Martin Schachner, BSc

Design and Implementation of a Co-Simulation Environment based on Virtual Platforms

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger

Institute for Technical Informatics

Advisor

Ass.Prof. Dipl.Ing. Dr.techn. Christian Steger
Dipl.-Ing. Dr.techn. Ralph Weissnegger, BSc (CISC Semiconductor GmbH)

Graz, August 2017

AFFIDAVIT

Abstract

As many different stakeholders with different technical background have to work together for the realization of cyber-physical systems, the complexity of such projects increases continuously. Furthermore, developments, such as those of the automotive industry, are strongly driven by the demand for cost-reduction. But not only costs play a decisive role, nowadays compliance with standards to provide functional safety become more and more important. This leads to the need for a paradigm shift in the design and development of cyber-physical systems. The main focus must be on closer cooperation between developers, such that shorter time to market can be achieved.

This work presents a way how common methodologies from software development and those from classical engineering disciplines such as mechanical or electrical engineering can be combined. The idea is grounded on graphic, model-based solutions. An existing development framework was extended by the integration of virtual hardware prototypes. This approach makes it possible to execute and simulate appropriate application code even in early design phases.

The practical part describes the way of how to extend the existing modeling framework by means of graphic description for virtual hardware prototypes. Furthermore, a possible embedding of the created platforms in the physical environment is discussed and simulated. This interaction is shown by a simplified electrical vehicle model. In particular temperature data of a lithium-ion battery is monitored by an integrated hardware platform.

keywords: cyber-physical system development, model driven engineering, model based design, UML/MARTE, virtual hardware prototyping, co-simulation

Kurzfassung

Die Entwicklung von cyber-physischen Systemen gestaltet sich zunehmend schwieriger, da viele Interessensvertreter mit unterschiedlichem technischen Hintergrund an einem heterogenen Gesamtsystem arbeiten müssen. Weiters werden Entwicklungen, wie zum Beispiel jene der Automobilindustrie, stark durch die Kostenfrage getrieben. Doch nicht nur Kosten spielen eine entscheidende Rolle, auch die Einhaltung von Standards zur Gewährung von funktionaler Sicherheit werden immer wichtiger. Dadurch ergibt sich für die Hersteller von cyber-physischen Systemen die Notwendigkeit für einen Paradigmenwechsel in der Entwicklung. Hauptaugenmerk muss dabei auf eine engere Zusammenarbeit zwischen Entwicklern gelegt werden, um eine kürzere Time-to-Market realisieren zu können.

Mit dieser Arbeit wird eine Möglichkeit vorgestellt, wie gebräuchliche Methoden aus der Softwareentwicklung und jener von klassischen Ingenieursdisziplinen, wie dem Maschinenbau oder der Elektrotechnik verbunden werden können. Im Zentrum stehen dabei graphische, modellbasierte Lösungen. Dabei wurde ein bestehendes Entwicklungsframework um die Integration von virtuellen Hardwareprototypen erweitert, sodass schon in frühen Designphasen eine Simulation mit konkretem Applikationscode möglich wird.

Die praktische Arbeit beschreibt dabei die Möglichkeiten das bestehende Modellierungsframework um die graphische Beschreibung von Hardwareprototypen zu erweitern. Weiters wird gezeigt wie die erstellten Plattformen in die physikalische Umgebung eingebettet und simuliert werden können. Diese Interaktion wird anhand eines vereinfachten Elektrofahrzeugmodells gezeigt. Dabei werden Temperaturdaten einer Lithium-Ionen-Batterie mit Hilfe einer virtuellen Hardwareplatform überwacht.

Stichwörter: cyber-physical system development, model driven engineering, model based design, UML/MARTE, virtual hardware prototyping, co-simulation

Acknowledgment

Without the help of several people, this work and my time at Graz University of Technology would not have been possible. So I would like to take the opportunity and say thank you at this point.

First I would like to thank the Institute for Technical Informatics. Especially I would like to thank my supervisors Dr. Christian Steger and Dr. Ralph Weissnegger for their great support and excellent guidance through the project duration. It would not have been possible to do my researches, without the help of my colleague Markus Schuss, whom I want to thank as well.

I am also grateful to the company CISC Semiconductor and Dr. Markus Pistauer for the cooperation during the work on this thesis.

Last but not least, I want to thank my family and friends for giving me love, guidance and courage in all life situations.

Graz, August 2017 Martin Schachner

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

This thesis describes an approach to enhance cyber-physical system (CPS) development. The work was launched in the context of the SHARC project [55], which has the aim to allow system modeling in one single Integrated Development Environment (IDE). SHARC stands for *Simulation and Verification of Hierarchical Embedded Microelectronic Systems*; it uses graphical descriptions to represent a system and enables the simulation in a very early design phase. The SHARC IDE is based on a mapping between graphical Unified Modeling Language (UML) blocks and executables, which implement their functionality in SystemC Analog Mixed Signal (AMS). Previous works on this project have shown that the connection of the individual UML blocks contains enough information to generate an executable system [53]. The work in hand describes a way of how to extend the existing SHARC IDE by using virtual hardware prototyping. This approach enables more accurate descriptions and increases, therefore, the granularity of the developed system. Simulating such systems gains additional information, which has an impact on the overall design process.

The introduction provides a short overview about the underlying motivation and necessity for the written master thesis. Further on the reader can find the description of a predefined goal for this work and finally an outline in different chapters.

1.1 Motivation

Since the performance of electronic components is continuously improved and their size decreases at the same time, the application area has changed considerably in recent years. Microelectronics are widely used in all technical disciplines and enable the realization of novel functionality. One area that is particularly affected by this development is the automotive industry. A few decades ago, a vehicle consisted of mechanical components only. As the desire for more comfort became greater and meant also a market advantage, new approaches were achieved in the early 1980s and electrical/electronic systems were installed in cars more often. Due to technical advances, this approach became more important over the past years, resulting in a shift to an interdisciplinary development. In the meantime almost any innovation is implemented by means of computer-aided systems and the development towards electric vehicles and self-driving cars will lead to a further demand for interdisciplinary cooperation. The interaction of the physical environment with electronics is often referred as cyber-physical. The development of such systems requires, therefore, the interdisciplinary cooperation between mechanical/electrical engineers and computer scientists. These requirements present new challenges and problems for the development process. Therefore, efforts are being made to define functional specifications at a high level of abstraction to allow descriptions with many views on a single item. Those descriptions may be used by domain experts to further increase the granularity of the developed system. The original approach of so-called model-driven engineering arose in software development, but now it is also entering the system design.

No matter what engineering discipline, modern development happens in a virtual environment. An important concept is the simulation of the developed design, as it gives a first insight. Problems arise in this context mainly through stand-alone development, without taking the application environment into account. Conceptually, this requires the use of simulations at a high abstraction level, such that faulty designs can be found and respective countermeasures can be set in early design phases.

SystemC, a C++ library with different extensions was designed to cover these issues and allows to increase the granularity of systems continuously through the whole development process, without changing simulation nor development environments. Refinements on hardware/software systems are generally done in different steps, beginning from models that are synchronized via communication down to cycle-accurate descriptions. However, the original SystemC simulation kernel was not designed to simulate analog/continuous-time signals. Therefore SystemC AMS [8] was developed upon the needs to define models of physical systems. Due to the same underlying simulation principles, SystemC provides possibilities for the co-simulation of hardware/software system models and their physical environments, designed with SystemC AMS.

As already mentioned, the requirements for the development of cyber-physical systems must be seen differently. A system model has to cover the physical as well as the digital world. The developed SHARC framework is based upon an existing simulation environment for analog mixed signal models defined in SystemC AMS. This leads to the necessity for the integration of virtual hardware prototypes into the existing simulator to cover full system design and allow further software development on target platforms in early design phases.

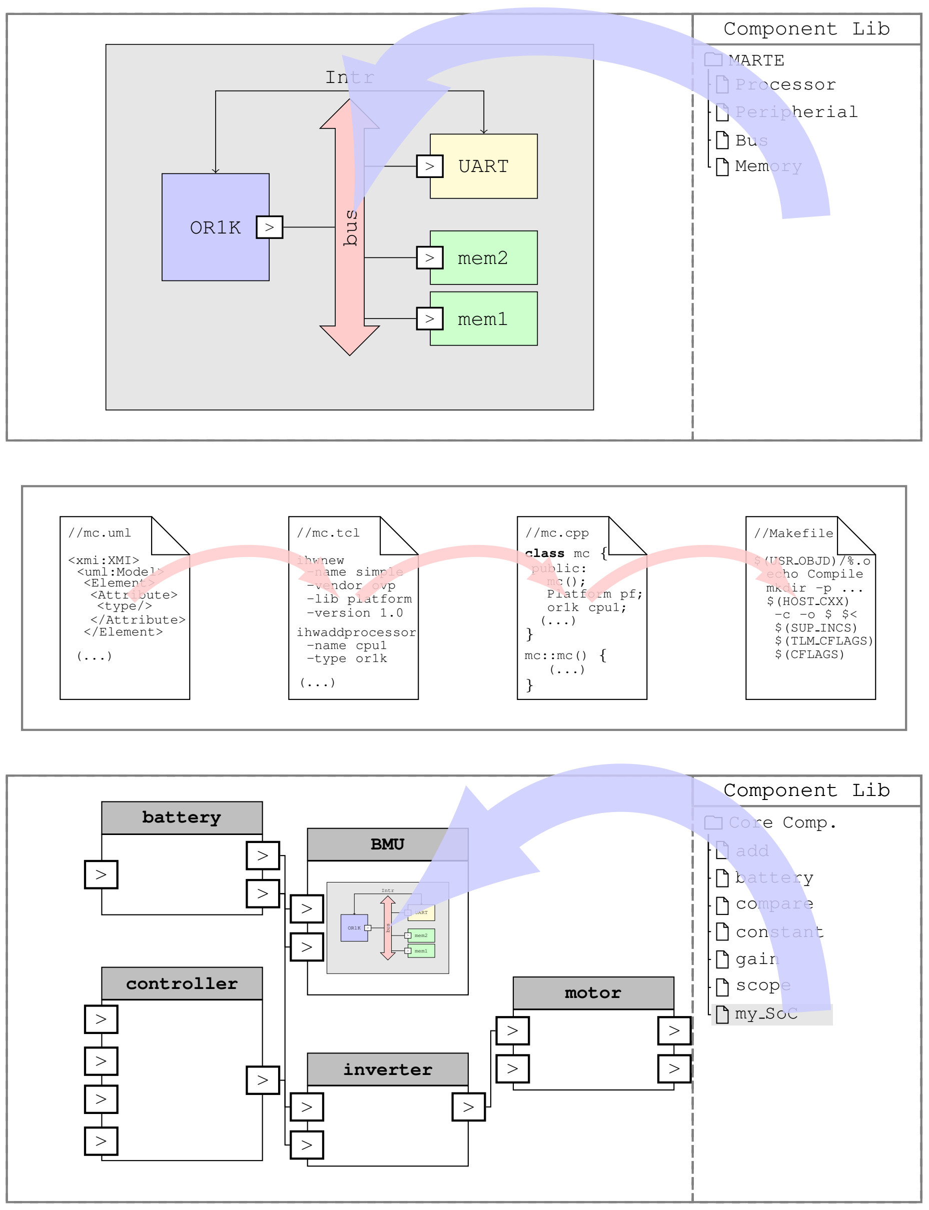
1.2 Objective

The outlined goal of the work in hand is the integration of virtual hardware prototypes into the given SystemC simulation environment, on which application software/firmware can be tested. Relying on the previous projects, similar approaches should be used to create those virtual hardware platforms. Meaning that graphical description should be sufficient to describe a so-called system on chip (SoC), which can be embedded in the simulation environment.

