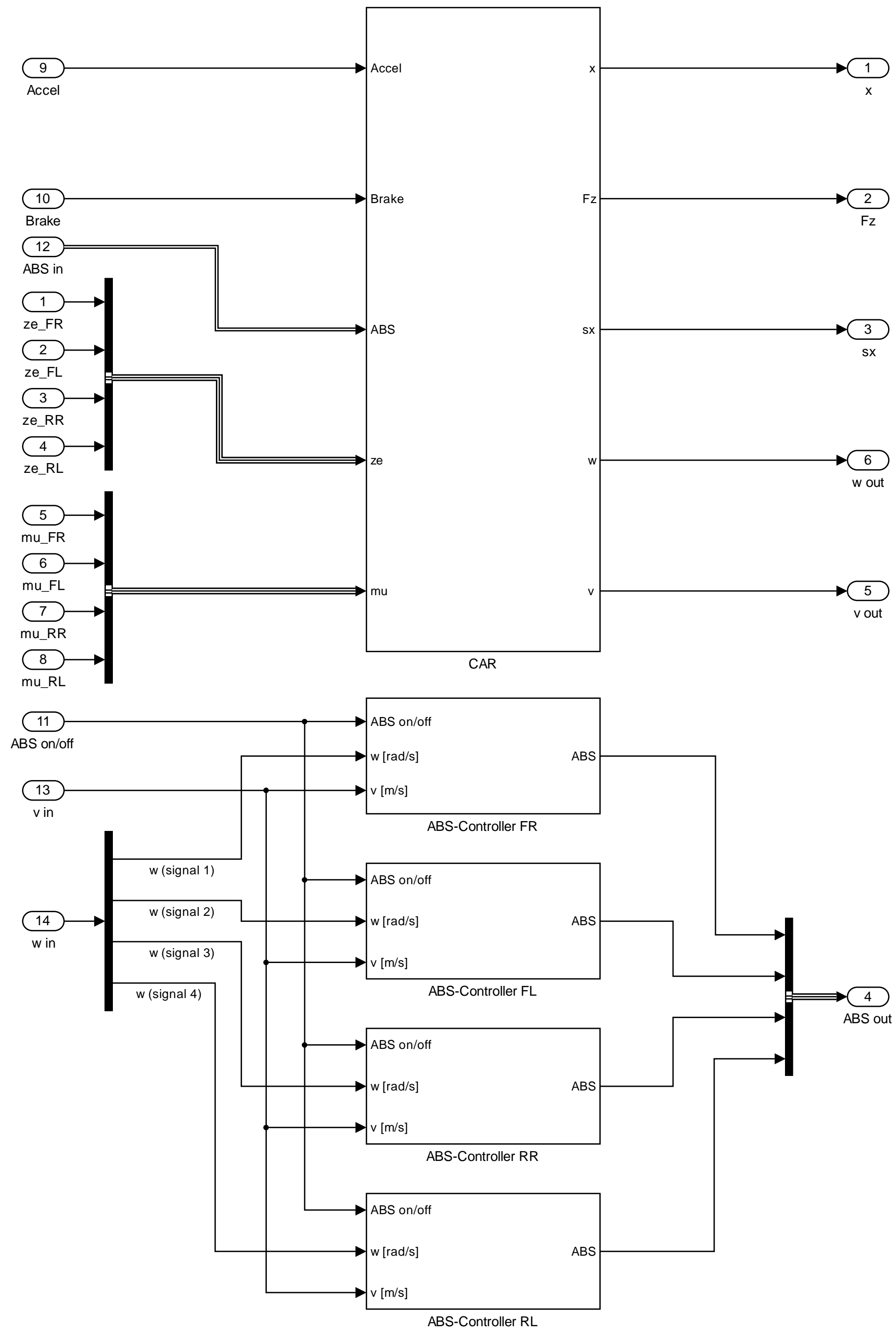


Figure 6.1: Block diagram of the ABS system, Source: C-Lab

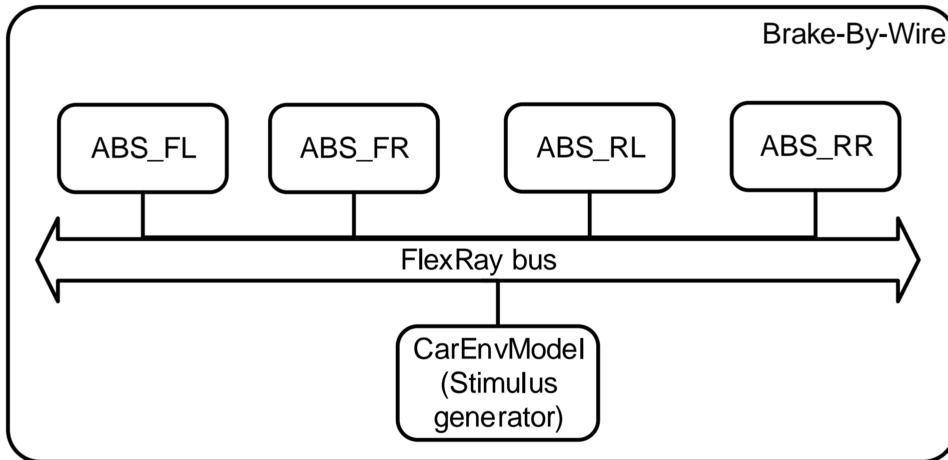


Figure 6.2: Network topology of the functional components of the BBW model

Based on this information, timing requirements can then be derived. Table 6.1 shows the communication matrix and FlexRay schedule of the BBW simulation model. The data provided by the CarEnvModel node are wheel speed (w_XX) for each wheel and the actual vehicle speed (V). These data are required by each ABS controller node to compute the actual brake torque (abs_XX) to be applied on each wheel.

Moreover, the configuration of the FlexRay network was done using the FlexRay network configuration tool flexConfig [38]. The bus communication cycle duration was set to one millisecond ($gdCycle = 1ms$) [41], which was the sampling rate of the MATLAB models.

The FlexRay communication cycle was segmented into a static segment with a duration of $400us$ and a Network Idle Time (NIT) with a duration of $600us$. No dynamic segment nor symbol window were used[1]. The static segment is further subdivided into 10 communication slots.

Slots	1	2	3	4	5	6	7	8	9	10
Data	V	w_FR	w_FL	w_RR	w_RL	abs_FR	abs_FL	abs_RR	abs_RL	-
CAR	w	w	w	w	w	r	r	r	r	-
ABSCtrlFR	r	r	-	-	-	w	-	-	-	-
ABSCtrlFL	r	-	r	-	-	-	w	-	-	-
ABSCtrlRR	r	-	-	r	-	-	-	w	-	-
ABSCtrlRL	r	-	-	-	r	-	-	-	w	-

Table 6.1: BBW communication matrix

[1]See Section 2.3.2 for more detail on the definition of the FlexRay communication schedule

6.2 Instrumenting of the simulation model

The goal of the verification process in this case study was to find out if *network* data are sent and received "on time" by the respective controller nodes according to the predefined FlexRay communication schedule depicted in Table 6.1. Since our main application scenario in this thesis is the integration and reuse of existing components, the design under verification should dispose of appropriate interfaces in order to monitor and analyze the design.

This section describes a possible approach that can be used to make data relevant transaction visible in the overall test-environment. In this case study, the interfaces of the controller nodes were extended to be able to signal read and write transactions related to the specified timing events.

For instance, to notify the transmission of the brake torque by an ABS controller, the signal absCtrlVR_sent was added to the ABS controller model. Following this approach, interfaces of all network nodes were systematically extended.

6.3 Reference model

Concerning the actual validation of the derived PSL formulas, we generated a reference waveform displaying event traces from the expected design behavior during a simulation run. The generated waveform is shown in Figure 6.3. As depicted in the figure, the waveform reflects the predefined FlexRay communication schedule presented in Table 6.1. The generation of the event traces figure was performed using the EDA tool QuestaSim [45].

Thus, this waveform provides a *golden model* that will be used later during the comparison of the simulated behavior with the expected behavior. Additionally, this also helps us validate our defined PSL transformation rules.

Moreover, Figure 6.3 displays all related events that should occur during one communication cycle of the predefined FlexRay cycle. Furthermore, event occurrences are depicted by a rising edge in the signal path. As already mentioned, our communication cycle has a duration of $1ms$.

The clock used by the components for the notification of the transaction events was configured with a period of $40us$ which corresponds to the duration of the slot duration within the FlexRay schedule.

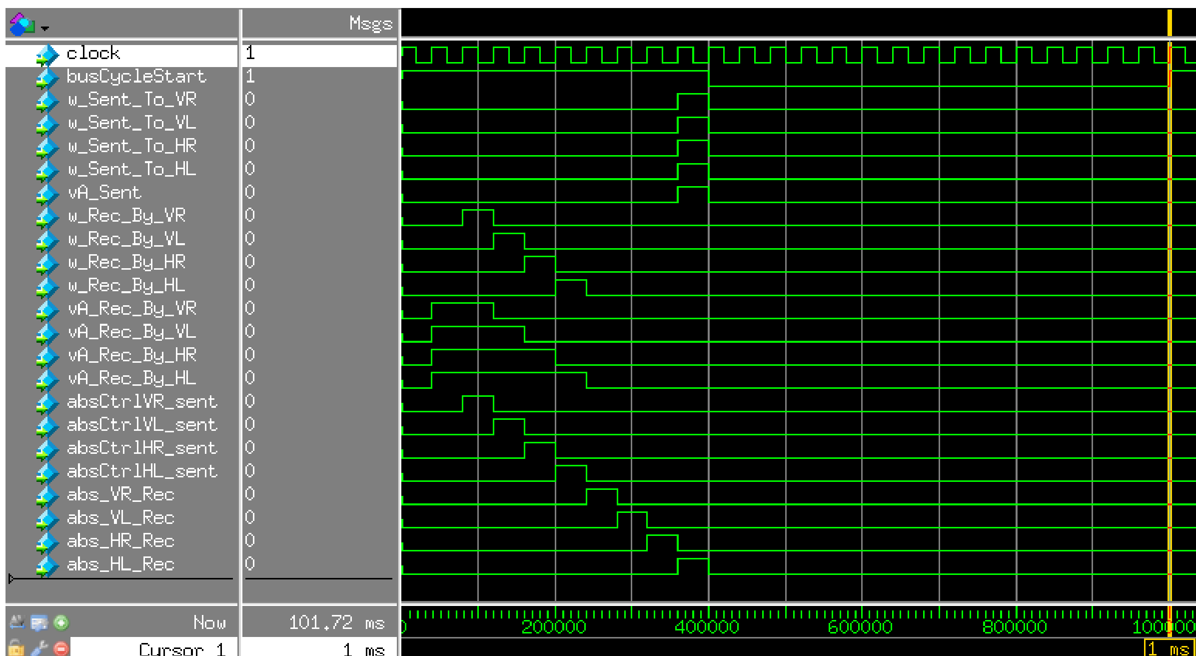


Figure 6.3: Waveform of the simulation run showing traces of all data events under observation

6.4 Specifying the timing requirements

The timing requirements were derived from the bus communication schedule. In this case study, only the static segment of the FlexRay communication schedule defined in Section 2.3.2 was used to configure the network bus communication. As a result, the bus communication cycle is composed of the static segment subdivided into 10 equal slots with a slot duration of $40\mu s$ and a NIT of $600\mu s$. This results in a communication cycle duration of $1ms$. The communication matrix can be seen in Table 6.1.

The following requirements were derived:

1. RepeatConstraint: The wheel speed value must be repeatedly transmitted by the CarEnvModel to the ABS_FR controller. Therefore, the notification of transmission by means of the event (w_Sent_To_VR) must always occur once within a time range of $[0 : 1]$ *milliseconds*. Since the clock used for the evaluation of of the PSL properties is configured with a period of $40\mu s$, this corresponds to the following range: $[0 : 25]$ clock ticks.

2. StrongDelayConstraint: The notification of the event absCtrlVR_sent must always occur within a range of $[0 : 1]$ *milliseconds* ($[0 : 25]$ clock ticks) immediately after the notification of the event vA_sent.

3. RepetitionConstraint: The notification of the event w_Sent_To_VR can and must always occur within a range of [0 : 1] *milliseconds* ([0 : 25] clock ticks).

4. AgeConstraint: The vehicle speed data processed by the ABS Controller must be at most 1 *milliseconds* old. This corresponds to the bus communication cycle duration.

5. ReactionConstraint: After the reception of the vehicle speed value, the ABS torque value must be computed and transmitted within a time frame of $[0 : 200]\mu s$

6. InputSynchronizationConstraint: The wheel speed data must be simultaneous acquired by all ABS controllers within a time window of $360\mu s$ after each bus communication cycle start.

7. OutputSynchronizationConstraint: The ABS torque values must be sent by all ABS controllers within a time frame of $360\mu s$ after each bus communication cycle start.

6.5 Evaluation results

The evaluation was performed using the EDA tool QuestaSim [45]. Questasim is an electronic design verification tool from Mentor Graphic that provides an advanced verification platform with multi-language simulation support. The languages supported are SystemVerilog, SystemC, VHDL, Verilog and PSL. The verification unit used used to group all verification directives and PSL statements is shown in Appendix A.

The evaluation results are presented using the following structure: For each timing requirement, the corresponding PSL assertion directive will be shown, additionally waveforms displaying each assertion directive will be shown. Finally, a table summarizing the number of assertion passes and fails will be given. The waveform window can be interpreted as follows: triangles pointing downwards (red) indicate that the given assertion of the PSL property has failed, whereas the triangles pointing upwards (green) indicate that the assertion has passed. The simulation of the BBW model was done for a duration of $81ms$, which corresponds to the simulation of 81 FlexRay communication cycles.

6.5.1 Repeat, StrongDelay and Repetition timing constraints

The following Property definitions and assertions list the basic timing constraints derived from the requirements specification. Figure 6.4 depicts a

waveform showing an extract of the event traces that resulted the simulation run.

```
1 property RepeatConstraint;
2 RepeatConstraint = always {[*0:25];
     rose(w_Sent_To_VR)}@(posedge clock);
3 assert RepeatConstraint;
```

```
1 property StrongDelayConstraint;
2 StrongDelayConstraint = always {rose(vA_sent)} |=> {[*0:25];
     rose(absCtrlVR_sent)}@(posedge clock);
3 assert StrongDelayConstraint;
```

```
1 property RepetitionConstraint;
2 RepetitionConstraint = always (({[*0:25]; rose(vA_Rec_By_VR)})
     && ({rose(vA_Rec_By_VR)} |=> {[*0:8];
     rose(absCtrlVR_sent)}))@(posedge clock);
3 assert RepetitionConstraint;
```

As a reference timing point for the definition of the RepetitionConstraint, the event vA_Rec_By_VR was chosen (denoted by X in Formula 5.6). Concerning the RepeatConstraint, the specified range ($[0:1]milisecond$) was imposed on the event vA_Rec_By_VR, and a StrongDelayConstraint with the range $[0:400]\mu s$ was imposed between the occurrences of the events vA_Rec_By_VR and the actual event under observation absCtrlVR_sent.

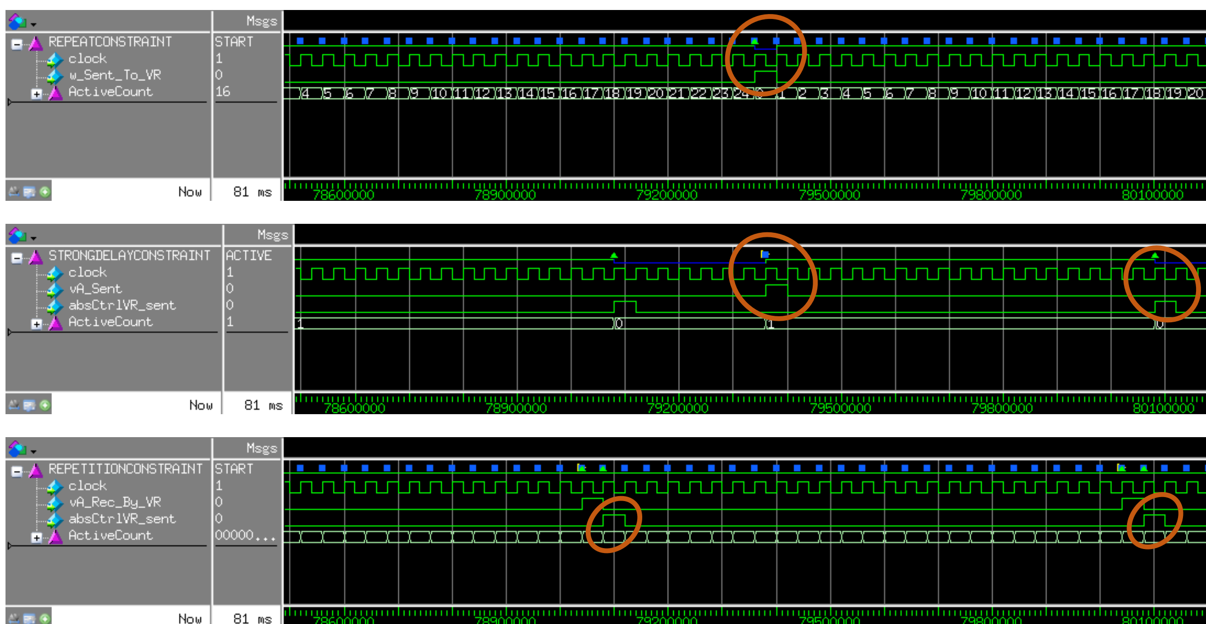


Figure 6.4: Assertion analysis window for the basic timing constraints

The waveform depicted in Figure 6.4 shows an extract of the event traces. All circles highlight some possible assertion results. Concerning the StrongDelayConstraint, the first circle highlights the occurrence of the stimulus event

(*vA_Sent*) whereas the second one highlights the occurrence of the response event (*absCtrlVR_sent*). In the third windows of the figure, the expected occurrences of the timing events are highlighted. The overall results of the analysis generated by the verification tool is summarized in Table 6.2. All specified basic constraints were satisfied.

Assertion Results	Repeat	StrongDelay	Repetition
Failure	0	0	0
Passes	2010	80	2002

Table 6.2: Assertion analysis results for Repeat-, StrongDelay-, and RepetitionConstraint

6.5.2 Evaluation of the AgeConstraint and ReactionConstraint

The following Listings show the PSL property instantiations of the AgeConstraint and ReactionConstraint derived from the requirements specification.

```
1 property AgeConstraint;
2 AgeConstraint = always {vA_Sent[*0:inf]; [*0:25];
     absCtrlVL_sent}@(posedge clock)
3 assert AgeConstraint;
```

```
1 property ReactionConstraint;
2 ReactionConstraint = always {vA_Rec_By_HL; [*0:5];
     absCtrlHL_sent[*0:inf]}@(posedge clock)
3 assert ReactionConstraint;
```

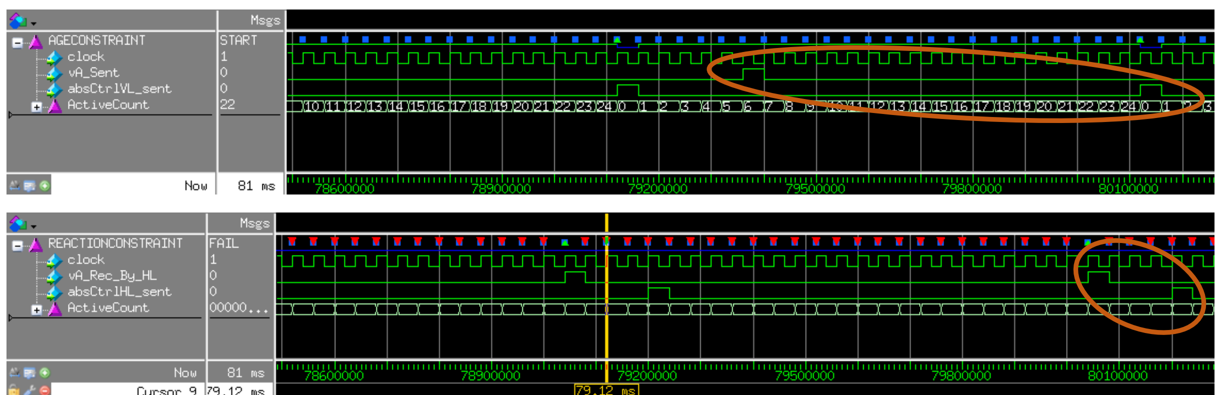


Figure 6.5: Assertion analysis window for Age- and ReactionConstraints

Figure 6.5 depicts an extract of the recorded timing event traces related to the AgeConstraint and ReactionConstraint. Furthermore, the figure shows for illustrative purposes some correlated event occurrences. Table 6.3 shows

the overall results obtained after the simulation run. For the AgeConstraint, all assertions passed, whereas 1941 assertions out of 2026 failed for the ReactionConstraint. The evaluation of the ReactionConstraint could help detect timing errors, which is an indication that the FlexRay schedule configuration needs to be optimized.