In contrast to the current approach, an existing library should be used which provides a large number of hardware models from different manufacturers. A further criterion is the possibility of the elegant co-simulation of digital and analog/mixed signal models. The Open Virtual Platforms (OVP) framework was chosen since it meets both criteria and provides additional tools which facilitate the platform development.

Thus the concepts to extend the used IDE, which is an Eclipse Rich Client Platform, should be developed and described in this thesis. Previous works on this project have shown how to extend the IDE, by Eclipse's plug-in mechanism. In a similar way, a toolchain should be implemented which derives necessary information of the graphical platform representations, such that it can be translated to compilable SystemC Code. For the integration of the generated platform description, the implementation of the software interface will be required. All these steps should be done automatically and lead to a representative UML block of the OVP platform, which can be integrated into the simulation platform, together with other components. A conceptional mockup of the toolchain can be seen in figure 1.1. One main challenge within this project is, however, the definition of interfaces between SystemC AMS and the OVP platform with an underlying SystemC TLM-2.0 (Transaction Level Modeling in its second version) description.

The main use case of the SHARC platform is a simplified electric vehicle (eVehicle), where the driving speed and the road load influence the system behavior. The commercial success of electric cars in the next few years depends primarily on the deployed batteries; in this context particularly the range and the durability are important. In addition, also reliability and safety have to be ensured since failures might have harmful effects on traffic occupants. The eVehicle model uses, therefore, a detailed lithium-ion battery pack, which computes internally the new state of charge, the temperature, and the module voltage. In order to guarantee safety, so-called battery management systems (BMS) are assembled, which control and monitor the behavior of the battery. A BMS is a typical application for a cyber-physical system and is, therefore, used as a use case to demonstrate and test the introduced approach.



```
//mc.uml

<xmi:XMI>
 <uml:Model>
  <Element>
   <Attribute>
    <type/>
   </Attribute>
  </Element>

 (...)
```

```
//mc.tcl

ihwnew
 -name simple
 -vendor ovp
 -lib platform
 -version 1.0
ihwaddprocessor
 -name cpu1
 -type or1k

 (...)
```

```
//mc.cpp
class mc {
 public:
   mc();
   Platform pf;
   or1k cpu1;
  (...)
}
mc::mc() {
   (...)
}
```

```
//Makefile

$(USR_OBJD)/%.o
 echo Compile
 mkdir -p ...
 $(HOST_CXX)
  -c -o $ $<
  $(SUP_INCS)
  $(TLM_CFLAGS)
  $(CFLAGS)
```

Figure 1.1: Schematic representation for the necessary steps to integrate virtual hardware platforms into the existing simulation and modeling framework.

1.3 Outline

Chapter 2 provides an insight into the most important concepts used in the context of this thesis. The term cyber-physical system is introduced first. Challenges arising in this area are shown by means of development standards in the automotive sector. Another important topic is the explanation of modeling approaches on which system simulations are carried out. Finally, this chapter concludes with the concept of hardware design for the creation of virtual hardware prototypes.

The third chapter deals with state-of-the-art methods for the development of cyber-physical systems and the need for stronger cooperations between different development groups and simulation tools. Furthermore, the previous work on this project will be examined. This includes modeling in SystemC and UML, as well as the possibilities of cloud-based verification.

Chapter 4 presents the corresponding design of the work. First, the reader will find an introduction to the system description language SystemC with the distinction in SystemC AMS and SystemC TLM-2.0. Further on, Open Virtual Platforms, the library, and simulation framework for the creation of the virtual hardware prototypes in this work is discussed. UML and its extension MARTE (Modeling and Analysis of Real Time and Embedded systems) offer possibilities for graphical modeling of the used hardware components, thus both modeling languages will be explained briefly. Finally, this chapter gives a short overview of Eclipse and its UML editor Papyrus, which was enhanced in this project via plug-ins.

Chapter 5 provides an overview of the implementation concepts of this work. One component is the mapping between graphical UML descriptions and components that can be used to describe hardware platforms. Furthermore, the reader will find a description of how those created models can be embedded in the existing simulation framework, which allows the simulation of an entire cyber-physical system model. Furthermore, a concrete example is sketched in which the granularity of an existing system is increased by the integration of communicating hardware platforms.

The sixth and final chapter provides a summary of the achievements and a comprehensive outlook on future projects that can be realized in this context.

Chapter 2

Background

This chapter opens with a discussion of the context in which the present work has been written. Thus it provides a scientific definition of the term cyber-physical system, the developmental history, as well as the staidly growing challenges in this area. These challenges are highlighted by the means of the ISO 26262, a standard for the development of electronic driving assistance systems in the automotive sector, which is becoming of greater importance. Thereafter, some development methods are shown which have proved themselves over the years in the various fields and are used in the realization of cyber-physical systems and in compliance with standards such as the ISO 26262. Due to the outlined goal, which is to integrate virtual hardware prototypes into the existing simulation framework, the terms virtual models respectively prototypes have to be explained in more detail. The chapter is concluded with an introduction to hardware descriptions on different levels of abstraction and an overview of basic components for a simple computing platform, such as processors, memories or a bus system.

2.1 Cyber-physical systems

This thesis was written as part of a project with the aim to develop a framework to enhance cyber-physical system design. Therefore definition and evolution of CPS should introduce the domain and further on illustrate the problems which arise within this field of work.

2.1.1 Definition

Through the last decades, a clear trend in the evolution of computation could be seen. In the mainframe era, people worked together with one computer, later on, this changed to the usage of workstation computers for each person in the nineties. But even during the early nineties researchers like Mark D. Weiser, chief scientist at Xerox PARC, saw a further evolution of computing, such that it will become a ubiquitous part of our everyday life. In [52] he predicted the evolution of future computing, similar to the technical development of small and efficient electric motors at the beginning of the twentieth century. In that time it got possible to give machines their own motive force, which was in contrast to the approach of using a system of shafts and pulleys which were connected to one central engine. Analog to this technical evolution he predicted:

> *Specialized elements of hardware and software, connected by wires, radio waves and infrared, will be so ubiquitous that no one will notice their presence.*

From today's perspective, his prediction should be right; and although a big media focus is set on customer devices such as smart phones, from an engineering perspective the steadily increasing amount of microelectronics, assembled in nano-world to large-scale wide-area systems are of bigger interest. Shrinking processor sizes and less power consumption give the impression of disappearance for users and this trend is crucial for the ubiquity of computer aided systems.

Due to this fact, the last years of computer engineering were characterized by different trends and key words. The term embedded system, for instance, characterizes a computer system with a dedicated application, assembled in a larger electrical or mechanical system. The range of embedded systems reaches from simple portable devices, with a single microcontroller to large complex systems, consisting of a variety of peripherals and networks, e.g. vehicles and avionics.

Another key word which arose in the last few years is the terminology of so-called cyber-physical systems, which is the combination of physical devices, known as the plant, with computers. In that sense, feedback loops, which monitor the environment affect the computations and vice versa [56, 25]. In that manner, CPS are a certain kind of embedded systems with the requirement to be real-time capable. Applications of CPS can often be found in distributed environments and its components have to fulfill requirements with respect to performance, availability, safety, and security. These properties should also hold for the communication with CPSs, which can be wire-bound or wireless [46]. A very accurate description can be found in [36], the authors boil the nature of CPS down to an essence. The way of computing in CPS is in somehow different since it has to cope mainly with the uncertainty and the noise in the dynamic physical environment. Further on failures in both domains have to be compensated. This is aggravated by the fact that CPS vary in scale and complexity. The development requires structured methods and powerful tools, such that accidents and failures can be canceled out and robust systems emerge.

A closer look at the properties of CPS shows that they cannot be associated with a certain field of research. Sztipanovits [48] claims for instance that the research on CPS has to be done at the intersection of physical, biological, engineering and information sciences. This approach implies that system designers have to cope with a large heterogeneity of different components and combine the requirements of different stakeholders.

2.1.2 The evolution of CPS

Cyber-physical systems are not an invention of the current century. Right after the development of the Z3 by Conrad Zuse, he invented a system with the purpose to survey aircraft wings. Within this computing system, measured sensor values got processed by a computing unit. In this case, data became the variables in the application program. The first CPS was born [46].

Especially aviation has played a pioneering role in the development of embedded computing. In the 60s of the past century, the so-called space race between Russia and the United States of America has set new standards. In comparison to general purpose mainframes used in business, space missions required highly reliable computing systems, designed to fulfill certain functionality, whilst minimizing weight and size. Those demands led to the design of redundant systems, as well as the formal verification [49]. Other areas which affected the development of CPS were the development of robotics and factory automation in the late 70s and 80s. Through modern communication techniques, today's focus lies on the so-called Internet of Things, with the aim to share environmental information among other CPS. This leads in general to new possibilities, since synergy effects might be useable to build systems with totally different properties such as swarm intelligence. The enormous capabilities of this area can be seen in [44], it predicts that the IoT retail market will be almost 5 times bigger in 2025 than it was 10 years before. From this short timeline perspective, one can see that we have gained lots of know-how in the field of real-time and embedded systems engineering, but this trend does not seem to stop. In fact, quite the opposite is the case, the European Union, for instance, investigates substantial funds in the development of smart factories, which relies on the idea to handle complex dynamic manufacturing networks by connecting embedded electronics. Similar funding exist also in the USA, where the US National Science Foundation (NSF) supports projects on sensor-based autonomous systems, distributed robotics, autonomous vehicles and ambient assisted living since 2006. Also the German Federal Ministry of Education and Research has made some attempts to cope with the ongoing research by setting up a cooperative study project with the German industry to develop an *Integrated research agenda Cyber-Physical Systems* [46].

As Weiser predicted [52], the development of computing goes towards ubiquity and will transform how humans interact with the surrounding environment in the future.

2.1.3 Challenges for the design of CPS

One of the most important development aspects of CPS is the field of its application. Naturally the design of CPS depends on the criticality of its application. In order to ensure the safety of CPS, standards were introduced, which must be followed by system developers. Standards have different advantages, on the one hand, they guarantee the transparency for costumers, on the other, they make errors detectable. In the case of failures, it protects companies which have provable followed the standard within their development process. One of the first safety standards was the domain-independent standard for functional safety of electrical/electronic/programmable electronic safety related systems (IEC 61508) [42]. This standard has been adapted by different domains such as railway, medical or nuclear power. One of the most important safety standards is specified by the ISO 26262, it contains the procedure for system development in the automotive field [20].

CPS are highly heterogeneous systems including mechanics as well as electronic components and corresponding software. Due to the properties of CPS, the design requires a interdisciplinary engineering approach, which makes it necessary that domain experts with different fields of expertise work together to realize a certain function. In contrast to the needs of general purpose computing, the control of dynamic systems such as physical processes requires different approaches, since they are sensitive to the feedback loop, and disturbance. Ignoring such aspects in early design phases might lead to failures and redesigns later on [31].

The research area of CPS focuses therefore on modeling approaches, which should lead to coherent designs, reusing existing components and models, as well as their analysis and verification. To cope with the mentioned issue of redesigns, the design on system level should already provide enough information to obtain the affection of the computation on the physical process and vice versa. Indispensable in this sense is the use of simulation tools and test benches, which are on the one hand capable to satisfy the needs of different stakeholders [46] and provide enough possibilities to model both environments, the physics as well as the electronics. A fundamental scientific and technological basis can lead to the final break through in the work on CPS. From an economical point of view especially time to market is crucial and has to be minimized. In addition, the development has to be in accordance with the given safety standards of the respective domain. According to [31], the central issue is to find methods, which guarantee that different project groups can work on different tasks in parallel and share their results.

2.1.4 Automotive: Example for the usage of CPS

The application of CPS is of general purpose and can be seen in the context of simple systems such as robotic lawn mowers up to systems of high complexity like power plants.

However, especially the current trend in the automotive area, like the evolution towards electric vehicles and the development of self-driving cars justifies the emphasize in this industry on good design strategies, which include all aspects discussed in in the previous section.

Evolution and trends in the automotive industry

Through the last years, one could see a clear trend in the automotive area; most of the new innovative functions were caused by electronic systems. But not only new systems follow this trend, also pure mechanical systems got replaced by interconnected mechatronic systems. One illustrative example is therefore shown in table 2.1. It highlights the evolution of a window lifter, from mechanical to a fully connected solution of electronic control units (ECUs). In this case, the ECUs can retrieve further information of other ECUs. This makes it possible to implement further features, like automatically closing windows in case of rain [45].

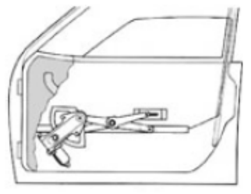
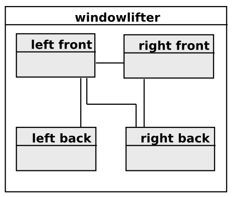
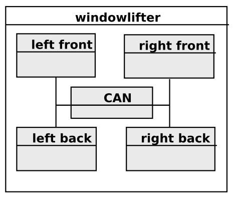
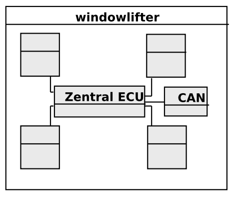
		 windowlifter left front right front CAN left back right back	
Mechnical opening and closing of the window by the use of crank.	Electric opening with switches and motors.	Electric opening with connected ECUs. Additional functionality like rain closing can be realized.	Electric opening with connected LIN slaves on central ECU. Additional functionality like rain closing can be realized.

Table 2.1: Examples for the implementation of a window lifter which is realized mechanically, electrically and electronically.

Besides the trend to mechatronic systems, almost all innovations, which make driving more convenient, rely on electronic control systems. Figure 2.1 shows that some of those additional features went through an exponential market penetration and are assembled in any car nowadays [45].

The increasing number of functions in vehicles implies the necessity to interconnect these electronics. Today's cars contain almost 100 ECUs, which build a complex computing network with various properties. But it is not only the variety of ECUs, it is also the increasing amount of transferred data, which let the industry be faced with huge challenges. From single bit transfers, used to switch gadgets on or off, communication goes towards

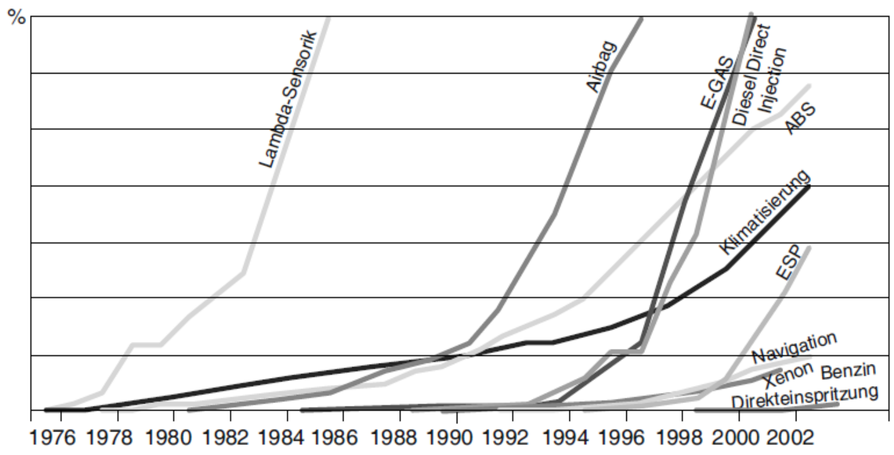


Figure 2.1: Time line for the market penetration of different electric components, which are assembled in modern vehicles [45].

streaming audio or video data, from multimedia sources or cameras. A view years ago the electronics of a car has been an encapsulated system. Especially now, in the days of IoT, intercommunication seems to be the next milestone for automotive systems. IoT requires to open systems and integrate multiple interfaces for the communication to the environment or traffic participants [45].

Through the communication to the internet, cars have also become targets for hackers. In their work on automotive security, the authors of [23] have identified and exploited potential security risks of the CAN bus. In this case, they were able to gain control over the motor controller and the braking system. Since this time, there has been an increasing number of publications in this area, which also led to many recalls by automobile manufacturers to close security gaps.

The aim of the functional safety standard ISO 26262 for road vehicles

As already mentioned, the ISO 26262 provides the essential guide for the development of electronic driving assistance systems in cars. The main goal is to define a so-called safety life cycle, which involves structured approaches that must be carried out during development, production, operation, maintenance, and disposal. The standardization is therefore divided into different parts, where 3 to 7 address the product life cycle. Part 4 is development on system level, part 5 describes approaches for development on hardware level and 6 the software development. Any development has to start with the definition of an item, which describes a tangible function of the vehicle. Having a constructive view on the item, it consists of an array of systems and their respective elements. In the case of the outlined electric window lifter, the item description would only contain the definition of the function itself. Namely, the centralized opening and closing of any window in any driving situation performed by the driver and on the other hand the automatic closing of the windows in case of rain. Furthermore, a functional concept, which contains the actuators, controllers, and sensors is designed for this purpose. Developers should also think about other interfaces, in this case, messages from a rain sensor to make decisions whether the window should be automatically closed.

In the next step, the so-called hazard analysis and risk assessment is performed. It is used to identify potential risks, arising from the defined item. In this case, scenarios are examined in which a malfunction would have serious consequences for involved parties. The hazardous event is classified by 3 different metrics: Firstly severity describes the potential damage for involved people. Secondly, the exposure of the event if it occurs and finally controllability of the hazardous event. The combination of those three assessments results in a so-called automotive safety integrity level (ASIL) for the considered event. A so-called Safety Goal is defined for the highest obtained ASIL, which refers to the function of the item itself. In the case of the window lifter, the window should not close when body parts of a person are in the opened window. There are many different constructive approaches to realize functional safety in this context.

Issues for the development of vehicle electronics

One should not forget that any development of the automotive industry is primarily driven by the price of the product and its time to market. One can imagine that the malfunction of an electronic power steering system has other consequences than a broken light in the sun visor. The development of components with a higher ASIL is always associated with a significant design overhead since such systems have to prove their full function at all times. For this reason, it is important to find design strategies, which shorten the development times and comply with the safety standard at the same time.

To guarantee comprehension, a further discussion of the development methodologies for digital systems in the automotive field is given in the following sections.

2.2 Development aspects of CPS

Especially the development of bigger systems led to the demand for structured development methods, which incorporates all sub-development steps, leading to an overall system.

Due to the fact that software has to cover very complex procedures, involving also knowledge from other domain experts, its development requires special treatment. Therefore research had been done on how to define good workflows, which bring the involved parties on one table. Nowadays it can be seen as some pioneering work, which is also adapted to other areas of system design that have to deal with similar issues. A big focus has always been on reuse and adoption of existing components, as well as defining requirements. In this sense, it is necessary to split up the overall system in sub-systems and find appropriate ways to define interfaces on which different developers, parts, and phases can rely on. Development strategies basically allocate human resources to certain activities and define the time interval in which the work should be done. Furthermore, they try to find concepts of how to define intermediate goals and how to ensure the quality of sub deliverables. For the development process, it is crucial that such methods guarantee the current status representation of the entire system.

It's quite common that the first development phase is considered to deal with the definition of system requirements. In contrast to that, the last state might have a wide range, varying from finalizing the development to the best way of disposal. Most system development methodologies rely on the principles of the waterfall model, described by Royce in the 70s of the previous century [38]. His approach shows a hierarchical structure of so-called implementation steps. Each step is finished by the definition of certain documents, which determine the starting point for the next phase. This concept includes the definition of requirements and goes towards design, coding, testing and the final delivery to customers. Another commonly used method is the so-called rapid application development, which was introduced by James Martin [29]. Martin breaks with the waterfall model and claims that almost any knowledge is gained during the implementation itself, which should be fed back to the initial requirements and the overall design. The execution/simulation of prototypes with limited functionality should indicate whether a design works or not.

V-Model

Putting an eye on the ISO 26262, the development methodology of electronic driving assistance systems is based on the V-model, which should be highlighted throughout this section. Its main idea relies on the waterfall model, but additionally, it integrates the idea of quality management, within the development phases. The bottom up approach reflects redundancy, which is a core concept of how to design safety-critical systems. Testing and verification can be seen as a way of monitoring the implementation.

The V-model is often adopted to cope with the needs of certain domains. Thus it is used in the ISO 26262 standard [20] for the system development, as well as for hardware and software development. Figure 2.2 highlights the importance of the V-model for the standard. Similar to the waterfall model the granularity becomes more accurate in the development. Even if a V-model was designed to fulfill certain properties, the central aspect of the V-model is testing. Depending on the granularity of a system, one has to consider different test types. On the lowest level of abstraction testing is done on single components, thereafter tests concern the integration of components. Further on system tests, and finally, the validation with the initially defined system requirements/specification.