Assertion results	AgeConstraint	ReactionConstraint
Failure	0	1941
Passes	2004	85

Table 6.3: Assertion analysis results for AgeConstraint and ReactionConstraint

6.5.3 Evaluation of synchronization related timing Constraints

The assertion directives derived for the synchronization constraint can be viewed in the following Listings. As derived for the timing requirements, the InputSynchronizationConstraint was specified on the wheel speed values received by all ABS controllers. The OutputSynchronizationConstraint was specified on the ABS torque values sent by all ABS controllers. Both constraints had a tolerance window of 360*us* starting from the notification of the bus communication cycle start event.

```
1 property InputSynchronizationConstraint;
2 InputSynchronizationConstraint = always ({rose(busCycleStart)}
     |=> {[*0:9]; rose(w_Rec_By_VR)}) && ({rose(busCycleStart)}
     |=> {[*0:9]; rose(w_Rec_By_VL)}) @(posedge clock));
3 assert InputSynchronizationConstraint;
```

```
1 property OutputSynchronizationConstraint;
2 OutputSynchronizationConstraint = always ({rose(busCycleStart)}
     |=> {[*0:9]; rose(absCtrlVR_sent)}) && ({rose(busCycleStart)}
     |=> {[*0:9]; rose(absCtrlVL_sent)}) @(posedge clock));
3 assert OutputSynchronizationConstraint;
```

As shown in Table 6.4 no assertion failed during the simulation run. The corresponding waveform is depicted in Figure 6.6. The circles highlight some correlated event occurrences.

Assertion Results	InputSynchronization	OutputSynchronization
Failure	0	0
Passes	80	80

Table 6.4: Assertion result analysis for Synchronization related constraint

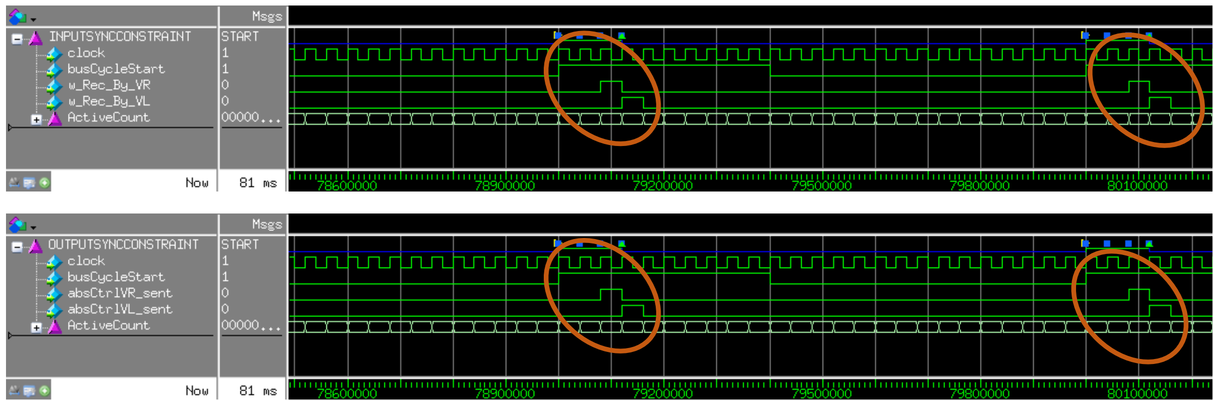


Figure 6.6: Assertion analysis window for SynchronizationConstraints

6.6 Summary and discussion

As one of the main contributions of this thesis we have developed an approach how to conduct the timing verification process. This was performed using the EDA tool QuestaSim [45].

In this chapter, we have demonstrated the feasibility of our timing verification approach by a case study consisting of a SystemC model of a Brake-By-Wire system including die ABS functionality. Since the design model used was already correct with respect to the FlexRay communication schedule as shown by the generated simulation traces in Figure 6.3, this model was also used as a reference model. To be able to apply this approach, the design interfaces under observation shall be made visible to the verification environment of the simulation tool. In some cases depending on the model of the design under verification, this step might require a prior instrumentation of the design model, which is a relatively simple task.

Some timing errors regarding the ReactionConstraints were detected. The violation of the ReactionConstraints was related to the granularity of the model. All timing constraints related to the communication schedule were fulfilled by the SystemC model as expected. Therefore, no contradiction regarding the correctness of the transformation rules were found.

This chapter also demonstrates the correctness of the transformation rules from TADL2 to PSL as introduced in Chapter 5. However, it does not provide a formal proof of the correctness of the transformation rules.

Chapter 7

Synchronization

7.1 Background

7.1.1 Data smoothing: Robust LOWESS/LOESS

Before going into the details of data smoothing, let's first specify what we mean with the term *data sequence*. Data that come as paired observations are usually displayed by drawing an $x-y$ plot. Plots of y versus x show at a glance how x and y are related to each other. When the x-values are equally spaced, their structure is so simple and regular that y often receives most of the attention. A list of such data may even omit the x-values in favor of reporting the interval at which the data were recorded.

In this thesis, the term *data sequence* refers to the y-values. Monthly unemployment rate or daily high and low temperatures at a weather station are typical examples. When the sequence comes about by recording a value for each successive time interval, as in these examples, the y-values are also called *time series*, since the order of the data values in the sequence is defined by time. In this thesis, the data sequences in question are sequences where the x-values are equally spaced. We refer to [94] for further details.

LOWESS/LOESS [1] is an acronym for "Locally Weighted Scatter-plot Smoothing". This data smoothing technique is also known as locally weighted polynomial regression and was introduced by Cleveland in 1979 [25]. It is a very popular curve fitting technique that works well on large, and densely sampled data sets.

[1] The differentiation between LOWESS and LOESS is ambiguous. Beside the fact that they both use locally weighted linear regression for data smoothing, they are referred to in some documents as being two different names describing the same method [26, 71], but in others they are differentiated by the model they use for the regression process. In [93] for instance, LOWESS is described as a method that uses a linear polynomial, while LOESS uses a quadratic polynomial. In this thesis, the LOWESS variant will be used.

Let $N_{i=1}^{n} = \{X_1, \ldots, X_n\}$ denote a data sequence of n two-dimensional data points $X_i = (x_i, y_i)$ with $i = 1, \ldots, n$. Further, let the smoothing parameter be f with $0 < f \leq 1$ and let the span $L = f \cdot n$ be rounded up to the next larger integer, where L denotes the number of data points to be used for the computation of each fitted value. Moreover, increasing f also increases the size of the smoothing window, hence the neighborhood of influential points and therefore, the smoothness of the smoothed points.

The data smoothing procedure Robust LOWESS is based on robust locally weighted regression.

During the simple locally weighted regression process, weights (also referred to as horizontal weights) $w_k(x_i)$ are computed for all data points X_k, $k = 1, \ldots, n$ in the neighborhood of X_i using a weight function W. The weight function has to be chosen in such a way that $w_k(x_i)$ becomes zero for all x_k that are out of the data smoothing window defined by the span L.

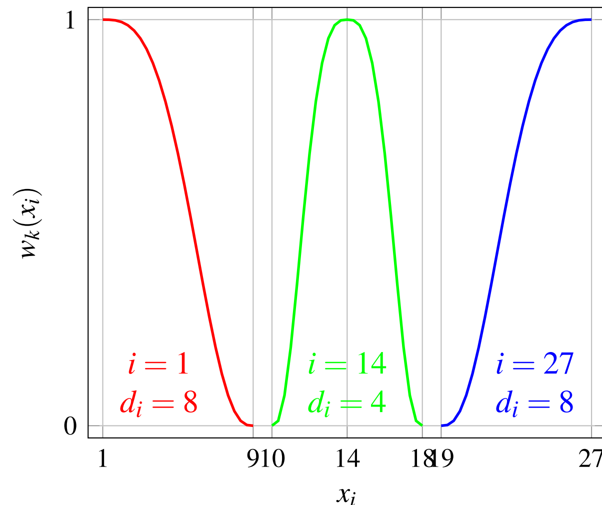


Figure 7.1: Weight function example $w_k(x_i)$ for leftmost ($i = 1$), interior ($i = 14$) and rightmost ($i = 27$) data points (span $L = 9$, f = 0.3, n = 29)

Figure 7.1 depicts the weight function for end- and for interior data point with a span of $L = 9$. As depicted in the figure, the weight function is symmetric, if the smooth calculation involves the same number of neighboring data points on both sides of the data point to be fitted. Furthermore, the value smoothing parameter never changes during the whole data smoothing process. For the leftmost ($i = 1$) and rightmost ($i = 27$) data points respectively, the shape of the weight function is truncated by half. Moreover, for $k = i$, $w_k(x_i)$ is the maximum weight.

On the other hand, the robust locally weighted regression variant includes an additional calculation of robust weights (also referred to as vertical weights), which makes the overall process more resistant to outliers. The computation

of the robust weights is based on the value of the residuals $y_i - \tilde{y}_i$, where $\tilde{y}_i$ is the smoothed value obtained from a previous smoothing iteration. Large residuals result in small weights and small residuals in large weights.

Figure 7.2 depicts the difference between both locally weighted regression approaches. As it can be seen, the robust regression variant depicted with blue triangles, is more resistant to outliers in contrast to the standard variant (depicted with red squares). The original data sequence is depicted with points.

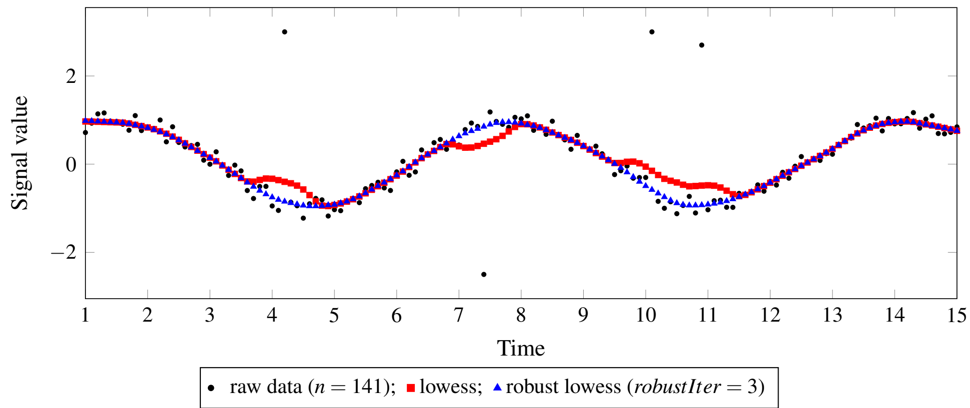


Figure 7.2: Locally weighted regression vs robust locally weighted regression, $L = 15$, $f = 0.1$)

Basically, the data smoothing procedure is designed to accommodate data for which

$$y_i = g(x_i) + \varepsilon_i$$

holds, where $g(x)$ is the smooth function and ε_i is a random variable [25]. $\tilde{y}_i$ is an estimate of $g(x_i)$.

For each i let d_i be the distance along the x-axis from x_i to the outermost data point within the smoothing window. Thus, the Robust LOWESS algorithm procedure is defined by the following steps:

Step 1 For each data point x_i compute the estimates $\tilde{\beta}_j(x_i)$, $j = 0, ..., n$, of the parameters in a polynomial regression of degree d using the weighted least square method [2] with weights $w_k(x_i)$ for $X_k = (x_k, y_k)$. The $\tilde{\beta}_j(x_i)$ are the values of β_j that minimize

$$\sum_{k=1}^{n} w_k(x_i)(y_k - \beta_0 - \beta_1 x_k - ... - \beta_d {x_k}^d)^2.$$

where

$$w_k(x_i) = W\left(\frac{x_k - x_i}{d_i}\right)$$

[2]A least square method minimizes the square distance of every data point and the line of best fit and finds the coefficients of a polynomial $P(X)$ of degree d that fits the data.

for $k = 1, \ldots, n$.

Typically, the following tri-cube function is used as weight function:

$$W(x) = \begin{cases} (1 - |x|^3)^3 & \text{for } |x| < 1 \\ 0 & \text{for } |x| \geq 1. \end{cases} \tag{7.1}$$

Furthermore, a polynomial of degree $d = 1$ (local linear regression) or degree $d = 2$ (local quadratic regression) is typically used, but higher order polynomials are also possible. Thus, the smoothed data point at $X_i = (x_i, y_i)$ is $\tilde{X}_i = (x_i, \tilde{y}_i)$ where $\tilde{y}_i$ is the smoothed value of the regression at x_i and

$$\tilde{y}_i = \sum_{j=0}^{d} \tilde{\beta}_j(x_i) x_i{}^j.$$

Step 2 Let B be the bi-square weight function that is defined by

$$B(x) = \begin{cases} (1 - x^2)^2, & \text{for } |x| < 1 \\ 0, & \text{for } |x| \geq 1, \end{cases} \tag{7.2}$$

and let $\varepsilon_i = y_i - \tilde{y}_i$ be the residuals from the current smoothed values. Let MAD be the median of the absolute deviations $|\varepsilon_i|$. In statistics, Median Absolute Deviation (MAD) is a robust alternative for the estimation of the standard-deviation [67]. For an univariate data set $x_1, \cdots, x_n$ the MAD is computed as follows:

$$MAD = Median_i(|x_i - Median_j(x_j)|) \quad \text{,with } 1 \leq i, j \leq n. \tag{7.3}$$

The robustness weights are defined by

$$\delta_k = B\left(\frac{\varepsilon_k}{6MAD}\right).$$

Step 3 Compute new $\tilde{y}_i$ for each i by fitting a d-th degree polynomial using weighted least square method with weight $\delta_k \cdot w_k(x_i)$ at (x_k, y_k).

Step 4 Repeatedly carry out steps 2 and 3 a total of *robustIter* times. The final smoothed value $\tilde{y}_i$ is the robust locally weighted regression fitted value for each $i = 1, 2, \cdots, n$ and the parameter *robustIter* that specifies the number of iterations should be specified.

Using the LOWESS method with a span of five, the smoothed values and associated regressions for the first four data points of a generated data set are shown in Figure 7.3. Notice that the span does not change as the smoothing

process progresses from data point to data point. However, depending on the number of nearest neighbors, the regression weight function might not be symmetric about the data point to be smoothed. In particular, plots (a) and (b) use an asymmetric weight function, while plots (c) and (d) use a symmetric weight function.

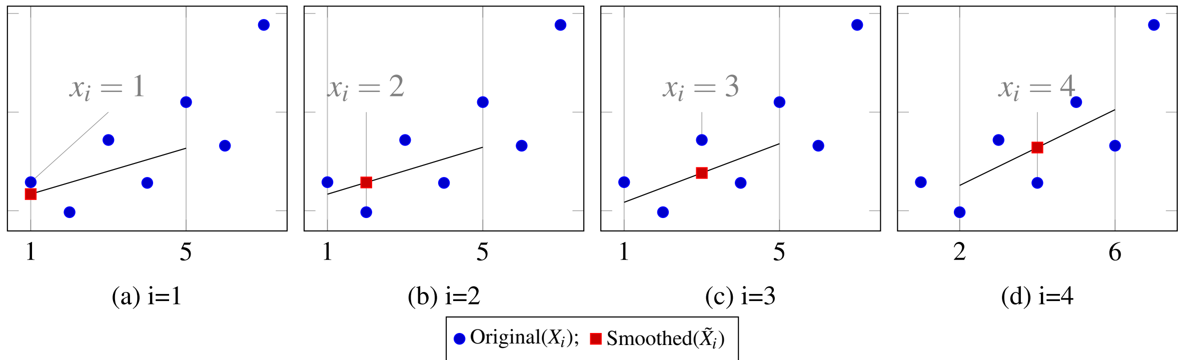


Figure 7.3: Weighted regression example for $i = 1, \ldots, 4$ and $L = 5$

7.1.2 Multirate Systems

This section gives a rough introduction into the principles and techniques that can be used to adapt the sample rate of Multirate systems. For a more detailed description please refer to [92, 32, 75].

Basically, there exist two types of signals, namely *discrete* and *continuous*-time signals. A continuous-time signal is a time function $x(t)$, which is defined for all time t in an interval, usually an infinite interval. Whereas discrete time signals only have values at discrete points in time n (denoted by $x[n]$, where n is an integer value).

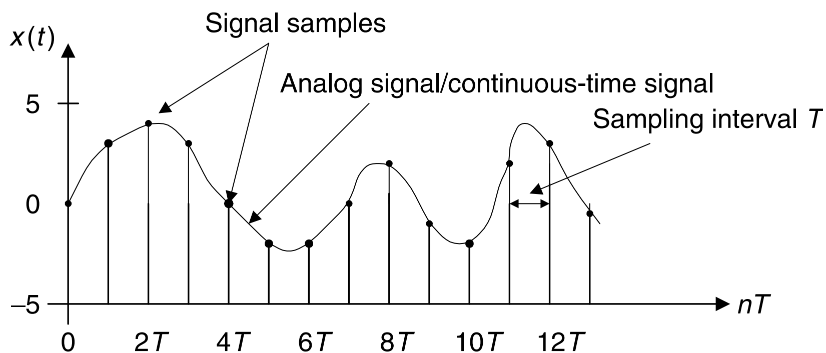


Figure 7.4: Display of the analog (continuous) signal and display of digital samples versus the sampling time instants [92].

Computers or Digital Signal Processors (DSPs) typically work with discrete-time signals. This is due to the fact that continuous-time signals contain infinite number of points. Infinite number of points are not appropriate to be processed by DSPs or computers, since they would require an infinite number of memory and processing power for computation [32]. As shown in Figure 7.4, discrete-time signals result from the sampling of continuous-time signals, but there are also signals that are discrete by nature like the stock market for instance.

Since there exists one amplitude level for each sampling interval, each sample amplitude level can be sketched at its corresponding sampling time instant (see Figure 7.4) where 14 samples at their sampling time instants are plotted, each using a vertical bar with a solid circle at its top.

Moreover, the dynamics of continuous-time signals is represented using differential equations (derivatives and integrals). The analysis of the dynamic of such continuous-time signals is done using the Laplace transforms, if they are Linear Time Invariant (LTI)[3]. Furthermore, to do frequency domain analysis Fourier Transforms (FT) is typically used. Whereas, the dynamic of discrete time signals is represented by means of difference equations (differences and sums); analysis is done using Z-transforms, and finally Discrete time Fourier Transforms (DFT) is typically used for frequency domain analysis.

The choice of the domain (time or frequency) for signal analysis depends on the type of application context. For example, looking at continuous-time signals, both Laplace transforms and FT are very closely related. Laplace transforms are more suitable for the analysis of control systems, while FTs are more suitable for communication systems where signal frequency values are important.

For frequency domain analysis, the representation of a digital discrete signal is given in terms of its frequency components. For this purpose, the signal spectrum needs to be developed. The algorithm transforming time domain signal samples into the frequency domain components is known as DFT. The DFT establishes a relationship between the time domain representation and the frequency domain representation [75].

As shown in Figure 7.5, spectral plots are more adequate to display frequency information of digital signals. The figure illustrates the time domain representation of a sinusoid characterized by 32 samples and sampled at a rate of 8000Hz; the bottom plot shows the frequency domain representation (signal spectrum). The signal spectrum clearly shows that the amplitude peak is located at the frequency of 1000Hz in the calculated spectrum.

[3]LTI systems can be specified in terms of its impulse, transfer function, or frequency response [75].

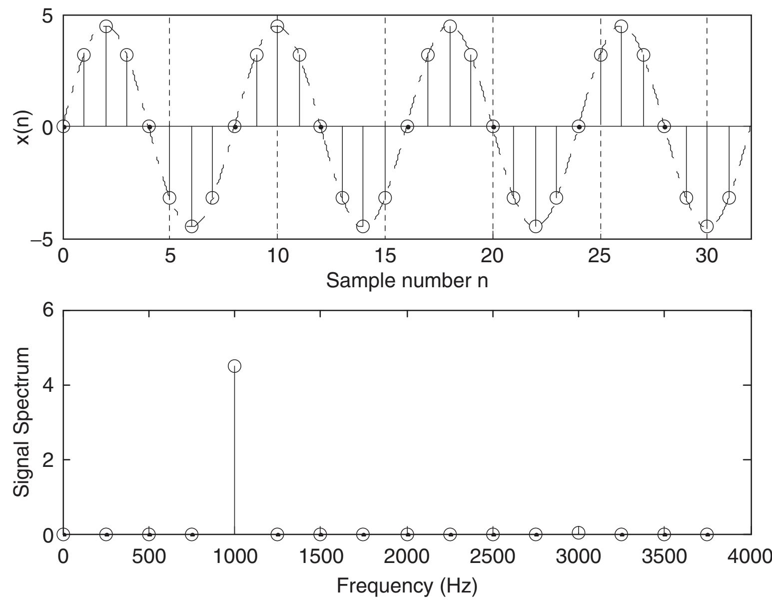


Figure 7.5: Example of the digital signal and its amplitude spectrum [92]. The signal is sampled at a rate of 8000Hz.

Multirate systems are complex systems composed of several interconnected sub-components operating with different sampling rates (sampling frequencies). The sampling rate defines the number of samples per unit of time (usually seconds). Speech and audio processing are typical application areas. To keep the system synchronized, rising or lowering of the sample rates is required. Basically, the sampling rate of a signal can be manipulated in three different ways [32]:

- Decimation/Downsampling: reduction of the sampling rate by an integer factor M.
- Interpolation/Upsampling: increase of the sampling rate by an integer factor L.
- Resampling: changing the sampling rate by a non-integer factor (L/M). This is a combination of both decimation and interpolation, and describes the interpolation with factor L followed by a decimation with Factor M.

7.1.3 Downsampling

Downsampling refers to the reduction of a sequence of data by an integer factor $M > 0$ as shown in Equation 7.4 [92].

$$y(m) = x(mM). \tag{7.4}$$

Equation 7.4 denotes the downsampling of a data sequence $x(n)$ by an integer factor M, where $y(m)$ denotes the downsampled sequence resulting from the selection of every M-th data point (sample). This is illustrated in the following example.

Let's consider an original data sequence $x(n): 9, 8, 5, 9, 10, 7, 5, 3, -1, -4, -6, -6, -5, -3, \cdots$ with a sampling period $T = 0.1$ seconds (sampling rate is $f_s = \frac{1}{T} = 10$ samples per second) and a downsampling factor of $M = 3$.

Hence, the resulting downsampled sequence is $y(m) = 9, 9, 5, -4, -5, \cdots$ with a corresponding sampling period of $T = 3 * 0.1 = 0.3$ seconds (sampling rate of 3.33 samples per second).

This rather trivial example is adequate to illustrate the basic idea behind downsamling. However, the reduction of the sampling frequency may lead to the distortion of the original signal in the time domain and to an overlapping of the signal spectrum in the frequency domain. This is referred to as *Aliasing* [66]. To overcome this problem, the conditions stated in Formula 7.5 must be fulfilled.

$$f_s \geq 2 \cdot f_{signal}. \tag{7.5}$$

Formula 7.5 is the *Nyquist sampling criterion* derived from the Nyquist-Shannon sampling theorem [66]. The theorem basically states that for a uniformly sampled system, the original signal can perfectly be recovered if the sampling frequency is at least twice as large as the highest frequency component of the original signal to be sampled [92].

After the downsampling operation, the new sampling frequency is

$$f_{sM} = \frac{1}{MT} = \frac{f_s}{M}, \tag{7.6}$$

where f_s denotes the original sampling rate.

Hence, the *Nyquist frequency* [4] is defined as follows:

$$\frac{f_{sM}}{2} = \frac{f_s}{2M}. \tag{7.7}$$

[4] The Nyquist frequency is named after the electronic engineer Harry Nyquist [18] and is sometimes referred to as the *folding frequency* of a sampling system.

Formula 7.7 states that the Nyquist frequency is half of the sampling rate of a discrete signal processing system. Therefore, to avoid aliasing noise into the downsampled data sequence, the following equation should hold

$$f_{max} < \frac{f_{sM}}{2}, \tag{7.8}$$

where f_{max} denotes the highest frequency component of the original signal in the frequency domain.

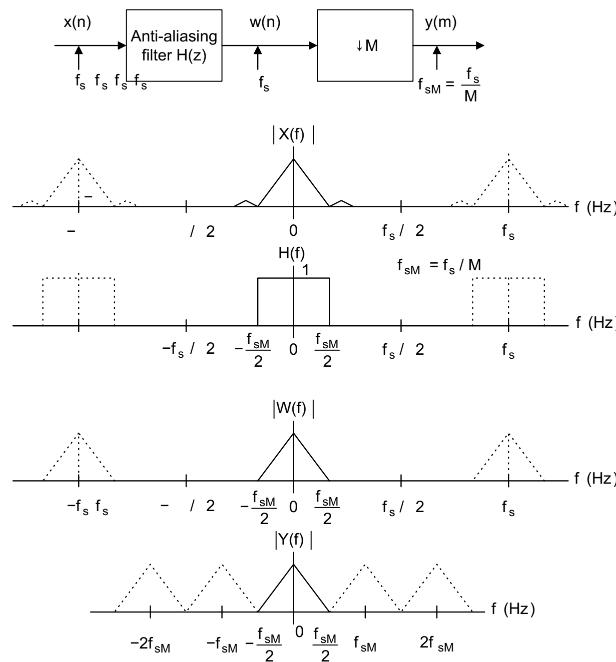


Figure 7.6: Frequency domain: spectrum after downsampling [92]

Figure 7.6 depicts a general block diagram for a typical downsampling process. As shown in the figure, the original signal $x(n)$ is first processed by an anti-aliasing filter H(z) before downsampling. Typically, an anti-aliasing filter is a Low-pass filter that passes low-frequency signals and reduces the amplitude of signals with higher frequencies than the cutoff frequency[5]. The filtered output can be written as

$$W(z) = H(z)X(z). \tag{7.9}$$

Where $X(z)$ is the z-transform of the original data sequence $x(n)$, and $H(z)$ is the Low-pass filter transfer function. After anti-aliasing filtering, the downsampled signal $y(m)$ takes its value from the filter output as:

$$y(m) = w(mM). \tag{7.10}$$

[5] A cutoff frequency is a boundary in a system's frequency response at which energy flowing through the system begins to be reduced rather than passing through.

The corresponding spectral plots for $x(n)$(|X(f)|), $w(n)(|W(f)|)$ and $y(m)(|Y(f)|)$ and $H(z)$ (H(f)) are shown on the right hand side of the figure. Basically, H(f) filters out all frequencies above $\frac{f_{sM}}{2}$.

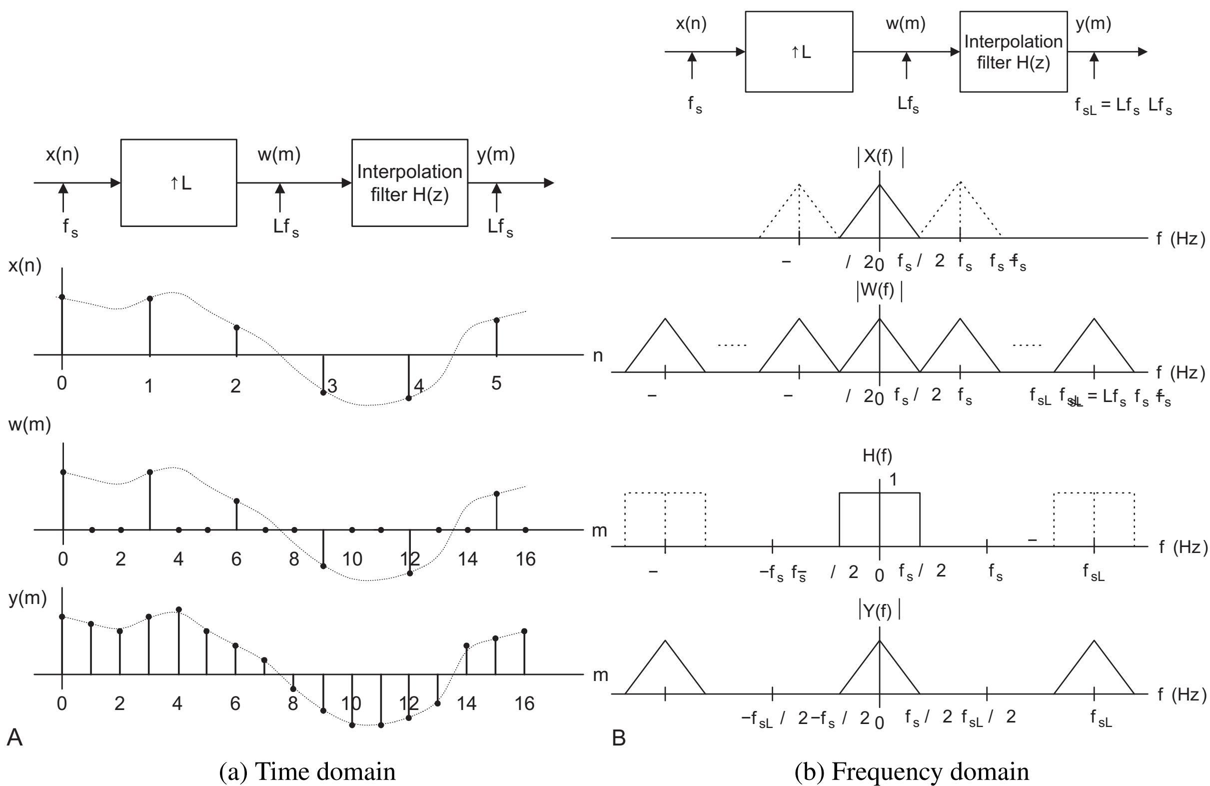
Interpolation, Upsampling

Upsampling describes the process of increasing the sampling rate by a positive integer factor L [92]. This process is typically described as follows:

$$y(m) = \begin{cases} x(\frac{m}{L}), & m = nL \\ 0, & \text{otherwise} \end{cases} \tag{7.11}$$

where $n = 0, 1, 2, \cdots$. x(n) is the data sequence to be upsampled by a factor of L, and $y(m)$ is the upsampled sequence. The basic scheme behind upsampling is illustrated in Figure 7.7.

As an example, let $x(n) : 8, 8, 4, -5, -6, \cdots$ and $L = 3$. Upsampling is done by adding $L-1$ zeros between the samples of the original data sequence. The resulting upsampled sequence is: $w(m) : 8, 0, 0, 8, 0, 0, 4, 0, 0, -5, 0, 0, -6, 0, 0, \cdots$. Afterward, the upsampled signal $w(m)$ is smoothed using an interpolation filter.



(a) Time domain

(b) Frequency domain

Figure 7.7: Interpolation [92]

Analogously to downsampling, assuming that the data sequence has a current sampling period T, the Nyquist frequency is given by $f_{max} = \frac{f_s}{2}$. The new sampling period is $\frac{T}{L}$, and the new sampling frequency is changed to:

$$f_{sL} = Lf_s. \tag{7.12}$$

The upsampling operation causes the introduction of unwanted spectral replicas in the frequency range from 0 Hz to the new Nyquist limit ($Lf_s = 2$ Hz). Those included spectral replicas are then removed using the interpolation filter. To remove those included spectral replicas, an interpolation filter with a stop frequency edge of $f_s = 2$ Hz must be attached, where the normalized stop frequency edge is given by

$$\Omega_{stop} = 2\pi \left(\frac{f_s}{2}\right) * \left(\frac{T}{L}\right) = \frac{\pi}{L} radians. \tag{7.13}$$

As shown on the right hand side of Figure 7.7, the desired spectrum for $y(n)$ is obtained after the filtering process. For further details see [92].

7.2 Our synchronization approach

Synchronization is fundamental for a wide range of applications. It refers to two distinct, but related concepts: synchronization of processes, and synchronization of data. Process synchronization refers to the general idea that multiple processes are to join or handshake at a certain point in time.

Data synchronization, on the other hand, describes the process of establishing data consistency between a source and a target process and the continuous harmonization of data over time. Since data is collected at different points in time and possibly from different sources, accurate reconstruction of data is necessary to ensure its integrity.

The synchronization approach presented in this thesis makes use of techniques applied in the field of Multirate systems described in Section 7.1.2.

As discussed in Chapter 2, electronic systems can be organized into three main categories: control, measurement and communication. Control systems are further subdivided into open-loop control and closed-loop control.

One application of the developed synchronization approach is depicted in Figure 7.8, it mainly targets Restbus simulation of measurement and control applications. This is particularly suitable for realistic situations where the Restbus simulator has a lower data processing rate than the bus network. In that case, at run-time, the bus network will generate much more data than the

simulator can process (*Undersampling*) and inversely the simulator will not produce network data fast enough (*Oversampling*).

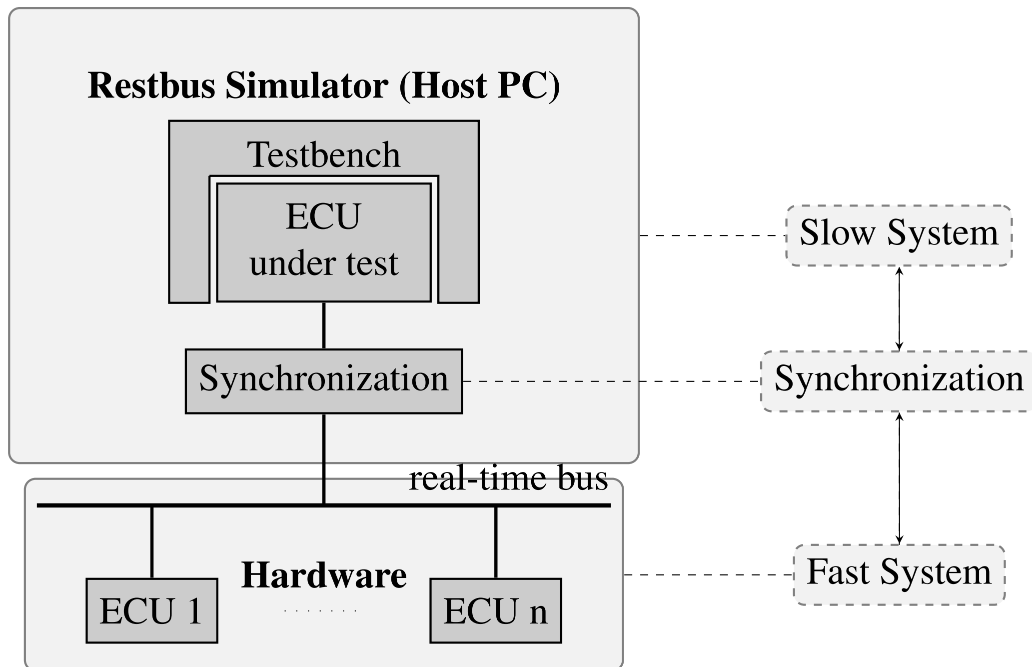


Figure 7.8: Application scenario of the synchronization algorithm

Our synchronization mechanism ensures a continuous communication between the Restbus simulator (RBS) and the bus network by dynamically adapting to the variation of the data processing speed of the simulated model.

The approach makes use of a special downsampling method during which special values such as peaks are detected. Peaks can be very important for the simulation of systems like steer-by-wire, where a peak represents the maximum steering angle, or an active damping system where a peak value can result from the shock of a wheel on a ramp. In those cases they are not outliers but specific values that need to be kept during our synchronization procedure.

The detected values can then be forwarded to the Restbus simulator. Furthermore, only values that do not affect the waveform of a data sequence are suppressed. Since the data exchanged between the Restbus simulator and the bus network is only used for functional testing or validation, the slight distortion of the waveform of the original data sequence resulting from the downsampling procedure is neglected. This distortion is caused by the violation of the Shannon-Nyquist sampling theorem discussed in Section 7.1.2.

To increase the data processing rate of the Restbus simulator (upsampling), our approach uses a simple interpolation technique, that consists of repeating

the last value produced by the Restbus simulator. More complex interpolation techniques could also be applied, especially when the behavioral characteristics of the simulated components is given [32].

In the following sections, a detailed description of the Up- and Downsampling procedures will be given.

7.3 Upsampling

Figures 7.9 and 7.10 illustrate the oversampling problem. The example in Figure 7.9 shows two communicating components, a sender and a receiver where T_R denotes the sampling period of the receiver, and T_S the data transmission rate of the sender. Here, the sender is assumed to have a lower data transmission rate than the sampling period of the receiver (i.e.: $T_S \leq T_R$). As shown in Figure 7.10 the receiver will not always get valid data.

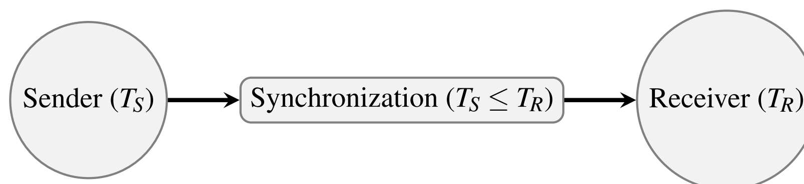


Figure 7.9: Synchronization problem, Receiver faster than sender

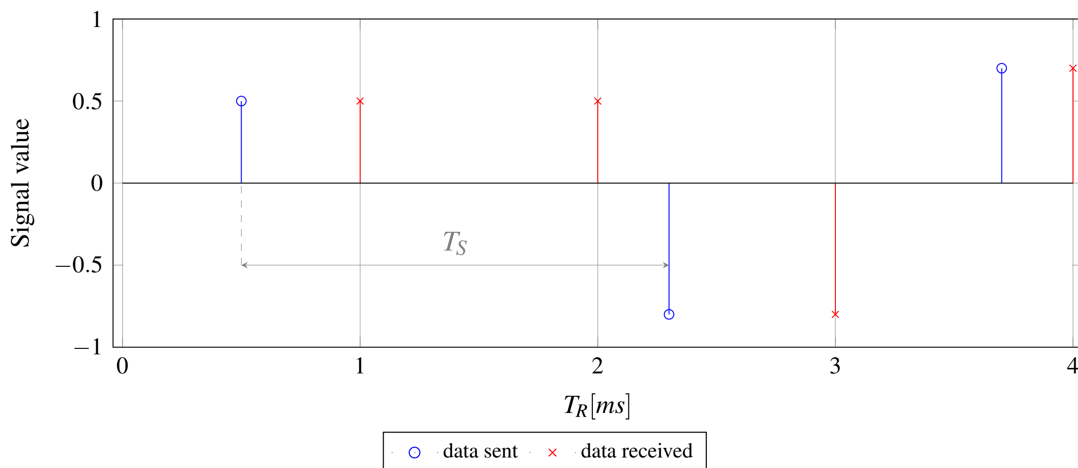


Figure 7.10: Problem: missing data at time 2 ms during communication between a slow sender and a fast receiver

A possible approach is to fill the missing values by mean of interpolation. The simplest interpolation technique is to repeat the last valid value. This is depicted in Figure 7.11. Complex interpolation functions can also be used if the behavior of the sender (simulated node) is known. Based on that behavior future values can be predicted.