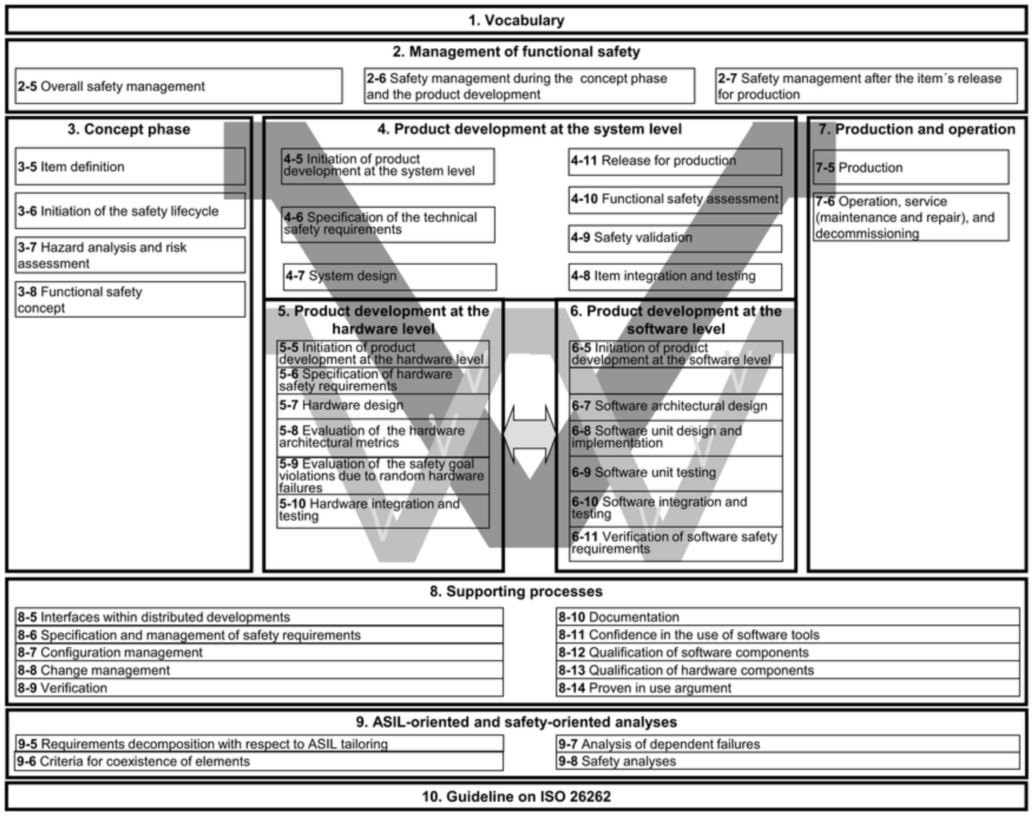

1. Vocabulary
2. Management of functional safety
2-5 Overall safety management
2-6 Safety management during the concept phase and the product development
2-7 Safety management after the item´s release for production
3. Concept phase
3-5 Item definition
3-6 Initiation of the safety lifecycle
3-7 Hazard analysis and risk assessment
3-8 Functional safety concept
4. Product development at the system level
4-5 Initiation of product development at the system level
4-6 Specification of the technical safety requirements
4-7 System design
4-11 Release for production
4-10 Functional safety assessment
4-9 Safety validation
4-8 Item integration and testing
7. Production and operation
7-5 Production
7-6 Operation, service (maintenance and repair), and decommissioning
5. Product development at the hardware level
5-5 Initiation of product development at the hardware level
5-6 Specification of hardware safety requirements
5-7 Hardware design
5-8 Evaluation of the hardware architectural metrics
5-9 Evaluation of the safety goal violations due to random hardware failures
5-10 Hardware integration and testing
6. Product development at the software level
6-5 Initiation of product development at the software level
6-7 Software architectural design
6-8 Software unit design and implementation
6-9 Software unit testing
6-10 Software integration and testing
6-11 Verification of software safety requirements
8. Supporting processes
8-5 Interfaces within distributed developments
8-6 Specification and management of safety requirements
8-7 Configuration management
8-8 Change management
8-9 Verification
8-10 Documentation
8-11 Confidence in the use of software tools
8-12 Qualification of software components
8-13 Qualification of hardware components
8-14 Proven in use argument
9. ASIL-oriented and safety-oriented analyses
9-5 Requirements decomposition with respect to ASIL tailoring
9-6 Criteria for coexistence of elements
9-7 Analysis of dependent failures
9-8 Safety analyses
10. Guideline on ISO 26262

Figure 2.2: Representation of the v-model, which plays a central role throughout the realization of systems in compliance with the ISO 26262 [20].

2.3 Models

Table 2.1 shows the schematic representation of a window lifting system, which might be assembled in modern vehicles. Although these representations vary in their working principles, they have in common that they represent an abstract view of a real system. Models play a central role in the system development, but depending on the context of the working group, they are often seen in different ways. Therefore the terminology following

Stachowiak [43] is provided:

- **Mapping**: The model is always an abstraction of an original, which might have to be constructed or remains completely imaginary.
- **Reduction**: A model specifies the necessary properties of its subject. In the model representation always some reduction of reality.
- **Pragmatic**: Models fulfill a replacement functionality which means that a model can be used instead of the original with respect to a certain purpose.

A model is a simplified representation of the reality. It allows scientists and engineers to make predictions. The main challenge for a designer is however to find a good trade-off between complexity and sufficiency, which always depends on the requirements for a certain design. Models are not only used to create designs and documentations. A common trend can be seen in their capabilities to make model-based analysis and diagnostics. In general, models are used to define dependencies and information in a real or virtual context. If relations emerge from reality, they have to be put together in the virtual environment and with an appropriate semantically interpretation [19].

Depending on the field of work, people have different understandings of the terminology and therefore two different views have to be introduced. The first term is the so-called model-driven engineering (MDE) respectively model-driven development (MDD), which is introduced as software development methodology. In contradiction to this, the terminology model-based design (MBD) describes a commonly used methodology to design control, signal processing or communication systems and has a certain importance for mechanical respectively electrical engineers. Due to the fact that the design of CPS requires an interdisciplinary cooperation, there is ongoing research on how to combine both approaches. One attempt to bring those approaches together has been presented in [53].

2.3.1 Model-driven engineering/development

Model-driven engineering is a software development methodology, with the aim to create models focusing on design perspectives. It should basically help to enhance the communication between different teams, working on the realization of a common goal. From a very abstract view, one can declare a modeling paradigm for MDE as effective, if it provides all information for developers who are in charge of the implementation [22]. Model-driven development in comparison goes even a step further, in that case, the models become essential artifacts and provide more than just an abstract design representation. Detailed model descriptions allow system/software development through model transformation techniques, such that models on a higher level of abstraction can be transformed to low-level models. The goal is to reach a description, which contains enough information to execute the model either through an interpreter or code generation [22].

Through the representation of a model on different levels of abstraction, MDE increases the automation in the program development process. In many cases, so-called graphical modeling languages, like UML, its extension MARTE or SysML (Systems Modeling Language) are used. Graphical modeling languages are extensively covered in section 4.3. The main concept of modeling languages is however the use of abstract and concrete syntax descriptions, a mapping between those and a semantic description. A description can however also be textual, this concept was used for the instantiation of the hardware platform. The respective description can be found in 4.2.3.

2.3.2 Model-based design

MBD is used in a different context than MDE/MDD and those terminologies should not get mixed. The meaning of MBD follows Aarenstrup [37].

The difference lies mainly in the context of the model itself. MBDs concern physical models which are designed for simulation purpose only, where MDE/MDD is used to describe software architectures and control flows. MBD places the model as the central component for the development of control units, signal processing, communication and other dynamic parts of the system. Therefore their working principles have to be seen differently. Models created by MBD consist mainly of basic mathematical functions, which allow the calculation of system properties at a certain time. The first challenge is, therefore, how to define the system in an abstract/algorithmic way. The second challenge is the question of how to realize the developed control units and integrate them in the real physical process. In the best case, the whole design flow can be done within one environment.

Aarenstrup provides an illustrative example of how the development of a motor controller might look like when using model-based design. Typically the overall system is split up in the plant/environment, which is a model of the engine on one hand and on the other an algorithmic description of the controller. The next step is determined by testing the high-level system with different input scenarios and verify the outputs. Furthermore, if the team is sure that the model is working correctly, more details would be added to the system and again tested continuously by simulation. This term is known as model elaboration, which should guarantee testing over the entire development process. One of the most critical parts is, however, the switch from the continuous to discrete-time, meaning the transformation from an algorithmic controller description into a hardware/software system. Increasing granularity should lead to the definition of accurate embedded software which can later be integrated into rapid prototypes and HiL (hardware in the loop) testing. Accurate models should even be able to develop production code. Casting an eye on the ISO 26262, the transformation is very difficult and can hardly be automated, especially since designed systems have to meet safety-critical requirements, which require a lot of engineering sensitivity.

2.3.3 Virtual electronic/hardware prototypes

As already mentioned in the previous section, prototyping plays a central role for MBD, which leads to the necessity of clarification. It describes the methodology to build preliminary models, in an early design phase. Those models allow first predictions of faulty designs, which would cause higher costs through redesigns. In literature often the terminologies rapid and virtual prototyping arise, which are different in their meaning. The term rapid prototyping is commonly used in production engineering since it basically describes the approach of a fast fabrication of a physical model, which is usually represented by a three-dimensional computer aided design (CAD). Due to new technology approaches, like 3D printing, it has gained a lot of attention and is mainly used in this context. Within MBD the term rapid prototyping describes mainly code generation from a virtual system description, which should be executed on real hardware [51] and tested within the real physical environment. Virtual prototypes, on the other hand, try to define pure virtual models of a system, which can be executed on a host. From its simulation, designers can draw conclusions of the current development stage. Both techniques affect almost any modern engineering approach, but since the focus of the thesis in hand was mainly set on the integration of virtual hardware prototypes into an existing simulation framework, main aspects of their development are addressed in the following section.

2.4 Modeling of digital systems

The term integrated electronic systems denotes a system which consists of two parts. These are, on the one hand, the microelectronic hardware and on the other hand the corresponding software. The hardware components are known as integrated circuits (ICs). A commonly used term in the field of CPS is the so-called System on Chip (SoC). It is an integration of different sub-components like microprocessors, memories and different kind of peripherals on a single chip. The communication of those components is typically achieved by a bus system. SoCs are commonly designed to monitor and control the plant of a CPS, a core component is, therefore, the Analog Digital Converter (ADC).

As already mentioned, besides the microelectronic hardware, the requirements for the software components increase continuously. A certain application is commonly executed on a real-time operating system, which provides necessary drivers for the communication to integrated peripherals. Because of the necessity to implement safety mechanisms, application software is getting more complex and therefore its design might exceed the hardware design significantly. Due to that fact, there is a high demand for the software development to start in early phases. As already mentioned, the common approach to cope with this issue is the use of virtual hardware prototypes.

Before we dive into details about how to model hardware platforms, the concept of abstraction in digital systems should be explained first since it is crucial for the comprehension of digital system designs.

2.4.1 Abstraction layers

One term, that is of central interest, when talking about digital hardware systems, is the term of abstraction. In the 1960s and 1970s, the development of digital systems was mainly established by the usage of functional tables with boolean equations, from which circuits could be deduced. This approach changed with the invention of so-called hardware description languages (HDLs), like VHDL and Verilog in the 80s. Instead of connecting gates, the structural behavior of the hardware is described on the so-called register transfer level (RTL). The development of electronics on the RTL is still of common interest. Through the last 10 years, new approaches emerged, however, with the aim to define layers with a higher grade of abstraction, such as electronic system level design and transaction level modeling (TLM). A common representation of those different abstraction layers is given by the Gajski-Kuhn chart shown in figure 2.3. It visualizes the properties of 3 different domains, namely structural, behavioral and timing on the different layers.

Register Transfer Level (RTL)

An RTL model describes the micro architecture of the hardware, which is the structure of clocked registers and flip-flops, as well as the unclocked combinatoric logic. In comparison to the abstraction on gate level, the time delay is not modeled, but instead only the moment when the clock changes is used. Therefore RTL models are clock cycle-accurate, meaning that a real chip would need the same amount of cycles for the execution of a certain task as the developed model. The success of RTL is mainly caused by the fact that such models can be synthesized to gate level, which can then be realized on an application-specific integrated circuit (ASIC) or field-programmable gate array (FPGA). However, Such an approach can be compared with compiling source code to a dedicated platform. Therefore it can be seen as the standard approach for hardware development.

Transaction Level Modeling (TLM)

TLM is in general not subject to such a strict timing behavior as RTL. The term 'level' might be misleading in this context and should, therefore, be seen more like a modeling

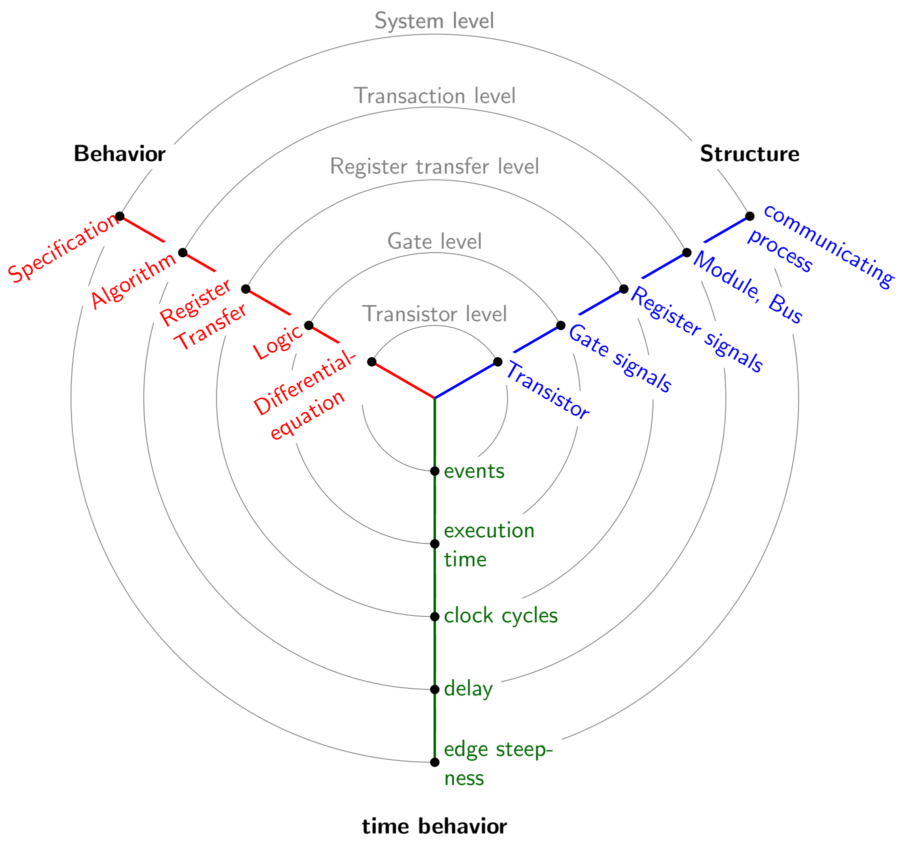


Figure 2.3: Modified Gajski-Kuhn Y-Diagramm [21].

technique [21]. TLM focuses on communication, meaning the bus connection, of data-processing units like processors, memories, and peripherals. The crucial point is the decoupling of data transaction and data processing. On RTL a bus model includes all signals in a cycle-accurate manner, TLM, on the other hand, does not have cycles. The communication is established through so-called channels, which implement callback functions. The main advantage of bus abstraction and clock decoupling can be seen in the fact that TLM systems can be simulated more efficiently than RTL systems. Modeling of user registers in peripherals is still done on TLM, this is in general accurate enough to develop application software for the platform. The rest of the functionality is still not written in the way of an RTL micro-architecture, it is more an algorithmic model which is implemented like a software function.

System Level Design

The system level is part of the design phase, where hardware and software functionality are not fully split up. More or less the main aim is to translate the specification in a model which can be independent from any time behavior [21]. This approach is called hardware/software co-design, and tries to focus on the requirements of the digital system itself. System Level Design can be seen as the transition from system to the hardware development, which is also shown in figure 2.2.

2.4.2 Execution performance

The performance of a simulation model can be verified and compared by measuring the execution time on a computer. One of the big problems in the simulation is that the program tools run on an operating system with concurrent programs. Furthermore, the simulation time depends mainly on the abstraction level of the model, resulting in larger simulation times for precise ones. The simulation time of an RTL model can differ from the execution time on real hardware by a factor of 100.000. RTL models are mostly not convenient for the development of system architectures and software, thus it is required to detach them from strict timing behavior, towards a more abstract way provided by TLM [21].

2.4.3 Components of virtual hardware platforms

The outlined goal of this thesis is the enhancement of the existing modeling framework by providing the capability to integrate computing platform into existing CPS models. As described in the previous section the simulation performance depends on the accuracy of the given model. Since we want to gain further details about the overall system and use them for a software development process, those platforms should be defined as TL-models. In compliance with Wolf [56] necessary components of a computing platform are defined as follows :

- **Processors:** Central processing units (CPUs) are defined by their instruction sets, which implement hardware functions. The software is nothing else than a sequence of instructions that can implement any kind of algorithm on a computing platform. The algorithm which established those capabilities is the so-called fetch-execute algorithm. As the name indicates, instructions are first decoded and then executed. This is the main idea and power behind modern computing, changing the instructions leads to a different behavior, without reimplementing hardware functionality. Basic computing platforms, such as the von Neumann or Harvard Architecture, consist of a CPU and one or two memories. Memories generally hold data and instructions, where the CPU itself has several internal registers, which store values, necessary for the computation. A register that is of certain interest is the program counter (PC). It is an address pointer to the instruction that should be executed next. High-performance simulations can only be achieved, when all components of a platform are modeled on a high level of abstraction. A common approach for CPU models is the usage of so-called instruction set simulators (ISS) which have the capabilities to execute the compiled binary code.

- **Bus system:** To establish a communication between CPU, memories and other peripherals a so-called bus system and respective protocols are needed. The communication between two devices is established by the so-called four-cycle handshake, which ensures that both parties are ready to transmit and receive data. In the common case, the CPU would be in the middle of any transaction, thinking of loading data from a fast I/O device into the memory, involvement of the CPU is not necessary. Direct memory access (DMA) addresses this issue and allows direct reads and writes to memory regions. Hardware components such as CPUs and DMAs can initiate bus communications and therefore they are known as bus masters. Components that can be addressed are so-called slaves. As already mentioned in the previous section, the main bottle neck, when simulating virtual hardware platforms on RTL, is the cycle-accurate communication between all components. Exactly this problem of device communication is addressed by TLM, with a higher level of abstraction.
- **Memory:** Memories are commonly organized as two-dimensional arrays, where a row and column address one bit. In computing systems especially random-access memories (RAM) are of certain interest because they can be read and written. Modern systems mainly use dynamic RAM, which exists in various types. They have the property to be fast, but decay over time, such that they have to be refreshed every few microseconds. Another commonly used memory type in the field of embedded systems is flash memory, which is a so-called read-only memory (ROM). ROMs are commonly used to store binaries and data, which are not changed by an application.
- **Input/Output Devices:** Components which are of certain interest are input and output devices (I/Os). They are necessary to interact with the environment and consist mainly of some analog parts. To establish a communication interface between I/Os and CPUs, registers are used that can be accessed by the CPU. The most common way to establish the communication between an I/O device and a processor is memory-mapped I/O. The approach of memory-mapped I/O is the usage of addresses, with which the registers of a device can be accessed. In this case the CPU can use normal reads and writes to access data. From a programmers view there are mainly two approaches of how data from I/Os can be retrieved:
    - **Busy wait** is a very naive approach for the communication between I/Os and CPUs. In this case the CPU is continuously checking the status register of the

device. This is also well-known as polling. Generally this approach is extremely inefficient and consumes lots of computing resources.

– **Interrupts** are more elegant in comparison to busy waits. Interrupts inform the CPU about the occurrence of a certain event, and forces it to execute the so-called interrupt service routine (ISR). If an interrupt occurs it has to be guaranteed that the execution of the foreground program can be continued. This is done by saving the PC and return to it after the execution of the ISR.

Chapter 3

Related work

First of all this chapter provides an overview of state-of-the-art methods for the development of CPS. The first part introduces a tool, which makes use of the MDE paradigm. The next part addresses state-of-the-art approaches for virtual prototyping, the role of SystemC, and respective solutions of leading EDA (electronic design automation) companies. Another part which has to be discussed is the term co-simulation, its importance and how it is applied to certain problems. An example of an industrial cooperation shows the need of leading simulation tools to work closer together.

Secondly, the previous work on the SHARC IDE and its simulation framework SysCore is highlighted. It combines MDE with MBD by using UML and SystemC AMS. This part should give a basic introduction to the working principles, which are used in chapter 4 and 5. The capabilities for modeling, simulation and verification are summarized.

3.1 Modeling electronics in compliance with the ISO 26262

As already mentioned, the ISO 26262 strongly restricts the methods for the design of CPS in the automotive industry and places the safety goal in the center of any development. Individual steps, which are defined in the system design, should be accessible by all project groups and define guidelines for the constructive design. Hillenbrand [19] introduces the tool PREEvision, developed by the Daimler AG and obeying the presented paradigm of MDE. Therefore a defined item is represented at different abstraction levels, the description language is EAST-ADL (Architecture Description Language), an extension of UML, developed to model electronic components in the automotive industry [11]. Figure 3.1 illustrates the schematic structure of the modeling language, which can be used to refine the system.