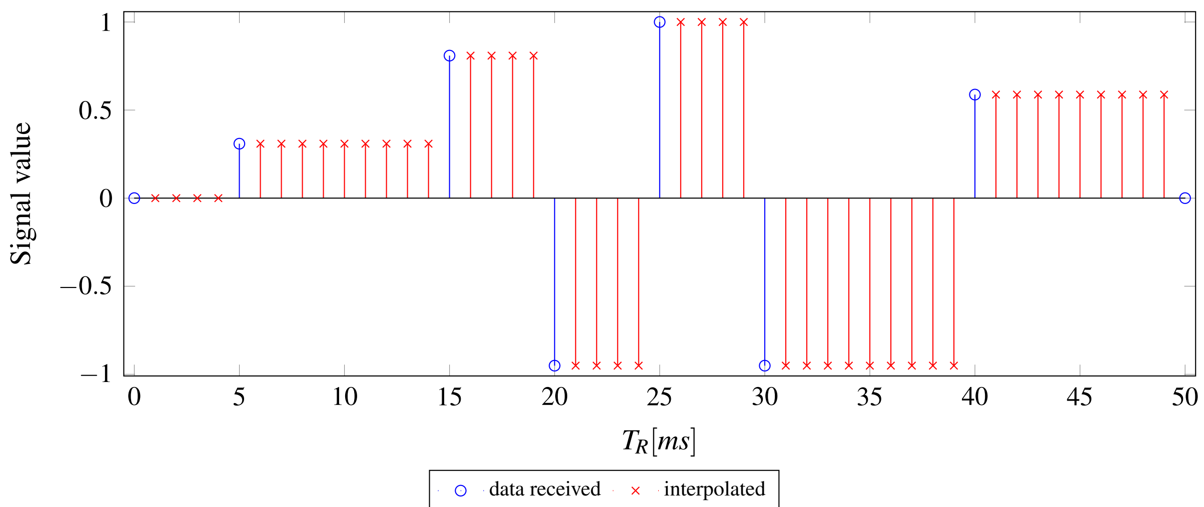


Figure 7.11: Interpolation: repetition of last valid value received

7.4 Downsampling

Figure 7.12 shows the structure of the buffer used for the synchronization process. As depicted in the figure, the buffer consists of a three level FIFO-based queuing mechanism. After the initialization phase of the buffer, incoming date migrate from the top to the bottom, that is, from the *receive-Queue* down to the *SendQueue* over the *ProcessQueue*. Both *ProcessQueue* and *SendQueue* have the same fixed-size, whereas the *ReceiveQueue* has a dynamically growing size.

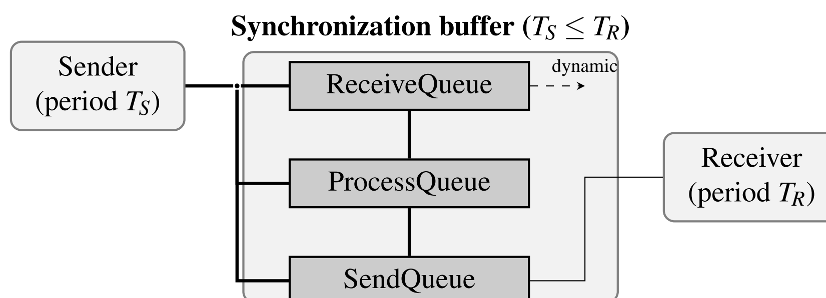


Figure 7.12: Synchronization buffer: 3 level queuing mechanism sender faster than receiver.

A buffer is instantiated for each data exchanged between sender and receiver. Complex data structures combining more than one data can also be built for more efficiency. The synchronization process itself is triggered each time data is received from the sender; in the case of Restbus simulation, at the end of each bus communication cycle. The procedure is subdivided into an initialization and a main phase where each phase consists of several steps.

7.4.1 Main phase

As depicted in Figure 7.13, during the main phase of the synchronization process, incoming data is always inserted into the ReceiveQueue. The ProcessQueue is only used in the main phase as an intermediate buffer between the ReceiveQueue and the SendQueue, in order to maintain a seamless fluid communication between the sender and the receiver. Thus, the main phase of the synchronization process is structured as follows:

Step1 After the newly received data has been stored into the ReceiveQueue, try to send oldest data from the SendQueue to the receiver. If the SendQueue is empty then go immediately to Step2. If the receiver is not ready, wait for the next communication cycle and try sending the oldest data again. This Step is depicted in Figure 7.13

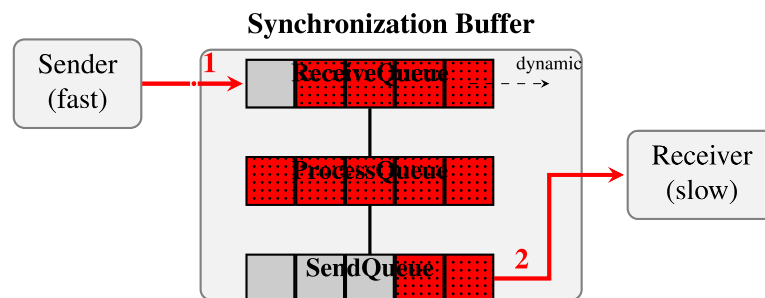
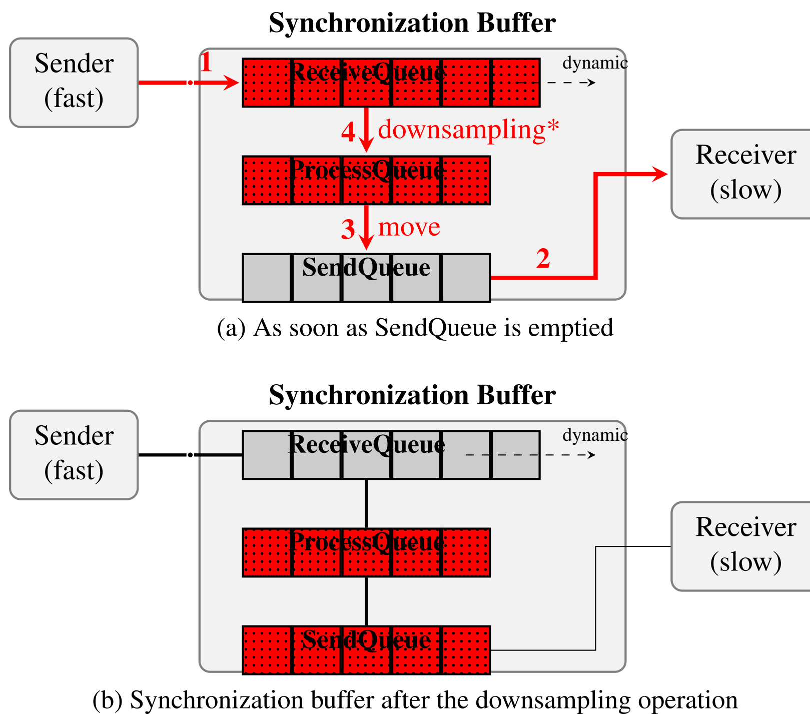


Figure 7.13: Incoming data are inserted into the ReceiveQueue (1). Data are taken form the SendQueue and sent to the receiver based on the FIFO principle (2)

Step2 Move content of the ProcessQueue to the SendQueue, then go immediately to Step3 (see Figure 7.14a (3)).

Step3 As depicted in Figure 7.14a, after the the content of the ProcessQueue has been moved (3), start the downsampling procedure on the content of the ReceiveQueue (4), then go to Step1 and wait for the next communication cycle. The downsampling process runs in a parallel process.

The result of the downsampling procedure in Step3 is stored into the ProcessQueue. Moreover, the downsampling process does not have to complete within the same synchronization step, since it runs as a separate process. However, it should yield before the SendQueue gets empty in order to maintain a seamless continuous communication. If this happens, the downsampled data sequence is discarded and the SendQueue is filled with the content of the ReceiveQueue based on the FIFO principle.



(a) As soon as SendQueue is emptied

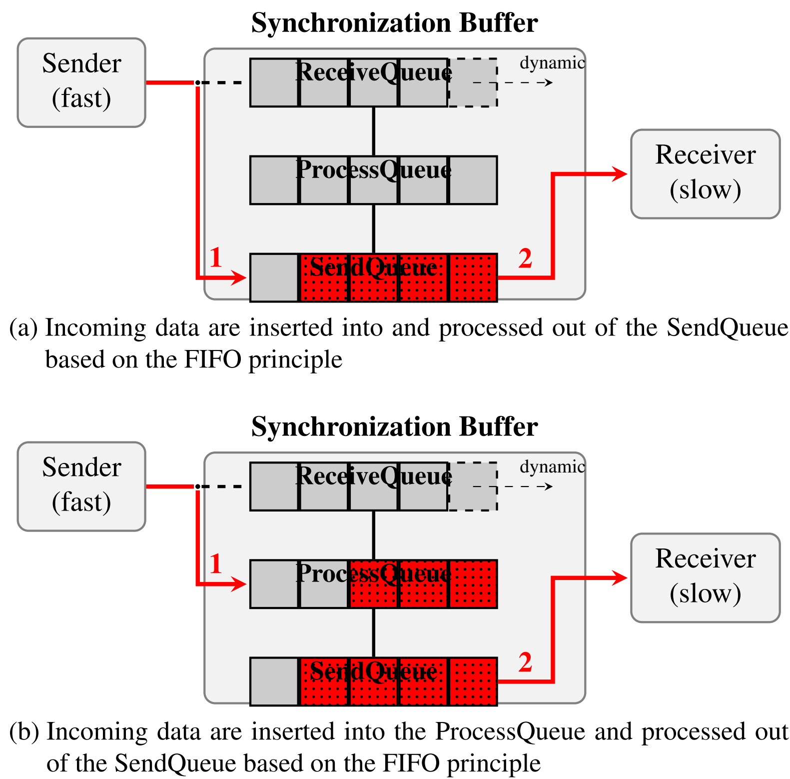
(b) Synchronization buffer after the downsampling operation

Figure 7.14: Downsampling during the main phase of the synchronization process

7.4.2 Initialization phase

Figure 7.15a illustrates the first step of the initialization phase. Each time new data is received from the sending node, the synchronization process first tries to forward it to the receiving node. In case of failure the data is stored into the SendQueue, which is a FIFO queue. This goes on until the SendQueue becomes full. Afterward, incoming data will are inserted into the ProcessQueue (as depicted in Figure 7.15b). After completion of the initialization phase, the ProcessQueue will only be used as an intermediate buffer between the ReceiveQueue and SendQueue at run-time and as target for the downsampling process. This enables a continuous communication between sending and receiving nodes.

The initialization phase of the synchronization process ends as soon as the ProcessQueue becomes full (Figure 7.16).



(a) Incoming data are inserted into and processed out of the SendQueue based on the FIFO principle

(b) Incoming data are inserted into the ProcessQueue and processed out of the SendQueue based on the FIFO principle

Figure 7.15: Initialization phase of the synchronization process

7.4.3 Downsampling with peak detection

Downsampling

This approach targets the synchronization between two nodes with different data processing rates. Further, our synchronization approach was particularly developed for situations where the sending node operates at a higher frequency than the receiving node.

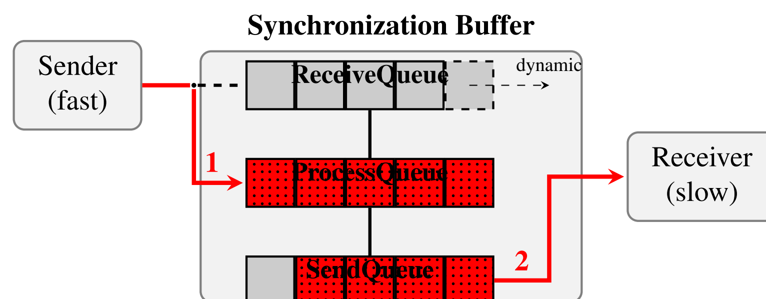


Figure 7.16: End of the initialization phase, ProcessQueue is full

We use a special downsampling approach that favors special values during the data reduction process. In our case peaks are favored. Nonetheless other selection criteria can also be specified. For instance, the *age* of the data might be specified if only the most recent values are relevant.

As discussed in Section 7.1.3, downsampling describes the process of reducing a larger sequence of data $N_1^n = \{y_1, \ldots, y_n\}$ containing n data points to a smaller data sequence $\check{N}_1^m = \{\check{y}_1, \ldots, \check{y}_m\}$ containing m data points with $m < n$. To achieve this, the data sequence N_1^n is subdivided into m equal intervals of the length $l = \frac{n-1}{m}$. For $1 \leq i < m$, each data point $\check{y}_i$ is chosen out of the i-th interval based on the following rule:

$$\check{y}_i = \begin{cases} \hat{n}_i & \text{, if peak } \hat{n} \text{ in interval i} \\ \bar{y} = \frac{1}{l} \sum y_j \in N_{\lceil i \rceil}^{\lfloor (i+l) \rfloor} & \text{, else.} \end{cases} \tag{7.14}$$

Formula 7.14 means that peak values are preferred, otherwise the mean value is computed in the interval. The following section will discuss the process of peak detection.

Peak detection

As discussed in Section 7.1, robust data smoothing methods are typically

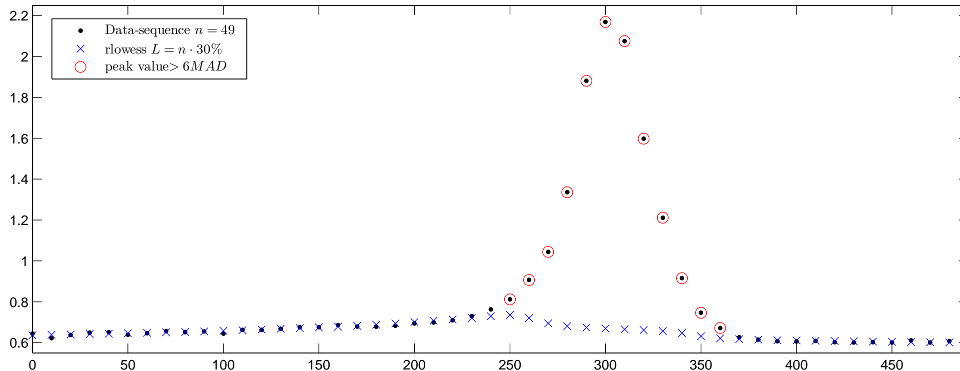


Figure 7.17: Outliers are filtered out during the data smoothing process

applied to reconstruct the original signal in noisy environments. They are robust against outliers and can filter out outliers during the data smoothing process. Figure 7.17 illustrates the result of such a data smoothing process. The smoothed data sequence is represented using crosses and the original sequence is represented by points. Peak values are emphasized using circles.

In the context of this thesis, we deal with discrete-time signals, since the discretization of the continuous-time signal provided by the network bus has already been conducted by the bus communication controller. Thus, we can

assume that the input data sequence is not noisy. For peak value detection we make use of the smoothing method Robust LOWESS, that will usually handle peak values as outliers. The aim here is to keep those values during the downsampling process in order to have a better approximation of the original data sequence.

Hence, before starting the actual data reduction process, the input data set is smoothed. After completion of the smoothing procedure, the difference D_1^n between the original sequence of data points $N_1^n = y_1, \ldots, y_n$ and the robust-smoothed data sequence $\tilde{N}_1^n = \tilde{y}_1, \ldots, \tilde{y}_n$. is built as follows:

$$D_1^n = N_1^n - \tilde{N}_1^n = \begin{pmatrix} \varepsilon_1 = y_1 - \tilde{y_1} \\ \vdots \\ \varepsilon_n = y_n - \tilde{y_n} \end{pmatrix}, \text{ n: number of data points.} \qquad (7.15)$$

A data point $\hat{n}_i$ is detected as peak value if the following formula holds

$$\hat{n}_i = y_i \text{ if } |\varepsilon_i| > threshold, \text{ with } i = 1, \cdots, n, \qquad (7.16)$$

where $threshold = 6 \cdot MAD$. $|\varepsilon_i|$ denotes the absolute value of the residual ε_i. The threshold value used here is the same value used by Robust LOWESS during the computation of the robust weights (see Equation 7.2). This enables the recognition of the suppressed data points.

As aforementioned, the downsampling process is applied on the data sequence contained by the ReceiveQueue and is triggered, when the SendQueue gets empty. This process does not have to yeild within the same bus communication cycle, since the content of the ProcessQueue has been moved to the SendQueue before starting the process. However, in order to keep a seamless continuous communication, it should terminate before the SendQueue gets empty again. This would otherwise cause a distortion of the overall signal path.

Different factors might influence the quality of the peak detection process. In addition to the choice of the right smoothing parameter, which is required by the smoothing method, the size of the queues in the synchronization buffer also has an impact on both the duration of the downsampling process and the quality of the peak detection algorithm.

As aforementioned, the ReceiveQueue has a dynamically growing size (denoted by $size(RQ)$), but due the limited amount of memory available on the simulation PC this size should be minimized.

SendQueue and ProcessQueue have an equal fixed maximum capacity. Choosing the right size for the SendQueue ($size(SQ)$) and ProcessQueue ($size(PQ)$) is of great importance, because on the one hand a small queue size would

lead to a situation where the downsampling process will be triggered too often, and on the other hand choosing a large queue size would increase the duration of the downsampling process.

Let T_S and T_R be the respective sampling periods of the sender (bus network) and the receiver (Restbus simulator), with $T_S < T_R$. However, since the data processing speed (sampling period) of the Restbus simulator is assumed to be unknown, and the correlation between the size of the Synchronization buffer and the sampling period ratio cannot be mathematically determined, the choice of the right parameter con only be experimentally determined by means of empirical analysis.

The evaluation of the synchronization approach will be discussed in Chapter 8.

7.5 Summary

In this chapter, we have presented a synchronization approach that handles possible oversampling and undersampling issues that may occur during the communication between an event-based simulator and a network bus. Furthermore, the proposed synchronization approach is dedicated but not limited to early validation of open-loop control systems.

As depicted in Figure 4.4, our synchronization approach enables the connection of a SystemC simulation model to a real bus network or a HIL test platform. Therefore, the approach provides an infrastructure for early validation of new functionality.

In this thesis we focused on the worst-case scenario where the Restbus simulator has a lower data processing rate than the hardware bus. This might be the case for the simulation of complex SystemC models. We use a special downsampling approach that favors specific values during the data reduction process in case of oversampling. In the scope of this thesis, peak values were favored in order to achieve a good approximation of the input signal after downsampling. Nonetheless other selection criteria can also be specified.

As introduced in Section 7.4.3, our downsampling algorithm incorporates the data smoothing technique Robust LOWESS [25] for the detection of peak sequences. Robust LOWESS is a popular data smoothing technique that works well on large, and densely sampled data sets. Furthermore, the technique is robust against outliers.

Since the input signal received from the bus network is always preprocessed by the bus network communication controller, the resulting data sequence can be assumed to be noiseless. Therefore, in order to be able to detect peak

sequences, the data smoothing algorithm is configured in such a way that those peak values are detected as outliers. After their detection they are kept instead of being suppressed. In case there are no peak values in the data sequence, the mean value is built.

The presented approach has some limitations. Due to the timing restrictions, this strongly depends on the complexity of the SystemC model being simulated on the host-PC. The choice of the configuration parameters for the synchronization buffer also plays an important role. This key configuration factors includes the choice of the queue sizes in the synchronization buffer and the configuration of the data smoothing algorithm.

As already mentioned, since the data processing rate (sampling period) of the Restbus simulator cannot be determined in most of the cases, and the correlation between the size of the synchronization buffer and the sampling period ratio cannot be mathematically determined, the choice of the right parameter needs to be experimentally investigated by means of empirical analysis. The evaluation of the approach will be done in Chapter 8.

Provided that the size of the synchronization buffer has been correctly set, the three buffer synchronization approach enables a seamless fluid communication since the processing queue is only used as intermediate buffer.