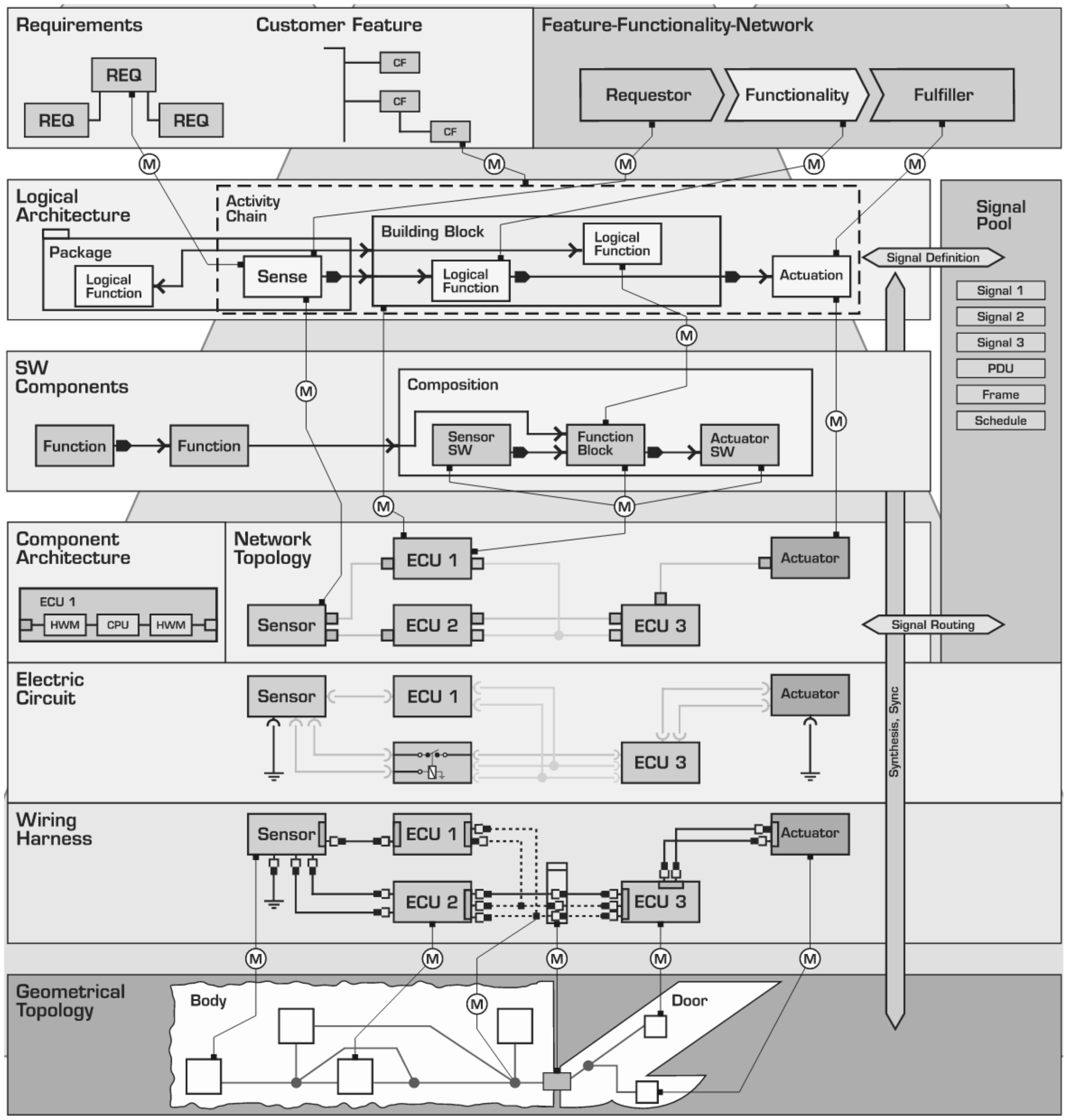


Figure 3.1: Different views on an item, defined with the modeling language EAST-ADL, which was designed to be compliant with the development requirements of ISO 26262 [19].

The different abstraction layers are described as follows:

- The **requirements layer** defines the requirements of the developed item in text form. Additionally, those descriptions can be set in relationships, which further leads to the representation of an entire network.

- **Logical Architecture** is used to create the concept of an item. In this case individual components and their connections are represented. This illustration, however, does not reveal anything about the use of hardware and software.
- With the **software components**, a concept for application software can be created. It is used for the representation of functions, communication interfaces, as well as logical and functional dependencies which arise during the implementation of the application. Communication is established via ports, which define interfaces and the corresponding communicated signals. Furthermore, this guarantees hierarchical compositions for structuring.
- The illustrated **component diagrams** refer to different layers of abstraction. Basically, there is a distinction in diagrams. Component diagrams represent the inner life of the used hardware such as CPUs, memories or peripherals. In contrast, the network topology is used to represent their logical connection. Further presentations, which are less important for the work in hand, illustrate the electric circuits, which contain the power supply and the mass concept of the electronics, as well as the realization of the wiring harness.
- The **Geometrical Topology** finally concentrates on the structural limitations, which arise when the electronics are wired in a car.

PREEvision focuses mainly on the usage of the MDE paradigm and therefore it does not include the simulation of given designs within the development environment. Although they provide some additional capabilities to combine it with simulation tools such as *Matlab/Simulink*, the issue to close the gap between different development environments remains.

3.2 State-of-the-art approaches for virtual prototyping

This section gives an insight into the development styles of the leading companies in the area of EDA, where especially time to market plays an important role to stay competitive.

Regardless which manufacturer, the main focus is therefore to develop a design flow that closes the gaps between the simulation of virtual prototypes, FPGA-based prototypes, and the corresponding test instances. Usually, there are corresponding stand-alone tools, which allow only fragments in the development. As a result, it might happen that in the elaboration of the development phase, the produced models can hardly be reused or integrated. Cadence [4] claims that the use of virtual prototypes, which have little to do with the actual hardware implementation, are responsible for poor design flows.

Today, SystemC has become de facto the standard for describing virtual TL platforms. Thus it is widely used by leading electronic manufacturers. Nevertheless, as already discussed in 2.4.1, there are some limitations for the creation of virtual prototypes since TLM descriptions are usually not synthesizable. In most cases, migration from one development stage to another has to be done by hand. Another point complicating the integration is the use of proprietary models, which can usually not be further configured by the user. This makes the step from TLM to RTL in general very difficult. To cope with this issue, the industry tries to develop open, connected and scalable solutions [4].

3.2.1 Open models

EDA manufacturers have indeed an interest in delivering open comprehensive and functional models, as they make it easier to place their products on the market. Thus it is not surprising, that libraries such as OVP gained attraction over the years and have tight cooperations with leading EDA developers. Cadence's *Virtual System Platform* tool offers, for instance, the use of OVP models, which are of certain interest for the thesis in hand. The use of standards guarantees models which have been appropriately reinforced by different manufacturers. Furthermore, vendors guarantee that the corresponding hardware realization exists.

3.2.2 Graphic modeling

The tool *Vista Virtual Prototyping* from Mentor [30] allows the creation of a virtual hardware prototype by deriving it from a graphical description. Therefore a library of standard TLM models is used, whose description can further be modified, if necessary. The created platform can be used for the development of application software. Using a visual description offers the possibility for an intuitive creation by combining function blocks.

Similar to the approach of integrating entire OVP platforms in Cadence's Virtual System Platform tool, Mentor provides finished TLM platforms which can be executed off-the-shelf.

3.3 Co-simulation CPS and SoC models

Co-simulation is one of the most important concepts for the development of CPS and attempts to simulate an entire system by simulating its subsystems. The main issue for modeling is the fact that is most likely done without taking the surrounding environment into account. In the actual co-simulation, communication between two systems takes place via data exchange. In this context, the use of different, often also proprietary simulation mechanisms, makes the connection of subsystems difficult [16]. Especially so-called hybrid co-simulation is of certain interest since it has to couple the simulation of discrete events (DE) with continuous-time (CT). To combine both, it is always essential to translate the respective signal into the other modeling approach. As described in [14] mainly appropriate scheduling of the co-simulation engine is responsible for the progress in time and the respective data exchange between discrete and physical world.

One interesting standard, which is worth mentioning, is the Functional Mockup Interface (FMI) [3]. It implements an additional layer that enables data exchange between simulation tools. But at the same time it protects the intellectual property (IP) of the model itself. FMI exist for different programming languages such as Java or Python as well as for dynamic modeling tools such as *Simulink* or *Matlab*.

3.3.1 Industrial example - Mathworks and Cadence

The importance of co-simulation is also highlighted with the announcement for a stronger cooperation between Matworks and Cadence in November 2016 [47]. Together they worked on an integration of the Cadence *PSPICE* (Personal Simulation Program with Integrated Circuit Emphasis) simulator with *Matlab* and *Simulink*, which enhances the PCB (printed circuit board) design. *Simulink* is the state-of-the-art tool to simulate physical and dynamic systems using the described MBD approach, whereas *PSPICE* is used to develop mixed signal electrical and electronic circuits.

Simulating both parts together in an integrated environment has the advantage to obtain results faster with fewer errors since input and output affect the other subsystem in the simulation. In general, the integration works on both sides, such that one can either use co-simulation within *Simulink* by integrating *PSPICE* models or by exporting behavioral models from Simulink through code generation and integrate them in *PSPICE*. This approach enhances also the verification methodology since stimuli comply with the conditions under which the system is working.

3.3.2 Conclusion

Gomes et.al. [16] showed that co-simulation depends very much on the context in which it is used. *PSPICE*, for instance, is based on SPICE (Simulation Program with Integrated Circuit Emphasis), which is the most popular analog circuit simulator. Whereas especially Verilog and VHDL are used for development of digital circuits. The cooperation of such companies highlights the demand for integrated solutions to develop CPS. Nevertheless, this proprietary tool cooperation is only important for the final realization of a certain part, namely the interaction between the plant and an analog circuit.

From this example, it can be seen that the gap between individual development approaches diverges. Therefore, especially in terms of simulation on a high level of abstraction, one has to find a way to model the environment, digital and analog circuits as well as corresponding software in a sufficient manner. SystemC and its extension SystemC AMS were designed to cover this demand. Because of its capabilities to simulate physical as well as analog and digital designs especially on a higher level of abstraction it has been used for the realization of this thesis. Further descriptions can be found in chapter 4.

3.4 SHARC IDE

One of the declared objectives of this project was the combination of the two paradigms, model-driven engineering, and model-based design as described in section 2.3. This section shows how systems with the developed SHARC IDE can be designed in a graphical way. Furthermore, the basic concepts of the simulation are introduced. These approaches provide the possibilities for simulation-based verification in the cloud [39].

These concepts provide a solid base to integrate hardware/software co-simulation into the existing SHARC IDE. An overview of main concepts of SHARC is given in the following sections.

3.4.1 Modeling with UML

UML plays a crucial role in the developed modeling framework since it is used to represent a modeled system design. At this point, it should be mentioned that UML is also of central importance for the implementation of the work in hand, and thus further explained in 4.3.1. However, this section provides an introduction to the previous achievements of SHARC.

SHARC's capabilities to describe system designs can be compared with the software components respectively with the logical architecture layer in figure 3.1. Designs are generally created in a container class, where function blocks are instantiated as UML properties. Similar to the software Components, designs contain communication interfaces, which are represented as MARTE ports. Further on, linking connectors denote communication channels between those properties. Figure 3.2 shows a graphical representation of an electric vehicle, which has been the main use case for previous projects and the work in hand. An important concept is the representation of properties as hierarchical compositions. Motor and converter are for instance modeled in another separated class, which means that it consists of respective functional blocks/properties and their connections. Such designs can be used as subsystems, which is the case in the given example.

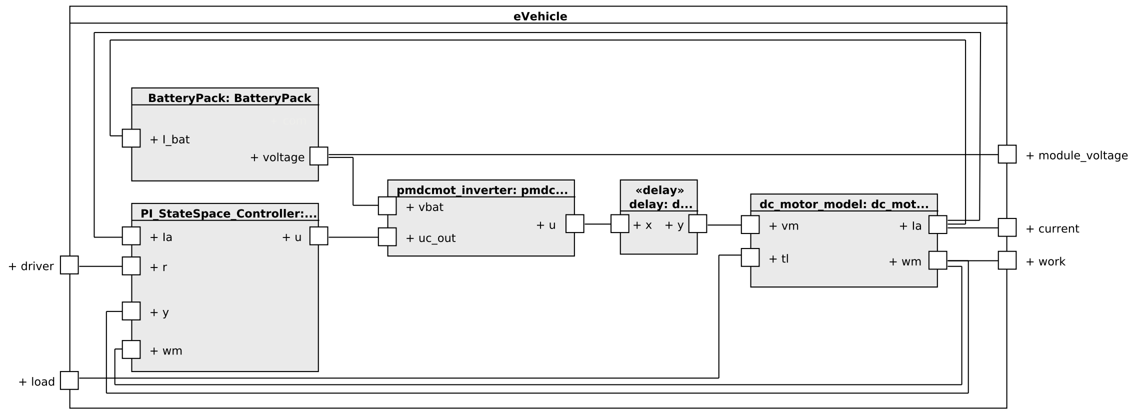
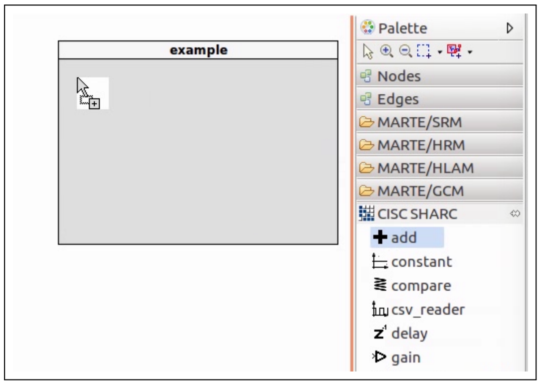


Figure 3.2: Electric vehicle modeled with the SHARC IDE. System components are connected through their ports. Inverter and battery are implemented as functional blocks, whereas controller and motor are subsystems, which refer to other compositions.

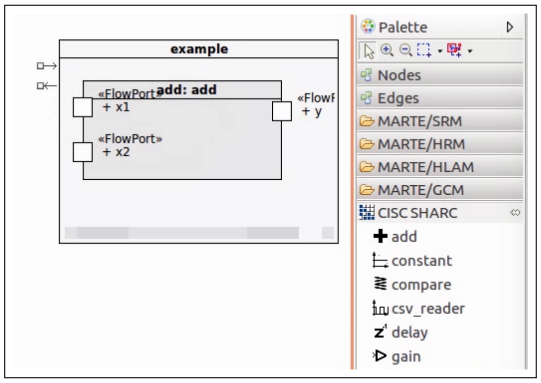
3.4.2 The SysCore simulation framework

As already mentioned, from an engineering perspective model-based-design is a common way for the development of complex systems. Due to this fact, we combined MDE with MBD in a sense that the graphical UML representation can be simulated. The graphical description leads to a modular representation of the overall system design, which contains the main information about the signal flow. For the simulation, it is necessary to equip function blocks with their algorithmic/mathematic descriptions. As described in chapter 4, SystemC AMS is used to implement the real functionality of used function blocks. Without going into further detail, each instantiable UML block has an underlying SystemC AMS implementation, compiled to a shared object (.so)/dynamic link library (.dll). Execution with the so-called SysCore simulator requires, therefore, a UML file as a starting point, which contains the graphical description of the system design. After parsing the content of the UML file, respective components are instantiated and simulated. With SysCore, modeled systems can be executed in a very early design phase, which makes it possible to discover faults as soon as they arise [53]. The implementation of core components for SysCore is further addressed in section 4.1.6

To enhance the usability of our implementation, we have developed plug-ins for the used Eclipse IDE to make the instantiation of predefined components more convenient when inserting library components from the palette, as shown in figure 3.3.



(a) Class appearance while inserting a component.



(b) Class appearance after the insertion of a component.

Figure 3.3: Instantiating a SysCore component from the palette.

3.4.3 System verification

As already described in section 2.3.2, verification and testing plays a central role in the system development process and shall therefore be discussed in the following section.

UML test instances

In previous works, it has been shown how to automatically create instances to test the created design [54]. These instances were in general designed to comply with standards such as the Universal Verification Methodology (UVM) [1], which can be used to define test scenarios that cover certain corner cases. The basic idea for the test instance relies again on the use of compositions, such that the given design is instantiated as property within a testbench class and used as the design under test (DUT). Further CSV readers for input stimuli and respective output monitors are created within the new testbench class. The result is a testbench model which consists of the outlined components and their according connections to the DUT. An example for such a test instance can be seen in figure 3.4.

As described above, it is important to define different scenarios which have to cover many different input combinations for the DUT. In general, there are two different approaches which affect the behavior of the system:

- **Internal parameters** allow the utilization of used function blocks as described previously. Therefore it might be necessary to vary those parameters to deliberately inject faults into the design under test. In the best case, such faults can be tolerated by the designed system.

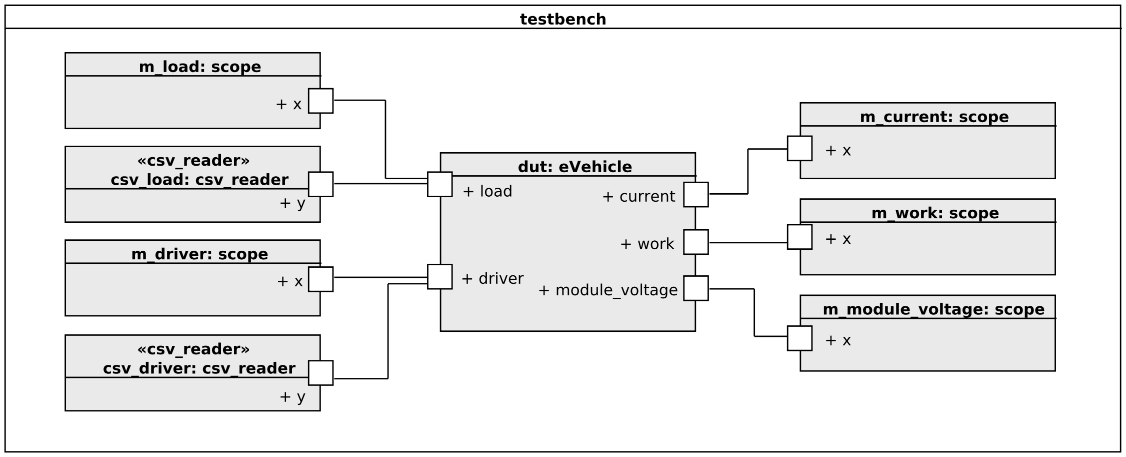


Figure 3.4: Automatically generated testbench, containing DUT, drivers and monitors.

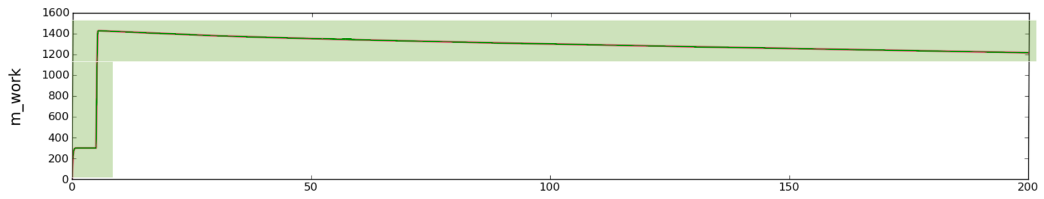
- **Input stimuli** are external signals which drive the DUT. In case of the simulation framework, those signals are defined as discretized mathematical functions, through the usage of `.csv` files, which define `<time - value>` tuples. Signals can be defined through the usage of `csv_reader`s, which are part of the SysCore library.

The difficulty for the verification can be explained by the characteristics of analog mixed signals. Thus output values can not be compared with a reference model since exact coverage can not be obtained. Moreover, a correct system functionality can only be defined by satisfying limited boundaries, as shown in figure 3.5. In case of the modeled eVehicle, it is, for instance, possible to monitor the behavior of the battery voltage. Such that fault scenarios can be observed with an implemented `SuccessValidator`. It basically compares an output signal with a reference signal and observes whether it exceeds the defined boundaries during the whole simulation duration.

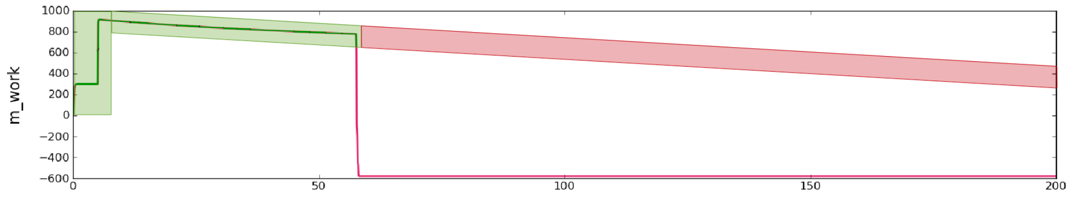
Simulation in the cloud

As mentioned above, the overall result of a simulated sequence does not affect other configurations in any way, which leads to the possibility of parallel execution of sequences. From this point, a cloud-based approach was developed. Without going into further details about task distribution to different worker instances, it should shortly be mentioned how parallel simulation is established:

- **Archive file:** Since a simulation is processed on a worker instance, the created .uml file bundle has to be sent to a worker instance. The package is simply sent to a webserver, on which worker instances can access it, whenever necessary. This avoids



(a) The scenario passes since boundaries are not violated.



(b) The scenario fails since the trace exceeds its predefined boundaries.

Figure 3.5: Examples for the observation of traces.

needless traffic, as well as it leads to a transparent storage of task descriptions.

- **Configuration file:** Secondly a user defined amount of configurations is created, which utilizes the simulation of a worker. configuration files are stored in a queue that is processed by a master instance. Through the RabbitMQ framework, it is possible to launch the simulation on the remaining worker instances. The configuration file is passed to the simulation core and utilizes duration, time step, internal values and the input signals.

After processing a certain simulation, its result can be seen from a web interface.

Chapter 4

Design

This chapter consists of a basic description of the components and concepts which are used in this thesis. A special focus is set on the capabilities of SystemC TLM-2.0 and SystemC AMS. This should provide the fundamentals to understand the core concepts of how to embed OVP models in the existing SysCore simulation framework. OVPs core technologies, the usage with TLM-2.0 and TCL (tool control language) respectively iGen, OVP's language for component and platform descriptions, are mainly discussed throughout this section. Furthermore this chapter outlines the basic concepts of UML and its extensions SysML and MARTE. The chapter is concluded with a discussion about the used Eclipse IDE and the possibility to write plug-ins for the purpose of extension. Moreover, this leads to the introduction of Papyrus, which is the used UML editor of the SHARC IDE.

4.1 The system-level modeling language SystemC

The work on SystemC was coordinated by the Open SystemC Initiative and was a collaboration of several renowned electronics and semiconductor manufacturers. Since 2005 SystemC has its own IEEE standard and is now supported by the Accellera Systems Initiative [2]. Because of this standardization, SystemC has gained more and more attraction trough the last decade, which is also caused by its abilities to cover a wide range of abstraction layers. One further advantage over other hardware description languages is the widespread use of C as a programming language for low-level applications, such that developers do not have to learn a new syntax and first system mockups can be done in a single environment [17].

Through the last years, the demand for a tighter interaction between embedded hardware/software systems and the environment in which it is embedded has steadily increased.

SystemC enables system designers various options for the refinement of hardware/software systems, beginning from general synchronization via communication, known as TLM, down to cycle-accurate RTL modeling. The original SystemC simulation core was nevertheless not designed to simulate analog/continuous-time. SystemC AMS was developed to supply the needs of the described issue and was built on the SystemC standard [2].

As already discussed in 3.4.2, SystemC's extension SystemC AMS is used to describe the functionality of the SysCore library components on the one hand, but on the other hand, SystemC and especially SystemC TLM-2.0 can be used for hardware description, which is the intention of the work in hand. Thus SystemC is very promising for any TLM description and the co-simulation between the analog and digital world, which makes it very important to understand the core concepts of the language.