Concerning the data processing rate ratio, since our goal is only to approximate the input signal, the violation of the Nyquist-Shannon can be neglected.

Chapter 8

Evaluation of Synchronization approach

This chapter demonstrates the application of our synchronization approach described in Chapter 7. The approach was validated on a HIL test environment consisting of a Steer-By-Wire system. This test environment has been used in several projects [90, 49].

The chapter is structured as follows: first an overview of the test platform will be given followed by the presentation of evaluation results. The evaluation includes the investigation of the impact of the size of synchronization buffer on communication delay during run-time simulation, and the impact of different configurations of the synchronization algorithm on peak sequence detection. Furthermore, the influence of the sampling rate ratio between sender and receiver on signal distortion is also investigated.

8.1 Evaluation platform

8.1.1 System Overview

The evaluation platform is composed of a Steer-By-Wire system as real-world application with hard real-time constraints, and a Restbus simulator implemented in SystemC running on a PC. During Restbus simulation, bus messages are exchanged between the physical network nodes and the virtual ones (i.e., simulated nodes). In such a simulation environment, one or more simulation PCs can be connected to the bus network.

The architecture of the overall system can be seen in Figure 8.1. The Restbus simulator communicates with the physical network through a FlexRay

device. Figure 8.1 also depicts the two main parts of the Restbus simulator, which are of the SystemC simulator, that simulates the components under investigation and the the so-called *Adapter* component. The Adapter component executes tasks like accessing the communication controller (CC) (here FlexRay PCI Card) using the corresponding CC device driver.

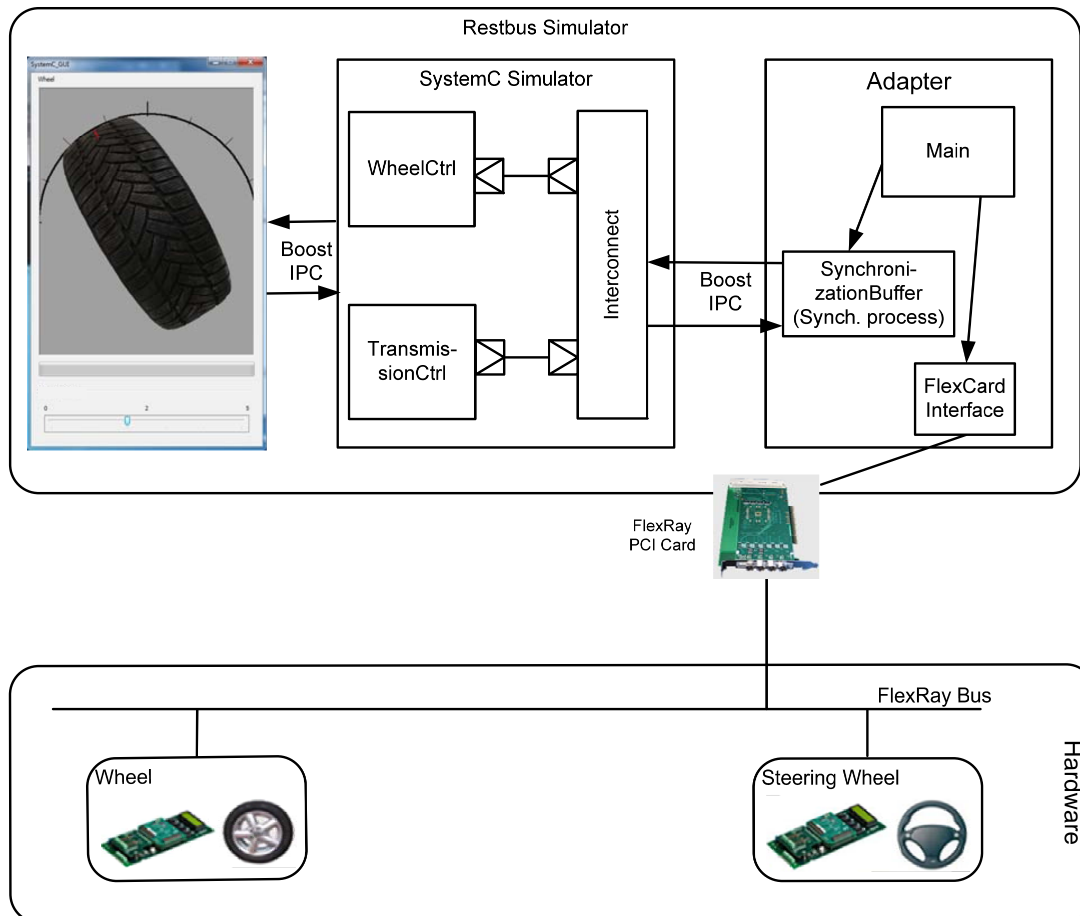


Figure 8.1: Overview of the steer-by-wire architrecture

8.1.2 Hardware architecture

Steer-By-Wire System

Figure 8.2 shows the platform used for the evaluation of the synchronization approach. The Steer-By-Wire hardware is composed of two main hardware components. The first component is an active steering wheel from Stirling Dynamics [35] (see Figure 8.3). The steering wheel is equipped with sensors and an electric engine as an actuator, which provides the realistic simulation and measurement of forces and values like torque, velocity, and rotation

Figure 8.2: Test environment

angle. The communication of the steering wheel with other components is realized by a CAN interface. The second component is a steering and damping test bed (see Figure 8.4) composed of a ramp with a tire and axle and an active damping system. There are two electrical actuators that realize the steering and the active suspension and a load cell to measure feedback forces of the wheel. For communication, the actuators have a CAN Interface and analog I/O where the load cell provides analogue values.

Figure 8.3: Active steering wheel

Figure 8.4: Wheel and damping testbed

Restbus simulator

The Restbus simulator runs on a standard host PC. For network bus access, the PC is equipped with a FlexRay communication controller FlexCard PM-C/PCI card from Eberspaecher Electronics [37] (Figure 8.1). Furthermore, the host PC used for the Restbus simulation is equipped with a Pentium 4 3GHz processor with 1GB of RAM and a Linux kernel 2.6.24.2, with Debian 4.0 as operating system.

8.1.3 Software architecture

Restbus simulator

The main focus of our Restbus simulation process lies on functional verification. Timing verification is only done on the pure SystemC simulation model, which also includes the simulation of the real nodes. As discussed in Chapter 4, this verification step is done to make sure that the simulated ECU-nodes are extracted from a consistent SystemC design model from the timing perspective.

For online simulation, the simulated ECU-nodes are functional SystemC models implemented at the TLM abstraction level. Therefore, used TLM models were used to speed up the SystemC simulation process. In fact, we focused

on the simulation of SW-Cs and made used the SystemC features for ECU hardware abstraction.

In general, the behaviour of a SW-C under test can either be handwritten or generated using a behavioural modeling tool like Matlab/Simulink [68], where the generated C-code is integrated in SystemC using SystemC modules [53]. Communication between modules of our SystemC simulator is realized with TLM 2.0 library. As introduced in Chapter 2, TLM 2.0 is part of the SystemC IEEE standard [56] . It provides two ways for timing modeling (coding styles): loosely-time and approximately-time. These coding styles offer different level of accuracy for timing modeling and simulation. The choice of the corresponding timing model depends on the desired accuracy. The modules *Interconnect* and the *Adapter* play the role of the middleware and basically abstract the AUTOSAR basic software (BSW) [28] and Runtime Environment (RTE) [30] layers. Therefore, the inputs required for our Restbus simulator design process are the implementation code of the SW-Cs and the network bus configuration data including communication matrix.

8.1.4 Applied tools

The tools applied for the design of the Steer-By-Wire system were TTX-Plan and TTX- Build from TTTech Automotive GmbH. The cluster design tool TTX-Plan was applied to generate communication schedules based on communication requirements. These communication requirements, whereas TTX-Build was used for the generation of the AUTOSAR communication stacks.

For the integration of the Restbus simulator into the existing FlexRay network, the tool FlexConfig Developer from Eberspaecher Electronics [38] was applied. The tools Eclipse CDT was used as an integrated development environment in combination with the SystemC simulation kernel reference library Version 2.2. Alternatively, the tools Microsoft Visual C++ or QuestaSim could also be used. The synthesis tools TargetLink from dSpace or Matlab/Simulink /Real-Time-Workshop suite from MathWorks can be applied for the generation of the software components behaviour from the Simulink model.

8.2 Evaluation results

This section presents the evaluation results of our synchronization approach. Hereby, we focused on the application scenario, where the simulated component performs a monitoring task. Further, we considered a situation where

the Restbus simulator has a lower data processing rate than the FlexRay bus network.

As discussed in Chapter 7, our synchronization algorithm incorporates a data smoothing step. Therefore, the choice of the configuration parameters for the data smoothing process has a direct impact on the performance and the quality of the synchronization process. Furthermore, the size of queues in the synchronization buffer also influence the quality of the downsampled input signal.

Therefore, both the size of the synchronization buffer and the configuration of the data smoothing process were identified as the key factors that have a strong impact on the synchronization process. Since the SendQueue and ProcessQueue of the synchronization buffer are both of the same size, it follows that the impact of the parameter $size(SQ)$ on the synchronization process needs to be investigated. Additionally, the smoothing parameter (L) of the data smoothing process LOcally WEighted Scatterplot Smoothing (LOWESS) will also be investigated.

For a good approximation of the discretized input signal, data points with high amplitude should to be detected and preserved during the downsampling process. Our Synchronization algorithm is configured in such a way that those specific data points are identified as peaks. In this chapter we investigated the impact of the data processing rates ratio on peak detection. Moreover, the impact of the aforementioned parameters on transmission delay during run-time simulation was also investigated.

In order to establish a common basis for the comparison of the different parameter settings during offline simulation, network data provided by the real FlexRay nodes were first recorded online in order to generate simulation traces. Afterward, the recorded data were then transmitted to the Restbus simulator using a stimulus generator as depicted in Figure 8.5.

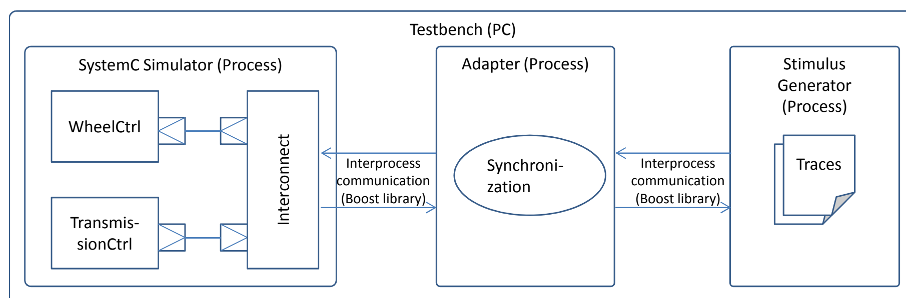
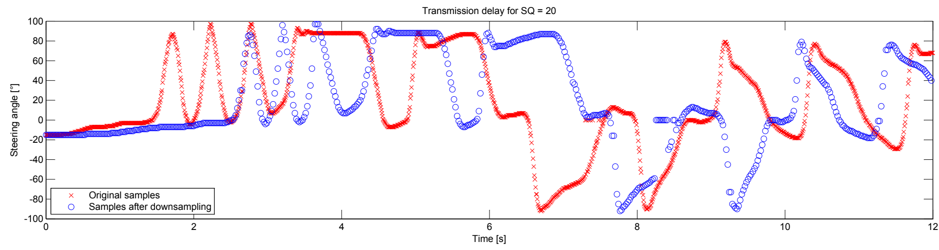
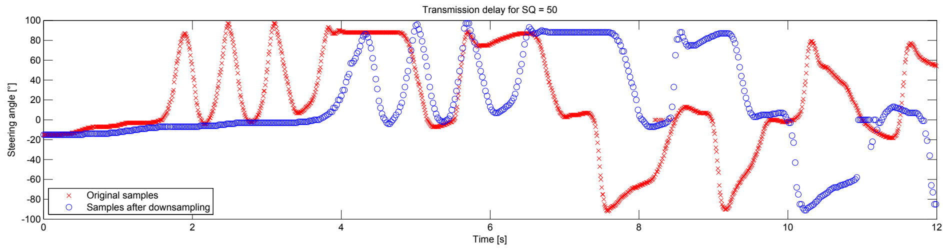


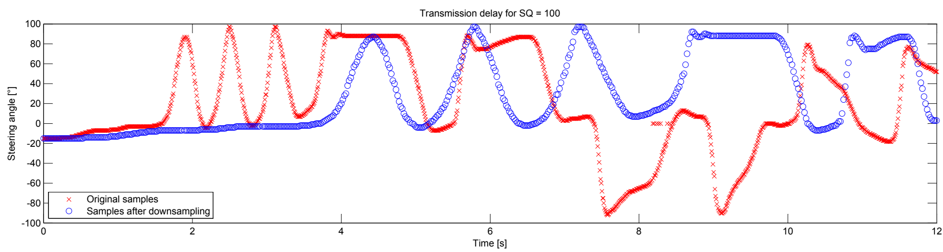
Figure 8.5: Architecture of the testbench used for the evaluation



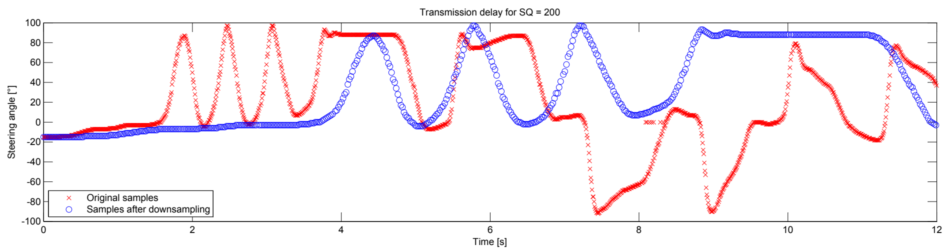
(a) Transmission delay for $size(SQ)_{max} = 20$



(b) Transmission delay for $size(SQ)_{max} = 50$



(c) Transmission delay for $size(SQ)_{max} = 100$



(d) Transmission delay for $size(SQ)_{max} = 200$

Figure 8.6: Delay caused be the downsampling process depending on the SendQueue size

8.2.1 Impact of the SendQueue-size on transmission delay

In this section we present some of the evaluation results regarding the investigation of the impact of the size of the synchronization buffer (i.e.: the size of the SendQueue ($size(SQ)_{max}$)) on the quality of the approximated input signal and on transmission delay. For this purpose, network communication data transmitted by the physical steering wheel of the steer-by-wire system were used. Hereby, with transmission delay we denote the time that elapses between the reception of the network data sent by the bus communication controller (before starting the synchronization process) to the reception of the data by the Restbus simulator (after the synchronization process). Therefore, communication data was recorded within the *Adapter* component and *Interconnect* component, respectively.

The following figures highlight the results of the investigation. They all display an interval of $7s$, where the waveform of the input signal is depicted by "x" symbols, whereas the resulting waveform after the synchronization process is represented by circles. The values on the Y-axis have a range from -90 to 90 degrees, which corresponds to the steering angle of the steering wheel, whereas time was measured on the X-axis in terms of seconds.

As it can be observed in the figures, both the input signal waveform and the resulting signal waveform after downsampling are similar, which is sufficient for early functional validation. This demonstrates the correctness of the downsampling approach. However, the experiments also showed that with increasing SendQueue size the transmission delay also increased (see Figure 8.6d). A better performance of the algorithm could be observed for $size(SQ)_{max} = 20$ (see Figure 8.6a).

8.2.2 Variation of the smoothing parameter of Robust LOWESS

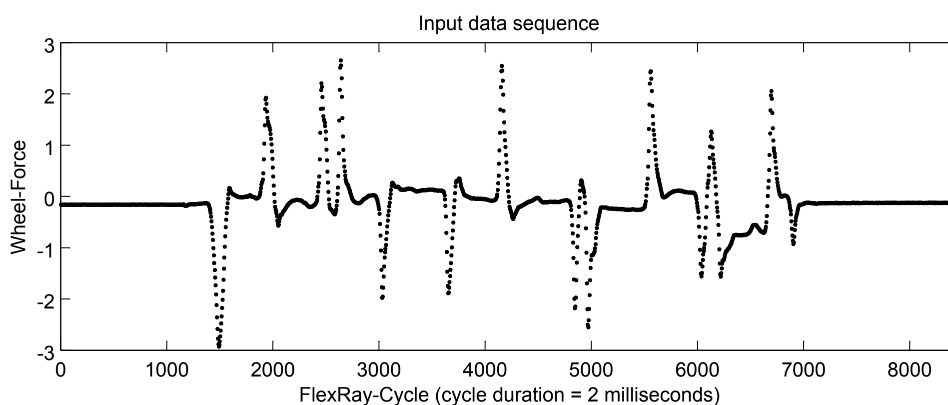


Figure 8.7: Wheel raw data

Figure 8.7 depicts the reference data sequence used for the remaining part of our experiments. The waveform displayed in the figure represents wheel-Force value transmitted by the physical wheel ECU to the physical steering wheel ECU as Force-Feedback information. The data sequence was recorded over a time period of 8451 FlexRay communication cycles with a cycle duration of 2 milliseconds this corresponds to a time window of approximately 17 seconds.

In order to investigate the influence of the smoothing parameters f on peak sequence detection, the maximum capacity of the SendQueue and ProcessingQueue was set to $size(SQ)_{max} = size(PQ)_{max} = 10$. Additionally, the sampling period of the Restbus simulator was set to $T_R = 150$ milliseconds. The results of the experiments are shown in the Figures 8.8a 8.8b 8.8c.

Three different smoothing parameter configurations were investigated: $f = 0,1$, $f = 0,3$, $f = 0,6$. This choice of these parameters was based on the fact that $f = 0,3$, is the configuration typically recommended in the literature [71]. Looking at the results of these experiments we could not draw any conclusion. Figure 8.8 shows as an example a time interval of $2s$ between bus communication cycle 1300 and 3300.

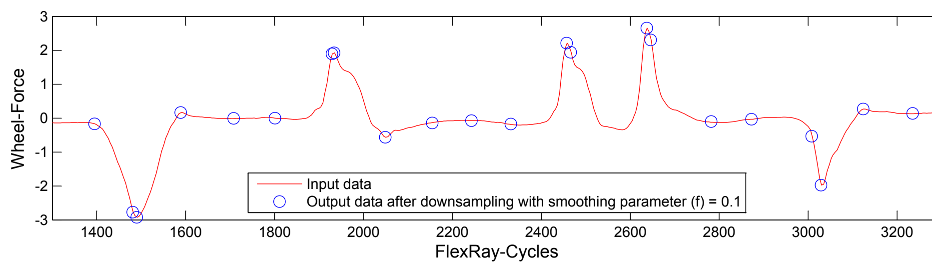
8.2.3 Impact of the SendQueue size on peak sequence detection

Since the SystemC simulation kernel uses a discrete event simulator (see Chapter 2), the dynamics of a specific Restbus simulation model strongly depends on the complexity of the model under investigation. Therefore, a general recommendation regarding the optimal size of the synchronization buffer cannot be given. However, given a concrete model, the optimal size can be empirically determined.

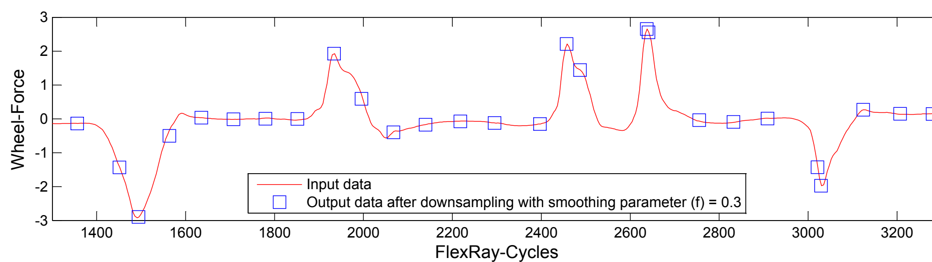
In contrast to the experiments conducted in Section 8.2.1, this section investigates the quality of the approximated input data sequence. Moreover, we mainly focused here on the quality of the approximated peak sequences (e.g.: time intervals $[1400, 1600]$, $[1900, 2100]$, $[2400, 2600]$).

For this experiment, the value of the smoothing parameter was set to $f = 0.3$. Figures 8.9a 8.9b 8.9c 8.9d show for illustrative purposes, an time window between the FlexRay communication cycles 1300 and 5200 of the recorded samples (see Figure 8.7).

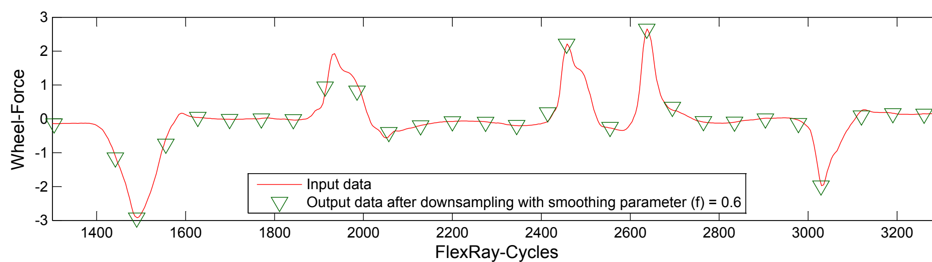
Four different SendQueue sizes were chosen $(5, 10, 50, 100)$. Again, as a general observation, the investigation showed that the original data sequence could be reconstructed by the synchronization process. Additionally, the experiments also showed a better approximation for the SendQueue size 50.



(a) Variation of the smoothing parameter of R. LOWESS with $size(SQ)_{max} = 10$ $L = 0.1$.

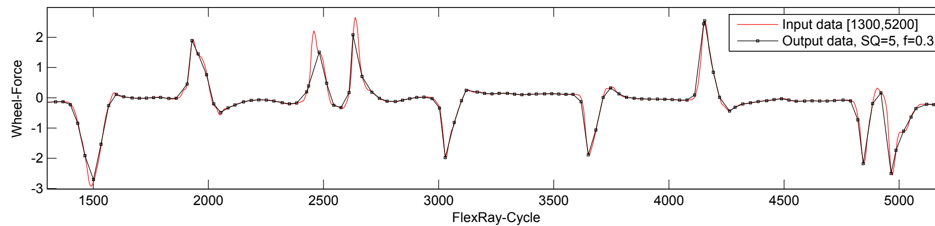


(b) Variation of the smoothing parameter of R. LOWESS with $size(SQ)_{max} = 10$ $L = 0.3$.

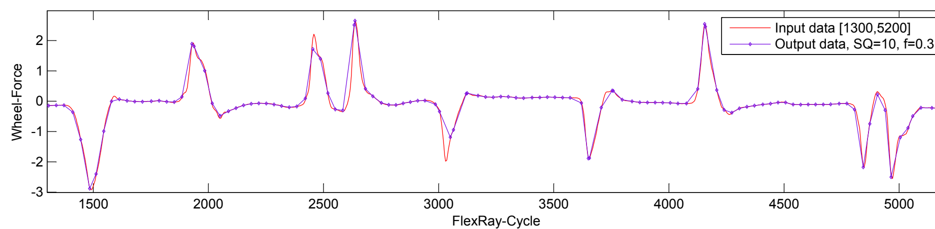


(c) Variation of the smoothing parameter of R. LOWESS with $size(SQ)_{max} = 10$ $L = 0.6$.

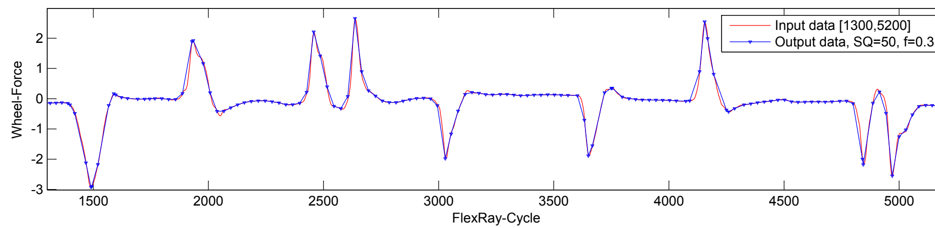
Figure 8.8: Variation of the smoothing parameter of R. LOWESS



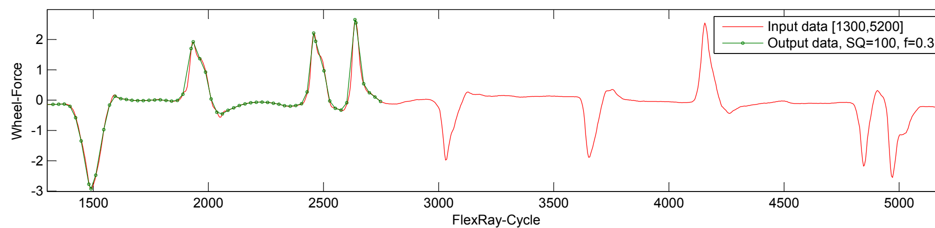
(a) Peak sequence detection for $SQ = 5$ and $f = 0,3$



(b) Peak sequence detection for $SQ = 10$ and $f = 0,3$



(c) Peak sequence detection for $SQ = 50$ and $f = 0,3$



(d) Peak sequence detection for $SQ = 100$ and $f = 0,3$

Figure 8.9: Impact of the SendQueue size on peak sequence detection

As depicted in Figure 8.9d, the synchronization process did not perform well for SendQueue size 100. This is an indication that this queue size was not appropriate. Basically, a large SendQueue size would lead to a large Receive-Queue at the moment the downsampling process starts. As a consequence, the downsampling process would not finish on time.

The processing speed ratio between the Restbus simulator and the bus network is also has an influence on the synchronization process. This investigation will be made in Section 8.2.4.

8.2.4 Impact of the data processing rate ratio on data synchronization

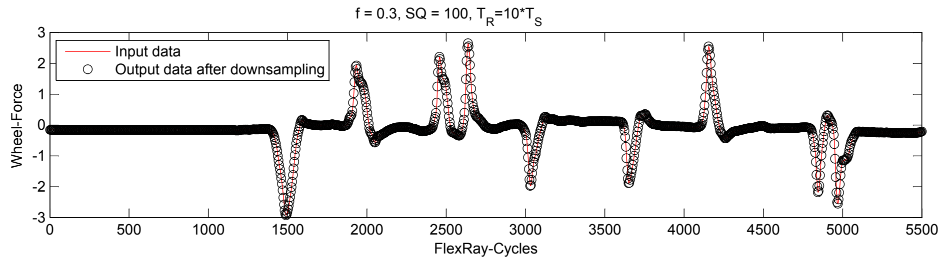
Figures 8.10 highlight additional observations made during our investigation. Hereby, the goal was to find out the limitations of our synchronization approach with respect to the adaptation of the synchronization algorithm to the variation of the data processing rate of the Restbus simulator.

Basically, these experiments can be seen as stress test of the synchronization algorithm. For this purpose, the data processing rate ratio between the Restbus simulator (T_R) and the physical bus network (T_S) was incrementally increased in order to better observe the impact on the quality of the data sequence received by the Restbus simulator.

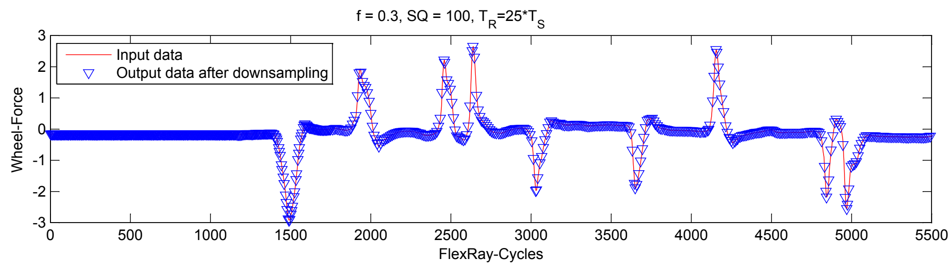
Important to mention here is that these figures only display the comparison of the input data sequence with the data sequence resulting from the synchronization process. The timing point at which the data points were sent or received by the respective components were not in the scope of the investigation.

As highlighted by the Figures 8.10a and 8.10b, we could observe a good performance of the synchronization algorithm regarding the quality of the approximated input data sequence. However, beyond a data processing rate ratio of 25 (see 8.10c and 8.10d), we could observe a deterioration of the performance of the synchronization algorithm. This is due to the fact that the transmission delay increases as the size of the SendQueue increases, an that the simulation was stopped as soon as the stimulus generator terminated (see Figure 8.5).

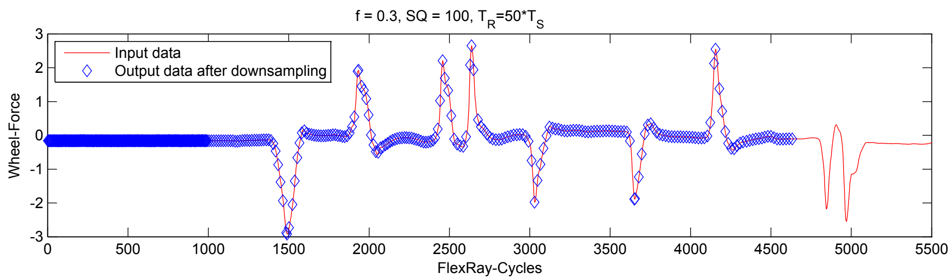
As a conclusion, for this concrete given Restbus simulation model a data processing rate ratio beyond 25 would not work.



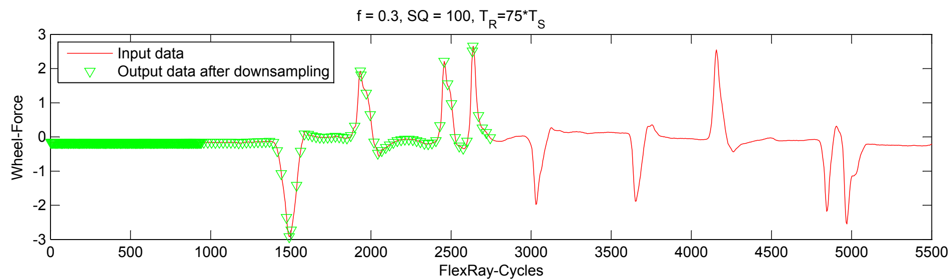
(a) Impact of the data processing rate ratio for $T_R = 10T_S$



(b) Impact of the data processing rate ratio for $T_R = 25T_S$



(c) Impact of the data processing rate ratio for $T_R = 50T_S$



(d) Impact of the data processing rate ratio for $T_R = 75T_S$

Figure 8.10: Impact of the processing rate ratio on peak sequence detection

8.2.5 Summary

The experiments in this chapter have demonstrated the correctness of our synchronization approach concerning the approximation of an input signal provided a bus network in a Restbus simulation environment. Based on the investigation regarding the configuration of the incorporated data smoothing procedure Robust LOWESS no conclusion could be made. Furthermore, we could observe that the size of the synchronization buffer has a strong impact on the quality of the approximated input data sequence. Therefore, the size of the SendQueue has to be carefully chosen. However, since the data processing rate of the Restbus simulator may vary at runtime, the optimal configuration of the Synchronization buffer needs to be experimentally defined.

As an overall conclusion, our approach has proven to be applicable for functional validation purposes in early phases of a system design process.

Chapter 9

Conclusion

9.1 Summary

In this thesis we have presented a framework for Restbus simulation. In contrast to existing Restbus simulation solutions available on the market, we introduced an approach that only requires a common off-the-shelf PC equipped with a bus network communication controller. This approach can be used for the simulation of monitoring or control automotive systems.

Furthermore, the Restbus simulator can either be applied to simulate the DUT or to run a testbench. The usage as testbench is needed when the system under test is a real (physical) component.

To guide through the development of the Restbus simulation model, we have introduced a design methodology (see Chapter 4) that covers aspects such as design modeling, timing analysis and simulation. The Restbus simulation framework supports the simulation of SystemC models.

To model the components of the Restbus simulation design, we make use the IP-XACT standard. Besides the introduction of the simulation framework, a further contribution of this thesis was the definition of timing extensions for IP-XACT to capture timing constraints and therefore enhance the IP-XACT component descriptions (see Chapter 5).

The captured timing information can then help verify that the simulation model conforms to its timing requirements before starting the actual Restbus simulation process.

PSL is used as formal specification language for the timing requirements. Hereby, our contribution is the definition of transformation rules from each timing constraint element of the timing extensions to executable PSL formulas.

Based on these transformation rules, timing requirements can be formalized in order to conduct a preliminary timing verification step on the simulation model that will run on the PC.

As aforementioned, our Restbus simulation framework supports the simulation of SystemC models. Since SystemC uses a discrete event simulator, the data processing rate of the Restbus simulation processes cannot always be determined, specially for complex simulation model (see Section 7.2).

As a core contribution of this thesis, we introduced a synchronization algorithm that handles potential data processing rate issues between the Restbus simulator running on the host PC and the bus network. The applicability of our synchronization approach depends on the complexity of the design to be simulated on the host PC.

Our synchronization approach makes use of data processing techniques applied in the field of Multirate systems. This is particularly useful for realistic situations where the Restbus simulator has a lower data processing rate than the bus network. In that case, at run-time, the bus network will generate much more data than the simulator can process and inversely the simulator will not produce network data fast enough.

Both the feasibility of the timing verification step and the performance of the synchronization approach have been evaluated in the Chapters 6 and 8 respectively.

The Evaluation of the timing verification approach was done using a case study consisting of a SystemC model of an automotive BBW application. The application is distributed over a set of virtual ECUs and includes ABS functionality. The implementation of the BBW model was done in our lab (C-Lab). The overall simulation model also includes a SystemC model of the FlexRay communication controller. By means of these experiments, we could demonstrate the use of the derived PSL formulas in commercial verification tool and thus the feasibility of our timing verification approach.

The evaluation of the synchronization approach was done in Chapter 8. The approach was validated on a HIL test environment consisting of a Steer-By-Wire system. This test environment has also been used in several projects [90, 49].

The steer-by-wire system setup consisted of tree communicating nodes namely: the Restbus simulator running on the PC and two real ECUs (steering wheel and wheel). The PC was connected to the bus network via FlexRay PCI Card.

Key parameters for the configuration of the synchronization algorithm buffer were identified. The experiments have demonstrated the appropriateness of our synchronization approach concerning the approximation of an input sig-

nal provided a bus network in a Restbus simulation environment where the Restbus simulator has a lower data processing rate than the bus network. We could observe that the size of the synchronization buffer has a strong impact on the quality of the approximated input data sequence. Therefore, the size of the SendQueue has to be carefully chosen.

However, since the data processing rate of the Restbus simulator may vary at runtime, the optimal configuration of the Synchronization algorithm needs to be experimentally defined.

9.2 Outlook

9.2.1 Synchronization

As discussed in Chapter 7, the proposed Synchronization approach incorporates the data smoothing technique Robust LOWESS. Investigations regarding the configuration of the data smoother did not come up with a clear conclusion. Further investigations could be made in that direction.

However, there are other data smoothing methods such as splines and wavelets. These methods could also be investigated in future work.

In this thesis, our focus lied on the application scenario monitoring, where the system running in the Restbus simulator has lower data processing rate than the bus network and needs to perform some monitoring tasks. This was to our opinion the more complex application scenario. However, other scenarios could also be investigated.

9.2.2 Timing verification

Concerning the proposed timing extensions, further timing constrains specified by TADL2 could also be included into the verification process.

The correctness of our transformation rules have only been validated using a reference model. In future work these transformation rules could be formally proven.

Appendix A

Verification unit

In PSL, a verification unit, is used to group verification directives and other PSL statements. The PSL Identifier following the keyword *vunit* is the name, by which this verification unit is known to the verification tools. A verification unit may contain HDL declarations, including declarations of signal names that are also declared in the design module or instance to which the verification unit is bound. This allows a verification unit to import a design signal written in one HDL into a verification unit written using another HDL flavor. It also allows a verification unit to give new behavior to a signal in the design under verification [52].
The following declaration of the verification unit shows the definition of the verification unit used for our timing verification experiments.

```
vunit abs_cc(abs_cc_vif)
{
   default clock = (posedge clock);

    //------------------------------------------------------------------------------------
    //-- Basic sequences declaration
    //------------------------------------------------------------------------------------
    sequence repeatSequenceDefaultSpan (
             boolean dataEvent; const lower, upper) =
    {[*lower:upper]; rose(dataEvent)};

    sequence repeatSequenceDefaultSpanUpper (
             boolean dataEvent; const lower) =
    {[*lower:inf]; rose(dataEvent)};

    sequence strongDelaySequence (
             boolean stimulus, response; const lower, upper) =
    {rose(stimulus); [*lower:upper]; rose(response)};

    //------------------------------------------------------------------------------------
    //-- Basic properties declaration
    //------------------------------------------------------------------------------------
```

```
property repeatConstraintDefaultSpan (
         boolean timingEvent; const lower, upper, span) =
always (repeatSequenceDefaultSpan (timingEvent, lower, upper));

property strongDelayConstraint (
         boolean stimulus, response; const lower, upper) =
always {rose(stimulus)} |=> {[*lower:upper]; rose(response)};

property repetitionConstraint (
         boolean refEvent, timingEvent; const lower, upper, span, jitter) =
always ({[*lower:upper]; rose(refEvent)}) && ({rose(refEvent)} |=>
        {[*0:jitter]; rose(timingEvent)});

//-- 1. AgeConstraint
property ageConstraint (boolean stimulus, response; const minimum, maximum) =
always {stimulus[*]; [*minimum:maximum]; response};

//-- 2. ReactionConstraint
property reactionConstraint (
         boolean refEvent, stimulus, response; const minimum, maximum) =
always {stimulus; [*minimum:maximum]; response[*]};

//-- 3. SporadicConstraint
property sporadicConstraint (
         boolean refEvent, timingEvent; const lower, upper, jitter, minimum) =
always ({[*lower:upper]; rose(refEvent)}) &&
       ({refEvent; [*0:jitter]; timingEvent}) &&
       ( {[*minimum:inf]; timingEvent});

//-- 4. InputSynchronizationConstraint
property inputSynchronizationConstraint (
         boolean refEvent, stimulus1, stimulus2; const offset, tolerance) =
always ({rose(refEvent)} |=> {[*offset:tolerance]; rose(stimulus1)}) &&
       ({rose(refEvent)} |=> {[*offset:tolerance]; rose(stimulus2)});

//-- 5. OutputSynchronizationConstraint
property outputSynchronizationConstraint (
         boolean stimulus, response1, response2; const offset, tolerance) =
always ({rose(stimulus)} |=> {[*offset:tolerance]; rose(response1)}) &&
       ({rose(stimulus)} |=> {[*offset:tolerance]; rose(response2)});

//-- 6. OrderConstraint
property orderConstraint (boolean refEvent, left, right) =
always (left; [*]; left before right);
```

```
    //-----------------------------------------------------------------------
    //--- Verification unit
    //-----------------------------------------------------------------------
    REPEATCONSTRAINT: assert repeatConstraintDefaultSpan(w_Sent_To_VR, 0, 25, 1);

    STRONGDELAYCONSTRAINT:
    assert strongDelayConstraint(vA_Sent, absCtrlVR_sent, 0, 25);

    REPETITIONCONSTRAINT:
    assert repetitionConstraint(vA_Rec_By_VR, absCtrlVR_sent, 0, 25, 1, 8);

    INPUTSYNCCONSTRAINT:
    assert inputSynchronizationConstraint(busCycleStart, w_Rec_By_VR, w_Rec_By_VL,
    0, 9);

    OUTPUTSYNCCONSTRAINT:
    assert outputSynchronizationConstraint(busCycleStart, absCtrlVR_sent,
    absCtrlVL_sent, 0, 9);

    AGECONSTRAINT:
    assert ageConstraint(vA_Sent, absCtrlVL_sent, 0, 25);

    REACTIONCONSTRAINT:
    assert reactionConstraint(busCycleStart, vA_Rec_By_HL, absCtrlHL_sent, 0, 5);
}
```

Appendix B

Pictorial representation of the IP-XACT Schema Extensions

The graphics for this document have been generated by taking screen-shots of the various files as they are displayed in Altova's XML environment XMLSpy. XMLSpy is a registered trademark of Altova GmbH. Within this document, pictorial representations of the information in the schema files illustrate the structure of the schema and define any constraints of the standard.

B.1 Diagrams

The diagrams used throughout this standard graphically detail the organization of the elements and attributes.

B.1.1 Elements and sequences

Figure B.1 shows the sequence-compositor. At the left is a branch element, element1, with some descriptive text below. element1 is connected to a sequence-compositor. The sequence-compositor defines the order the subelements appear in the branch element. subElement1 shall appear first inside of element1. This is followed by subElement2, subElement3, subElement4, and subElement5 before closing element1.
a) subElement1 is a mandatory element, as indicated by the solid line of the containing box. The type of the data contained in this element is set to string and it has a default value of ip-xact if the element is present, but left empty. b) subElement2 is an optional element, as indicated by the dashed-line of the containing box. c) subElement3 is an mandatory element that may appear multiple times, indicated by the doublesolid line of the containing box. The number of times the element may appear is indicated by the range of the numbers listed below the element. d) subElement4 is an optional element that may appear multiple times, as indicated by the doubledashed line of the containing box. The number of times the element may appear is indicated by the range of the numbers listed below the element. e) subElement5 is a mandatory branch element that contains further elements inside, as indicated by the small plus sign (+) in the small box on the right.

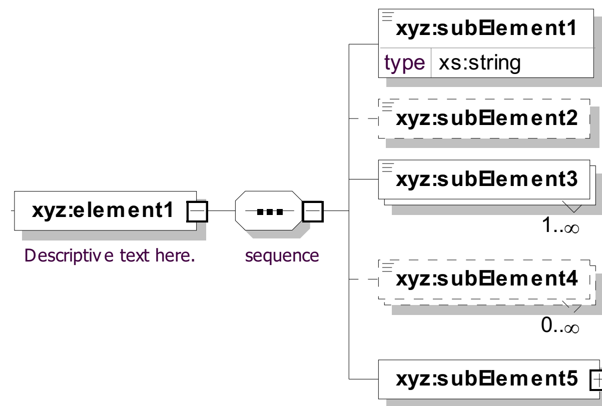


Figure B.1: Sequence-compositor

Figure B.2 shows variations of a sequence-compositor. root1 is connected to an optional sequencecompositor, as indicated by the symbol being drawn with a dashed line. The element1 may appear first inside of root1; if it does, it shall be followed by element2. Each subelement is connected to a sequence-compositor.

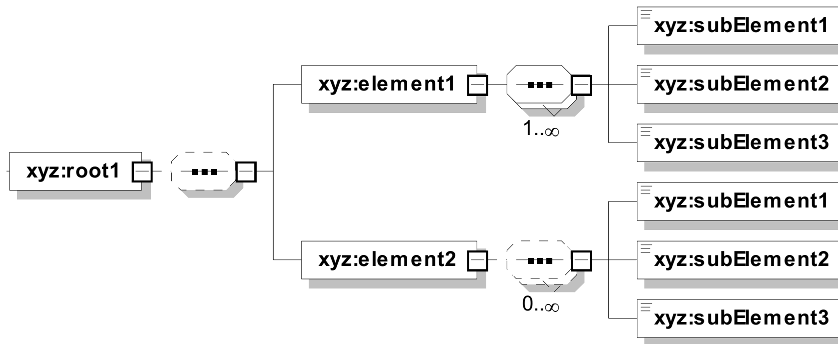


Figure B.2: Sequence-compositor variations

element1 may contain one or more of the following sequences in the following order: subElement1 and subElement2 and subElement3. The number of times the sequence-compositor may appear is indicated by the range of the numbers listed below the symbol. If the range is greater than 1, the sequence-compositor symbol is drawn with double lines. element2 is optional and may contain one or more of the following sequences in the following order: subElement1 and subElement2 and subElement3. The number of times the sequence-compositor may appear is indicated by the range of the numbers listed below the symbol. If the range starts at 0 and the maximum is greater then 1, the sequence-compositor symbol is drawn with double-dashed lines.

B.1.2 Elements and choices

Figure B.3 shows the variations of the choice-compositor. The root is connected to a choice-compositor. The choice-compositor specifies that one of the elements on the right side shall

be chosen. root may contain one of the following: element1, element2, or element3. Each subelement is connected to a choice-compositor.

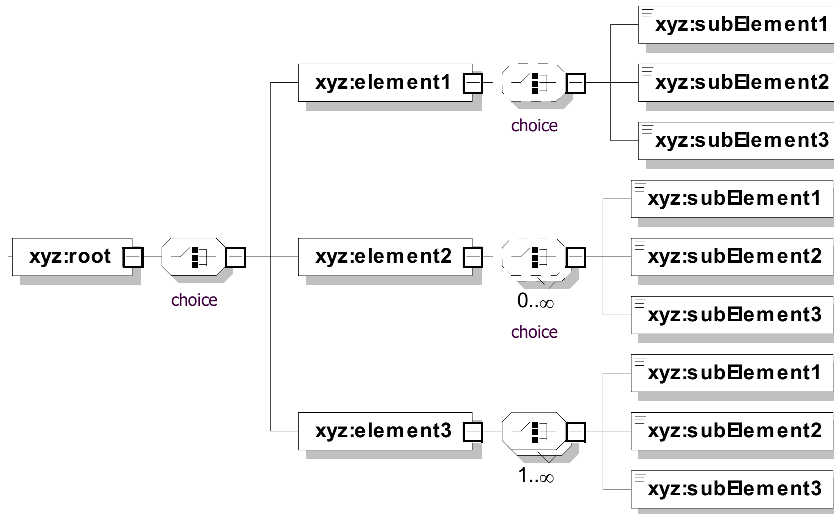


Figure B.3: Choice-compositor variations

a) element1 may contain one of the following: subElement1, subElement2, or subElement3, as indicated by the symbol being drawn with a dashed line. b) element2 may contain any (0 or more) of the following: subElement1, subElement2, or subElement3 in any order. The number of times the choice-compositor may appear is indicated by the range of the numbers listed below the symbol. If the range starts at 0, the choice-compositor is drawn with dashed lines. c) element3 may contain one or more of the following: subElement1, subElement2, or subElement3 in any order. The number of times the choice-compositor may appear is indicated by the range of the numbers listed below the symbol. If the range is greater than 1, the choice-compositor is drawn with double lines.

B.1.3 Elements, attributes, groups, and attributeGroups

Figure B.4 shows the use of attributes, groups, and attributeGroups. element1 contains two attributes, shown in the tab shaped box labeled attributes. attribute1 is optional, as indicated by the dashed containing box. attribute1 also has a type defined of integer and a default value of 7 if the attribute is not present. attribute2 is a required attribute, as indicated by the solid containing box, and is of type boolean with no default. The ordering in which attribute1 and attribute2 appear inside element1 is irrelevant.
a) eGroup1 is an element group inside element1. This group contains three subelements and the group symbol can be replaced by a solid line. The name of the group has no representation in the resulting output description. An element group can be optional, as indicated by a dashed outline (not shown) and it can also have a range, as indicated by numbers below the group symbol (not shown). b) aGroup1 is an attributeGroup inside element2 and element3. This attributeGroup contains two attributes, attribute7 and attribute8. Inside element2, the attributeGroup is shown in its collapsed form, as indicated by the small plus sign (+) inside the small box. Inside element3 the attribute- Group is shown in it expanded form, as indicated by

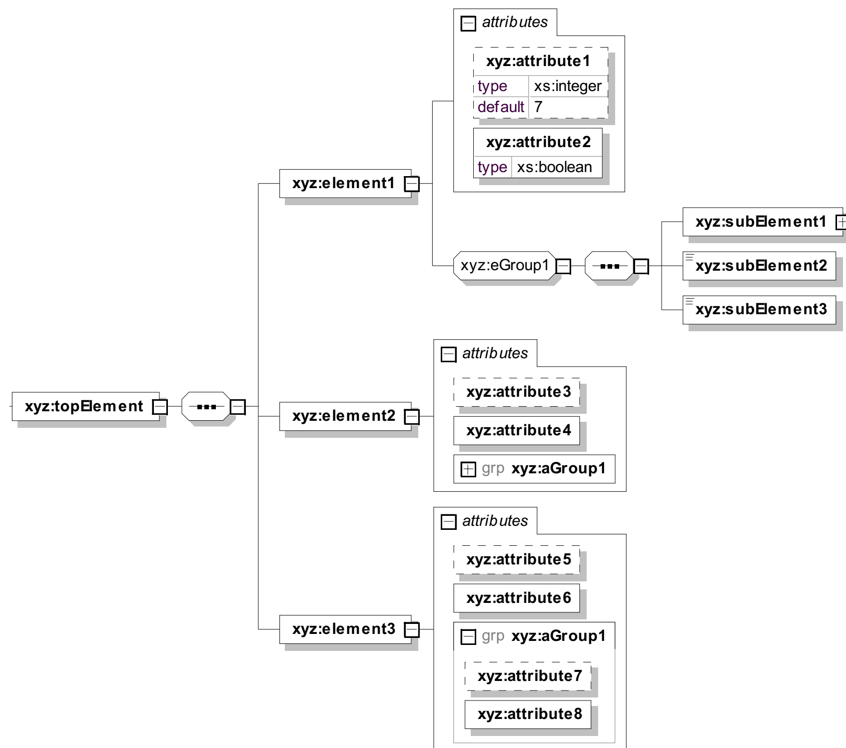


Figure B.4: Attributes, groups, and attributeGroups

the small minus sign (-) inside the small box. element2 contains four attributes: attribute3, attribute4, attribute7, and attribute8. element3 also contains four attributes: attribute5, attribute6, attribute7, and attribute8. The name of the attributeGroup has no representation in the resulting description.

B.1.4 Wildcards

Figure B.5 shows the use of wildcards. A wildcard is depicted by the rounded box with the any ##any text. Wildcards indicate that any well-formed attribute or element may be inserted into the containing element.

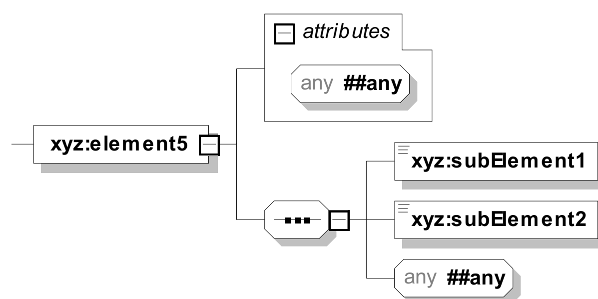


Figure B.5: Wildcards

List of Acronyms

ABS Anti-lock Braking System. 7, 94, 99, 100, 108, 109, 148

ABV Assertion-Based Verification. 54, 55, 59, 71

ACC Adaptive Cruise Control. 71, 87

API Application Programming Interface. 68

AT Approximately Timed. 66

AUTOSAR AUTomotive Open System ARchitecture. i, 3, 14–17, 21, 52, 68

BBW Brake-By-Wire. 7, 93, 99, 100, 102, 105, 148, 164, 167

BFM Bus Functional Model. 66

BMT Behavioral Modeling Tool. 67

BSW Basic Software. 15, 16, 19

CAN Controller Area Network. 28, 29, 51, 89

CC Communication Controller. 68

CIL Component-In-the-Loop. 27

COLA Component Language. 51

CSMA/CD+AMP Carrier Sense Multiple Access/Collision Detection with Arbitration on Message Priority. 28

CTL Computational Tree Logic. 48, 75

DFT Discrete time Fourier Transforms. 116

DSP Digital Signal Processor. 116

DUT Design Under Test. 58, 65, 67, 147

DUV Design Under Verification. 71, 75, 96

ECU Electronic Control Unit. i, 1, 2, 7, 14–16, 18–20, 31, 50, 51, 71, 73, 99, 136, 137, 141, 148

EDA Electronic Design Automation. 3, 26, 49, 75, 103, 105, 109

EMF Eclipse Modeling Framework. 52

ESL Electronic System-Level. 37

ET Event-triggered. 14

FIBEX Field Bus EXchange Format. 68

FOL First-Order Logic. 78

FT Fourier Transforms. 116

FTDMA Flexible Time Division Multiple Access. 30, 73

GDL General Description Language. 43

HDL Hardware Description Language. 40, 43, 44

HdS Hardware-dependent Software. 49, 50

HIL Hardware-In-the-Loop. 2, 5, 7, 27, 58, 63, 130, 133, 148

IMC Interface-Method-Call. 38

IP Intellectual Property. 33, 49

ISO International Standardization Organization. 28

IVL Intermediate Verification Language. 53

LOWESS LOcally WEighted Scatterplot Smoothing. 138

LT Loosely Timed. 66

LTI Linear Time Invariant. 116

LTL Linear-time Temporal Logic. 48, 75

MAD Median Absolute Deviation. 114

MBD Model-Based Design. 26, 27

MIL Model-In-the-Loop. 27

MPSoC MultiProcessor System-on-Chip. 50

NIT Network Idle Time. 102, 104

OEM Original Equipment Manufacturer. 1, 30

PSL Property Specification Language. viii, 42–48, 57, 58, 61, 63, 71, 74, 75, 80–86, 90, 91, 94, 96, 97, 99, 103, 105, 107, 109, 147, 148, 151

RBS Restbus Simulation. 50, 51, 58, 63, 122

RCP Rapid-Control-Prototyping. 27

RE Runnable Entities. 18

RTE Runtime Environment. 15, 16, 18, 19

RTL Register Transfer Level. 5, 52, 66, 76

RTOS Real-Time Operating System. 4

SAM System Architectural Model. 66

SCV SystemC Verification Library. i

SERE Sequential Extended Regular Expression. 43, 44, 48, 74, 75, 86, 167

SIL Software-In-the-Loop. 27

SLDL System-Level Design Language. i, 37

SW-C Software Component. 15, 16, 18, 19, 137

SysML Systems Modeling Language. 3

TADL Timing Augmented Description Language. 20, 21

TADL2 Timing Augmented Description Language 2. viii, 20–25, 57, 58, 71, 76–84, 86, 88, 89, 91, 93–95, 97, 109, 149, 167

TDMA Time Division Multiple Access. 30, 73

TLM Transaction Level Modeling. 5, 40, 52, 64, 66, 76, 136, 137

TT Time-triggered. 14

UT Un-Timed. 66

VFB Virtual Functional Bus. 18

XML Extended Markup Language. 77, 86

List of Figures

1.1 Proposed design flow . . . 6

2.1 Block diagram of an open-loop control system . . . 11
2.2 Block diagram of a closed-loop control system . . . 12
2.3 AUTOSAR layered software architecture . . . 15
2.4 Virtual Functional Bus Timing . . . 18
2.5 Software Component Timing . . . 18
2.6 System Timing . . . 19
2.7 Basic Software Timing . . . 20
2.8 ECU Timing . . . 20
2.9 A set of event occurrences satisfying a RepeatConstraint with span of 2 . . . 24
2.10 A set of event occurrences satisfying a DelayConstraint [80] . . . 25
2.11 A set of target event occurrences satisfying a SynchronizationConstraint. Overlapping tolerance windows and multiple event occurrences [80] . . . 25
2.12 Standard CAN frame with 11-Bit identifier . . . 29
2.13 Extended CAN frame with 29-Bit identifier . . . 29
2.14 Timing hierarchy within the communication cycle [41] . . . 30
2.15 FlexRay frame format [41] . . . 32
2.16 Active star topology combined with a passive bus topology [41] . . . 32
2.17 IP-XACT design environment (source: [48]) . . . 33
2.18 IP-XACT simplified Design structure. . . . 35
2.19 Tree view of the xml schema definition of an IP-XACT Ports . . . 36
2.20 SystemC language architecture [15] . . . 38
2.21 Artifacts of SystemC [15] . . . 39
2.22 SystemC simulation kernel [15] . . . 40
2.23 TLM2 Use Case, Codingstyles and Mechanisms [96] . . . 41
2.24 PSL is a Layered Language [58] . . . 43

4.1 Graphical view of an IP-XACT design example . . . 59
4.2 Restbus simulation design flow . . . 60
4.3 Divergence between simulated time and real-time . . . 61
4.4 Simulation models used during the design process . . . 62
4.5 System Architecture of Restbus simulation framework . . . 66
4.6 Abstraction refinement and TLM mapping source: [14] . . . 67
4.7 Generic behavior of the adapter component . . . 68

5.1 Simple example illustrating the need for the specification of timing constraints . 72
5.2 Resulting network topology after deployment 73
5.3 FlexRay communication schedule configuration 73
5.4 Analysis window of the distributed system events 74
5.5 A simplified design structure of IP-XACT 76
5.6 DataEvent contains a reference to the schema element PortRef 77
5.7 Event chain 78
5.8 Proposed timing extensions for IP-XACT 81
5.9 A set of event occurrences satisfying a RepeatConstraint for span=2. 82
5.10 A set of event occurrences satisfying a StrongDelayConstraint [80] 84
5.11 A set of event occurrences satisfying a RepetitionConstraint with $span = 2$ . . . 86
5.12 Graphical representation of the XML schema of the DelayConstraint element . 87
5.13 Sporadic constraint 89
5.14 Periodic constraint 91
5.15 EventChainIn: constrains two or more stimuli with one response 92
5.16 EventChainOut: constrains two or more responses with one stimulus 93
5.17 Input synchronization constraint 93
5.18 Synchronization Constraint 94
5.19 Order constraint 95

6.1 Block diagram of the ABS system, Source: C-Lab 101
6.2 Network topology of the functional components of the BBW model 102
6.3 Waveform of the simulation run showing traces of all data events under observation 104
6.4 Assertion analysis window for the basic timing constraints 106
6.5 Assertion analysis window for Age- and ReactionConstraints 107
6.6 Assertion analysis window for SynchronizationConstraints 109

7.1 Weight function example $w_k(x_i)$ for leftmost ($i = 1$), interior ($i = 14$) and rightmost ($i = 27$) data points (span $L = 9$, f = 0.3, n = 29) 112
7.2 Locally weighted regression vs robust locally weighted regression, $L = 15$, $f = 0.1$) 113
7.3 Weighted regression example for $i = 1, \ldots, 4$ and $L = 5$ 115
7.4 Display of the analog (continuous) signal and display of digital samples versus the sampling time instants [92]. 115
7.5 Example of the digital signal and its amplitude spectrum [92]. The signal is sampled at a rate of 8000Hz. 117
7.6 Frequency domain: spectrum after downsampling [92] 119
7.7 Interpolation [92] 120
7.8 Application scenario of the synchronization algorithm 122
7.9 Synchronization problem, Receiver faster than sender 123
7.10 Problem: missing data at time 2 ms during communication between a slow sender and a fast receiver 123
7.11 Interpolation: repetition of last valid value received 124
7.12 Synchronization buffer: 3 level queuing mechanism sender faster than receiver. 124

7.13 Incoming data are inserted into the ReceiveQueue (1). Data are taken form the SendQueue and sent to the receiver based on the FIFO principle (2) 125
7.14 Downsampling during the main phase of the synchronization process 126
7.15 Initialization phase of the synchronization process 127
7.16 End of the initialization phase, ProcessQueue is full 127
7.17 Outliers are filtered out during the data smoothing process 128

8.1 Overview of the steer-by-wire architrecture 134
8.2 Test environment 135
8.3 Active steering wheel 135
8.4 Wheel and damping testbed 136
8.5 Architecture of the testbench used for the evaluation 138
8.6 Delay caused be the downsampling process depending on the SendQueue size . 139
8.7 Wheel raw data 140
8.8 Variation of the smoothing parameter of R. LOWESS 142
8.9 Impact of the SendQueue size on peak sequence detection 143
8.10 Impact of the processing rate ratio on peak sequence detection 145

B.1 Sequence-compositor 156
B.2 Sequence-compositor variations 156
B.3 Choice-compositor variations 157
B.4 Attributes, groups, and attributeGroups 158
B.5 Wildcards 158

List of Tables

2.1 Notation used in the definition of the TADL2 semantics 22
2.2 Semantic definition of the TADL2 RepeatConstraint. 24
2.3 Semantic definition of the TADL2 DelayConstraint. 24
2.4 Semantic definition of the TADL2 SynchronizationConstraint. [80] 25
2.5 Union and Clocking operators 44
2.6 SERE Repetition operators 44
2.7 Sequence operators 45
2.8 Further foundation language operators 46
2.9 Boolean operators 46

5.1 Notation used in the definition of the TADL2 semantics 79

6.1 BBW communication matrix 102
6.2 Assertion analysis results for Repeat-, StrongDelay-, and RepetitionConstraint . 107
6.3 Assertion analysis results for AgeConstraint and ReactionConstraint 108
6.4 Assertion result analysis for Synchronization related constraint 108

List of Own Publications

[1] Markus Becker, Gilles Bertrand Gnokam Defo, Wolfgang Müller, and et al. “MOUSSE: scaling MOdelling and verification to complex heterogeneoUS embedded Systems Evolution”. In: *Design, Automation and Test in Europe (DATE 2012)*. Dresden, Mar. 2012.

[2] Gilles Bertrand Gnokam Defo, Christoph Kuznik, and Wolfgang Mueller. “Verification of a can bus model in SystemC with functional coverage”. In: *Proceedings of the fifth IEEE Symposium on Industrial Embedded Systems (SIES2010)*. 2010.

[3] Gilles Bertrand Gnokam Defo and Wolfgang Müller. “Synchronisation eines SystemC Restbus-Simulators mit einem Hardware-In-the-Loop FlexRay Netzwerk”. In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*. MBMV 2011. Feb. 2011.

[4] Wolfgang Mueller Kay Klobedanz Gilles Bertrand Defo. “Distributed Coordination of Task Migration for Fault-Tolerant FlexRay Networks”. In: *Proceedings of the fifth IEEE Symposium on Industrial Embedded Systems (SIES2010)*. 2010.

[5] K. Klobedanz, Gilles Bertrand Gnokam Defo, Henning Zabel, and et al. “Task Migration for Fault-Tolerant FlexRay Networks”. In: *IFIP Working Conference on Distributed and Parallel Embedded Systems (DIPES 2010)*. Brisbane, Australia: Springer, 2010.

[6] Tobias Knieper, Gilles Bertrand Gnokam Defo, Paul Kaufmann, and Marco Platzner. “On Robust Evolution of Digital Hardware”. In: *2nd IFIP Conference on Biologically Inspired Collaborative Computing (BICC 2008)*. Milano, Italy: Springer, Sept. 2008, pp. 213–222.

[7] Christoph Kuznik, Marcio F. S. Oliveira, Gilles Bertrand Gnokam Defo, and Wolfgang Müller. “Systematic Application of UCIS to Improve the Automation on Verification Closure”. In: *Proceedings of DVCON* (2013).

[8] Christoph Kuznik, Gilles Bertrand Gnokam Defo, and Wolfgang Müller. “Semi-automatische Generierung von Überdeckungsmetriken mittels methodischer Verikationsplan Verarbeitung”. In: *17. Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV 2014)* (Mar. 2014).

[9] Tao Xie, Gilles Bertrand Gnokam Defo, and Wolfgang Mueller. “An Eclipse-based Framework for the IP-XACT-enabled Assembly of Mixed-Level IPs”. In: *First Workshop on Hands-on Platforms and tools for model-based engineering of Embedded Systems (HoPES)*. HoPES 2010 within ECMFA 2010. Paris, 2010.

Bibliography

[10] Melih Çakmakci A. Galip Ulsoy Huei Peng. *Automotive Control Systems*. Cambridge University Press, 2012.

[11] Altova. *XMLSpy XML Editor*. 2013. URL: http://www.altova.com/xmlspy.html.

[12] ASAM Association for Standardisation of Automation and Measuring Systems. *FIBEX Field Bus Exchange Format*. www.asam.net. 2010.

[13] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008. ISBN: 026202649X, 9780262026499.

[14] D.C. Black, J. Donovan, B. Bunton, and A. Keist. *SystemC: From the Ground Up, Second Edition*. Springer, 2009. ISBN: 9780387699578. URL: http://books.google.de/books?id=YGDqjwEACAAJ.

[15] David C. Black, Jack Donovan, Bill Bunton, and Anna Keist. *SystemC: From the Ground Up*. Ed. by Kyle Smith and Richard Whitfield. Second. Springer Science+Business Media, LLC, 2010.

[16] R. Bosch. *Automotive Electrics Automotive Electronics*. Bentley Pub, 2004. ISBN: 9780837610504. URL: http://books.google.de/books?id=-hMCAAAACAAJ.

[17] R. Bosch. *CAN Specification Version 2.0*. 1991. URL: www.bosch-semiconductors.de/media/pdf_1/canliteratur/can2spec.pdf.

[18] James E. Brittain. "Electrical Engineering Hall of Fame: Harry Nyquist [Scanning Our Past]." In: *Proceedings of the IEEE* 98.8 (2010), pp. 1535–1537. URL: http://dblp.uni-trier.de/db/journals/pieee/pieee98.html#Brittain10f.

[19] Cadence Design Systems, Inc. *Incisive Formal Verifier*. 2013. URL: http://www.cadence.com/products/ld/formal_verifier/pages/default.aspx.

[20] Cadence Design Systems, Inc. *OVM-SC Library Reference Version 2.0.1*. February, 2009. URL: http://www.cadence.com.

[21] Lukai Cai and Daniel Gajski. "Transaction level modeling: an overview". In: *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. CODES+ISSS '03. Newport Beach, CA, USA: ACM, 2003, pp. 19–24. ISBN: 1-58113-742-7. DOI: 10.1145/944645.944651. URL: http://doi.acm.org/10.1145/944645.944651.

[22] A Cimatti, A Micheli, I Narasamdya, and M. Roveri. "Verifying SystemC: A software model checking approach". In: *Formal Methods in Computer-Aided Design (FMCAD), 2010*. 2010, pp. 51–59.

[23] Alessandro Cimatti, Andrea Micheli, Iman Narasamdya, and Marco Roveri. "Verifying SystemC: A software model checking approach." In: *FMCAD*. Ed. by Roderick Bloem and Natasha Sharygina. IEEE, 2010, pp. 51–59. URL: http://dblp.uni-trier.de/db/conf/fmcad/fmcad2010.html#CimattiMNR10.

[24] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model checking*. Cambridge, MA, USA: MIT Press, 1999. ISBN: 0-262-03270-8.

[25] William S. Cleveland. "Robust Locally Weighted Regression and Smoothing Scatterplots". In: *Journal of the American Statistical Association* 74 (1979), pp. 829–836.

[26] William S. Cleveland and Susan J. Devlin. "Locally Weighted Regression: An Approach to Regression Analysis by Local Fitting". English. In: *Journal of the American Statistical Association* 83.403 (1988), pp. 596–610. URL: http://www.jstor.org/stable/2289282.

[27] AUTOSAR Consortium. *AUTomotive Open System ARchitecture*. www.autosar.org. 2009.

[28] AUTOSAR Consortium. *General Specification of Basic Software Modules*. www.autosar.org. 2013.

[29] AUTOSAR Consortium. *Software Component Template*. http://www.autosar.org/download/R4.1/AUTOS 2013.

[30] AUTOSAR Consortium. *Specification of RTE*. www.autosar.org. 2013.

[31] AUTOSAR Consortium. *Specification of Timing Extensions*. www.autosar.org. 2011.

[32] Ronald E. Crochiere and Lawrence R. Rabiner. *Multirate digital signal processing*. Prentice-Hall, 1983.

[33] G. Di Guglielmo, L. Di Guglielmo, F. Fummi, and G. Pravadelli. "Enabling dynamic assertion-based verification of embedded software through model-driven design". In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*. 2012, pp. 212–217. DOI: 10.1109/DATE.2012.6176430.

[34] Doulos. *PSL The golden reference guide, Version 2 supporting PSL v1.1*. 2013. URL: http://www.doulos.com/content/products/golden_reference_guides.php#anchor_psl.

[35] Stirling Dynamics. *Stirling Dynamics Products*. 2014. URL: http://www.stirling-dynamics.com/.

[36] Cindy Eisner and Dana Fisman. *A Practical Introduction to PSL*. 1st ed. Springer US, 2006. ISBN: 978-0-387-36123-9.

[37] Eberspaecher Electronics. *FlexCard PMC/PCI, getting started*. www.eberspaecher.com. 2007.

[38] Eberspaecher Electronics. *FlexConfig user manual*. www.eberspaecher.com. 2009.

[39] Jack Erickson. *TLM Driven Design and Verification Time For a MethodologyShift*. Cadence Design, Systems, Inc., 2012. URL: http://www.cadence.com.

[40] José Ferreirós. "The road to modern logic-an interpretation". In: *Bulletin of Symbolic Logic* 7.4 (2001), pp. 441–484.

[41] Consortium FlexRay. *FlexRay Communications System Protocol Specification Version 2.1 Rev. A*. www.flexray.com. 2005.

[42] Harry D. Foster, Adam C. Krolnik, and David J. Lacey. *Assertion-Based Design*. 2nd. Springer Publishing Company, Incorporated, 2010. ISBN: 1441954627, 9781441954626.

[43] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A Practical Guide to SysML: Systems Modeling Language*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. ISBN: 0123743796, 9780080558363, 9780123743794.

[44] Ziv Glazberg et al. *PSL: Beyond Hardware Verification*.

[45] Mentor Grapics. *QuestaSim*. 2013. URL: http://www.mentor.com/products/fv/questa-verification-platform.

[46] Daniel Grosse, Markus Gross, Ulrich Kuehne, and Rolf Drechsler. "Simulation-based equivalence checking between SystemC models at different levels of abstraction". In: *Proceedings of the 21st edition of the great lakes symposium on Great lakes symposium on VLSI*. GLSVLSI '11. Lausanne, Switzerland: ACM, 2011, pp. 223–228. ISBN: 978-1-4503-0667-6. DOI: 10.1145/1973009.1973054. URL: http://doi.acm.org/10.1145/1973009.1973054.

[47] Giuseppe Di Guglielmo et al. "On the integration of model-driven design and dynamic assertion-based verification for embedded software". In: *Journal of Systems and Software* 86.8 (2013), pp. 2013 –2033. ISSN: 0164-1212. DOI: http://dx.doi.org/10.1016/j.jss.2012.08.061. URL: http://www.sciencedirect.com/science/article/pii/S0164121212002506.

[48] IP-XACT IEEE. *IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows*. www.accellera.org. 2009.

[49] ITEA2. *TIMMO-2-USE Project*. 2009. URL: http://www.timmo-2-use.org.

[50] Accelera Systems Iniciative. *TLM-2.0 Language Reference Manual*. Accelera Systems Iniciative, 2007.

[51] Accellera Systems Initiative. *Available IEEE Standards*. http://www.accellera.org/home/.

[52] Accellery Systems Initiative. *Standard for Property Specification Language (PSL)*. 2007. URL: http://www.accellera.org.

[53] Accellery Systems Initiative. *SystemC Language Reference Manual*. 2012. URL: http://www.accellera.org/home/.

[54] Accellery Systems Initiative. *Universal Verification Methodology (UVM)*. 2012. URL: http://www.accellera.org/downloads/standards/uvm.

[55] Open SystemC Initiative. *IEEE 1666 Open SystemC Language Reference Manual*. IEEE Standard Association, 2005.

[56] Open SystemC Initiative. *OSCI TLM2 USER MANUAL, Software version TLM 2.0 Draft 2, Dcument version 1.0.0*. 2007.

[57] National Instruments. *Fundamentals of Restbus Simulation*. http://www.ni.com/white-paper/13726/en. 2012.

[58] Jasper Design Automation. *Property Specification Language (PSL).* URL: http://jasper-da.com/.

[59] Christoph Kern and Mark R. Greenstreet. "Formal Verification in Hardware Design: A Survey". In: *ACM Trans. Des. Autom. Electron. Syst.* 4.2 (Apr. 1999), pp. 123–193. ISSN: 1084-4309. DOI: 10.1145/307988.307989. URL: http://doi.acm.org/10.1145/307988.307989.

[60] Moonzoo Kim, Yunho Kim, and Hotae Kim. "A Comparative Study of Software Model Checkers as Unit Testing Tools: An Industrial Case Study". In: *Software Engineering, IEEE Transactions on* 37.2 (2011), pp. 146–160. ISSN: 0098-5589. DOI: 10.1109/TSE.2010.68.

[61] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. 1st. Norwell, MA, USA: Kluwer Academic Publishers, 1997. ISBN: 0792398947.

[62] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. 2nd. Springer, 2011. ISBN: 978-1-4419-8236-0.

[63] Matthias Krause, Oliver Bringmann, Gökhan Hergenhan André andTabanoglu, and Wolfgang Rosentiel. "Timing simulation of interconnected AUTOSAR software-components". In: *Proceedings of the conference on Design, automation and test in Europe (DATE 07).* 2007, pp. 474–479.

[64] H.M. Le, D. Grosse, V. Herdt, and R. Drechsler. "Verifying SystemC using an intermediate verification language and symbolic simulation". In: *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE*. 2013, pp. 1–6.

[65] Djones Lettnin et al. "Verification of Temporal Properties in Automotive Embedded Software". In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE '08. Munich, Germany: ACM, 2008, pp. 164–169. ISBN: 978-3-9810801-3-1. DOI: 10.1145/1403375.1403417. URL: http://doi.acm.org/10.1145/1403375.1403417.

[66] R. J. Marks. *Introduction to Shannon Sampling and Interpolation Theory*. Springer, 1991. ISBN: 978-1-4613-9708-.

[67] Ricardo A. Maronna, R. Douglas Martin, and Victor J. Yohai. "Robust Statistics: Theory and Methods". In: ed. by David J. Balding et al. Wiley Series in Probability and Statistics, 2006. Chap. 1.2, pp. 2–5.

[68] MathWorks. URL: http://www.mathworks.com/products/simulink.

[69] MathWorks. URL: http://www.mathworks.de/products/simulink-coder/index.html.

[70] Deepak A. Mathaikutty. "Metamodeling Driven IP Reuse for System-on-chip Integration and Microprocessor Design". dissertation. Virginia Polytechnic Institute and State University, 2007.

[71] A. I. McLeod. *Robust Loess: S lowess Introduction and Summary*. course notes. 2004. URL: http://www.stats.uwo.ca/faculty/aim/2004/04-259/notes/default.htm.

[72] Razvan Nane et al. *IP-XACT Extensions for Reconfigurable Computing*. 2011.

[73] Nicolas Navet and Francoise Simonot-Lion. *Automotive Embedded Systems Handbook*. 1st. Boca Raton, FL, USA: CRC Press, Inc., 2008. ISBN: 084938026X, 9780849380266.

[74] Marcio F. S. Oliveira et al. "A SystemC Library for Advanced TLM Verification". In: *Proceeding of Design and Verification Conference (DVCON)*. Mar. 2012.

[75] Alan V. Oppenheim, Ronald W. Schafer, and John R. Buck. *Discrete-time Signal Processing (2Nd Ed.)* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1999. ISBN: 0-13-754920-2.

[76] Marie-Agnès Peraldi-Frati et al. "The TIMMO-2-USE project: Time modeling and analysis to use". In: *ERTS2012 International Congres on Embedded Real Time Software and Systems*. Toulouse, France, Feb. 2012.

[77] H. Posadas et al. "RTOS modeling in SystemC for real-time embedded SW simulation: A POSIX model". English. In: *Design Automation for Embedded Systems* 10.4 (2005), pp. 209–227. ISSN: 0929-5585. DOI: 10.1007/s10617-006-9725-1. URL: http://dx.doi.org/10.1007/s10617-006-9725-1.

[78] David Powell. *Failure Mode Assumptions and Assumption Coverage*. 1995.

[79] SCALOPES Project. *SCALOPES Deliverable DT4.3.6(M24) Guidelines for System-level design integration and optimization*. Deliverable. 2010.

[80] TIMMO-2-USE Project. *Deliverable D11*. 2012.

[81] TIMMO-2-USE Project. *TIMMO-2-USE: Deliverable D14: Brake-by-Wire Validator*. 2012.

[82] TIMMO-2-USE Project. *TIMMO-2-USE: TIMing MOdel - TOols, algorithms, languages, methodology, and USE cases*. 2012.

[83] W.B. Ribbens and N.P. Mansour. *Understanding Automotive Electronics, Sixth Edition*. Newnes, 2003. ISBN: 9780768012217.

[84] A. Schedl. *Goals and Architecture of FlexRay at BMW*. slides presented at the Vector FlexRay Symposium. 2007.

[85] G. Schirner, A. Gerstlauer, and R. Domer. "Automatic generation of hardware dependent software for MPSoCs from abstract system specifications". In: *Design Automation Conference, 2008. ASPDAC 2008. Asia and South Pacific*. 2008, pp. 271 –276. DOI: 10.1109/ASPDAC.2008.4483954.

[86] Satnam Singh. "Design and Verification of CoreConnectTM IP Using Esterel". English. In: *Correct Hardware Design and Verification Methods*. Ed. by Daniel Geist and Enrico Tronci. Vol. 2860. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 283–288. ISBN: 978-3-540-20363-6. DOI: 10.1007/978-3-540-39724-3_26. URL: http://dx.doi.org/10.1007/978-3-540-39724-3_26.

[87] Friedhelm Stappert, Jan Jonsson, Jürgen Mottok, and Rolf Johansson. "A design framework for end-to-end timing constrained automotive applications". In: *ERTS Embedded Real Time Software and System 2010*. Vol. 2. 2010.

[88] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. 2nd ed. Boston, MA: Addison-Wesley, 2009. ISBN: 978-0-321-33188-5. URL: http://my.safaribooksonline.com/9780321331885.

[89] Corrigan Steve. *Introduction to the Controller Area Network (CAN)*. 2002.

[90] ITEA2 Project TIMMO. *Timmo: Timing model*. 2007-2009.

[91] D. Tabakov and M.Y. Vardi. "Monitoring temporal SystemC properties". In: *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*. 2010, pp. 123–132. DOI: 10.1109/MEMCOD.2010.5558640.

[92] Li Tan. *Digital Signal Processing: Fundamentals and Applications*. Academic Press, 2008.

[93] Inc. The MathWorks. *MATLAB Curve Fitting Toolbox 3.0*. www.mathworks.com.

[94] John W. Tukey. "Exploratory Data Analysis". In: Addison Wesley, 1977. Chap. 7.

[95] Dirk Van Dalen. *Logic and structure*. Springer, 2013.

[96] Bart Vanthournout. *An Insider's View on the Making of the New TLM-2.0 Standard*. 2008.

[97] R. Vijayagopal, L. Michaels, A. Rousseau, and S. Halbach. "Automated Model Based Design Process to Evaluate Advanced Component Technologies". In: *SAE Technical Paper 2010-01-0936* (2010).

[98] Thomas Waggershauser and Dr. Robert von Höfen. *Restbussimulation für FlexRay-Netzwerke*. www.ixxat.de. 2008.

[99] Zhonglei Wang, Wolfgang Haberl, Stefan Kugele, and Michael Tautschnig. "Automatic Generation of Systemc Models from Component-based Designs for Early Design Validation and Performance Analysis". In: *Proceedings of the 7th International Workshop on Software and Performance*. WOSP '08. Princeton, NJ, USA: ACM, 2008, pp. 139–144. ISBN: 978-1-59593-873-2. DOI: 10.1145/1383559.1383577. URL: http://doi.acm.org/10.1145/1383559.1383577.

[100] STM CASE s.r.l. *The radCHECK Tool*. 2013. URL: http://www.verificationsuite.com.