4.1.1 SystemC's core components

This section summarizes the core components of the SystemC library. The descriptions refer to the explanations of Kesel [21]. SystemC is based on the programming language C++ and contains an additional simulation core, which is basically a scheduler. The scheduler controls the execution of concurrent threads and methods, which are similar to processes in other HDLs, like VHDL. In comparison to VHDL, compiled SystemC models include the simulation core, which means that further tools for the execution are not required. Figure 4.1 outlines library components, which are necessary to define the properties for RTL modeling, furthermore they provide the properties of TL-modeling, which is not possible with classic HDLs. Those components are shortly described in this section.

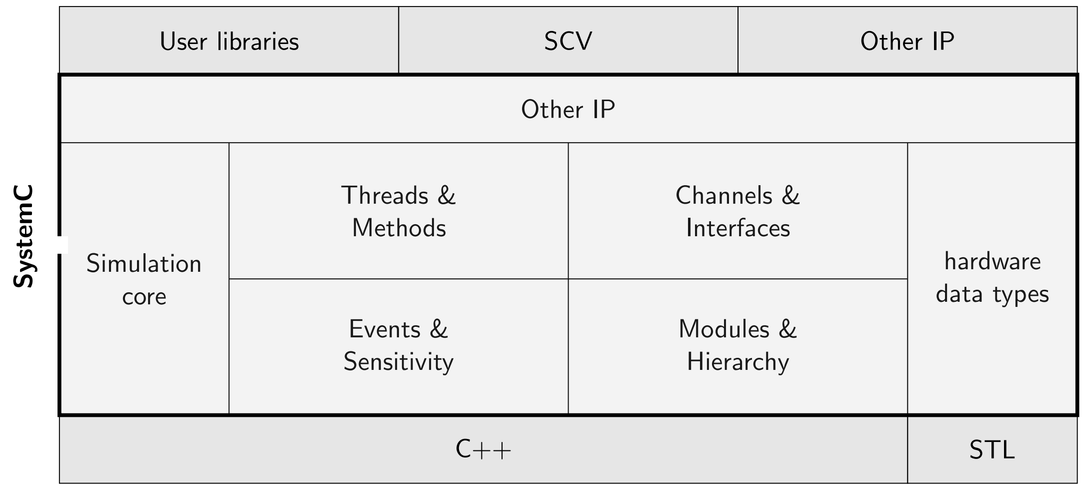


Figure 4.1: Architecture of SystemC's core components, as described in [9].

Threads and Methods

Threads and Methods are the counterpart of processes in other HDLs, which are necessary to establish the parallel execution of hardware components. A method is part of a module and implements its functionality. Processes are sensitive to events and can be distinguished in the following way:

- `SC_METHOD`s are entirely executed, without any possibility to stop within the code. Only after the execution, the control returns to the simulator. The decision when a process gets executed is done with the sensitivity list, which is created after process registration.
- `SC_THREAD`s differ from `SC_METHOD`s in the way they are handled by the simulator. They are started only once at the beginning of the simulation but can be suspended during their execution by calling the `wait()` method.

Events and Sensitivity

SystemC is, like other HDLs, based upon events, such as reacting on the positive edge of a clock signal. This approach allows the parallel execution of modules/processes and their synchronization. The simulation core reacts on the sensitivity list and the registered events.

Channels, Ports and Interfaces

Similar to VHDL, SystemC uses ports for the connection between modules. On RTL the simplest form of a communication can be established by the connection of ports with a signal. Ports are always associated with a direction, which can be in, out or bidirectional. Ports are always bounded to signals instantiated on a higher level.

But due to its TL modeling capabilities, SystemC defines more general approaches for the communication than common HDLs. One concept is the definition of so-called interfaces, which are class definitions that can be used to implement further functionality for a port. Another concept of SystemC are two different types of are so-called channels:

- Examples for **primitive channels** are the already mentioned signals but also, clocks, buffers, fifos, mutexes, and semaphores. Although they implement different functionality, they have in common that they are derived from the `sc_prim_channel`, which implements a deterministic request-update mechanism. Those channels are commonly used for RTL modeling in SystemC.

- **Hierarchical channels** are the main reason for SystemC's possibilities of TL modeling and allows the implementation of entire on-chip bus systems. In comparison to primitive channels there are no predefined hierarchical channels in the original SystemC library, but in the TLM-2.0 extension.

Modules and Hierarchy

Modules are the common way of how models can be structured in SystemC. This is analog to Verilog and VHDL and allows for definitions of hierarchical compositions, which makes it easier to reuse predefined models. Due to the object-oriented programming (OOP) properties of C++, it is obvious to rely on the class mechanism. Typically components of modules are ports, signals, objects of other modules, methods that describe the components behavior and the constructor, which is e.g. responsible to register processes.

Hardware data types

To cope with the requirements of typical HDLs, SystemC defines so-called hardware data types, which are mainly used to define a low-level connection of single components and processes. Hardware data types include for instance the definition of single bits, that can be used to model a bus system or different states of a signal.

4.1.2 SystemC's simulation core

The initial SystemC library was defined as open source, proof of concept library on which different stakeholders worked together to model their IPs.

With the standardization [2] a final reference simulator was published by the Open SystemC Initiative (OSCI). However, many semiconductor companies provide their own proprietary tools for the co-simulation of developed components. For the introduction of the simulation core, this thesis relies on the description of Kesel [21] and highlights the common simulation algorithm without going into further details. As already mentioned, the main property of the SystemC simulator can be seen in scheduling processes, such that they are executed in a deterministic manner.

Similar to VHDL the simulation of SystemC is also done event-driven. The outlined goal of a simulation algorithm is to be efficient and fast in its execution, which leads to the necessity to find a certain abstraction in comparison to the real physical behavior. RTL modeling is time and value discrete. Especially the term time discrete can be exploited by simulators since changes have only to be done at certain points. This is in contradiction to time continuous events, where finding a solution for a certain time can only be done by solving differential equations, which describe the given system. The abstract change of the signal value is called event. Within the simulation, values only have to be recalculated if event changes occur.

As already mentioned, modeling electronic components requires the simulation of concurrent processes. However those processes can not be simulated in a parallel manner, they have to be executed sequentially. The communication requires a mechanism that guarantees the right execution of processes and to supply values if they are requested from a certain process.

Assigning values is not done while execution, they are only pre-registered. The actual assignment is done after all processes have been executed, which is known as evaluation phase.

The main task of the scheduling algorithm is the execution of the user defined processes. It is split up into 5 phases which are shown in figure 4.2:

- The first step in the **Initialization** phase is the assignment of signals and the preparation for the execution of processes. This approach guarantees that each process is executed at least once. This is, for instance, the starting point of thread processes.
- The **Evaluation** phase contains the unsorted execution of the processes. When writing values to signals, this is registered by the simulation with a "update-request" on the signal. If all processes got simulated, the update phase is executed.
- Within the **Update** phase, all update requests are executed. In the case of value changes for a signal, so-called Delta notifications events are triggered, which are executed in the next phase.
- The **Delta-Notification** phase processes the previously defined events and marks the processes which are sensitive to the changes at execution. If the set of marked processes is not empty, those processes are executed in the evaluation phase again, without changing the simulation time. This is called delta cycle and is done until the set of marked processes is empty.
- **Time-Notification** is a leap in time to the timed event, which effects a process. Thereafter this process is executed in the evaluation phase again.

4.1.3 SystemC TLM-2.0

The previously described core components are mainly used to define RTL hardware models. Nevertheless, SystemC exceeds the ordinary capabilities of hardware description languages like VHDL and Verilog, through its mechanism of hierarchical channels, event objects, and dynamic processes. Thus SystemC allows to raise the level of abstraction to design complex

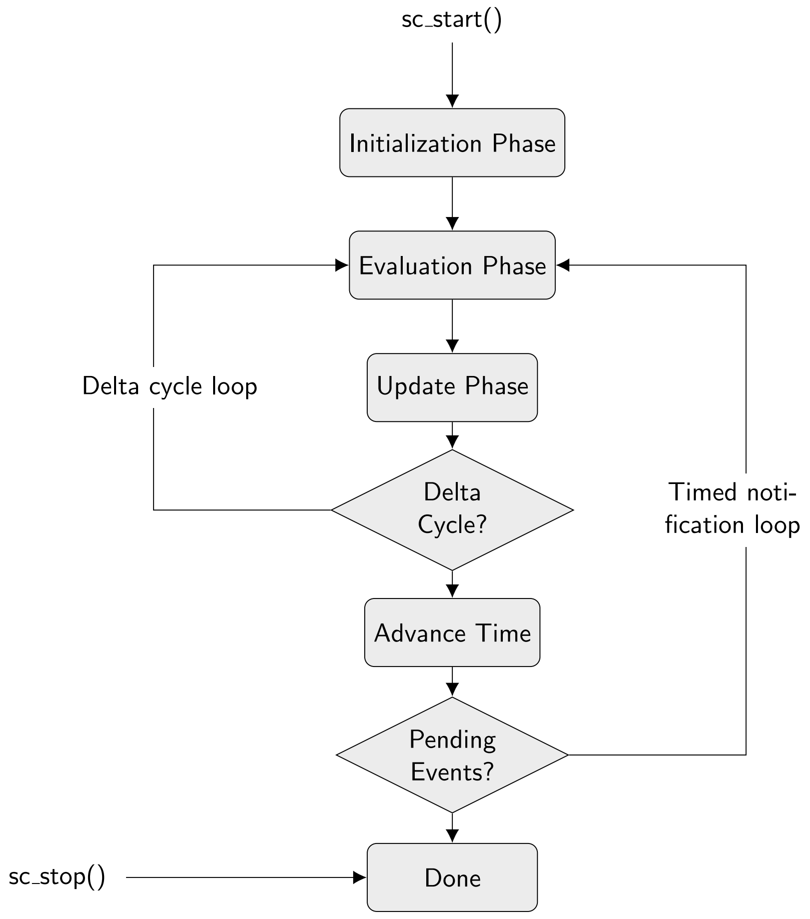


Figure 4.2: State chart of the SystemC scheduler [9]

virtual hardware prototypes, together with software components, like real-time operating systems and drivers for peripherals, as described in [21].

Execution on Register Transfer Level, no matter if VHDL, SystemC or Verilog is time intense, since various signals and processes have to be simulated together to fulfill cycle accuracy. TLM circumvents this overhead, through a higher level of abstraction within the communication of different components. To establish this requirement, hardware signals and ports are not modeled cycle-accurate in TLM. For instance, one bus transaction within a RTL model consists of various clock cycles, which cause big performance issues. The TLM-2.0 library extension was mainly developed to cope with this problems and establishes high simulation performances by the use of fast bus transactions. This means that instead of using transactions through signals, the communication is replaced by interfaces and sockets. A further advantage is interoperability, in such a way that the usage of certain bus interfaces, sockets, and a generic payload guarantee that different components, which implement the interface, can be connected and simulated as an entire system. Furthermore, there are no restrictions for the models of components. They can be designed in a detailed manner, on RTL or very abstract by the pure definition of their functionality, such as instruction set simulators for processors or algorithmic models of peripherals. Due to this fact, different vendors provide TLM-2.0 compatible models/IPs. The open source framework used in this thesis is called Open Virtual Platform. Besides this open source approach different commercial products have been developed by different hardware vendors such as Mentor, Synopsys or Cadence.

As shown in figure 4.3 the SystemC TLM-2.0 standard enables 2 different 'modeling/coding styles', which are always a trade-off between simulation accuracy and speed [12].

- **Loosely timed modeling** allows the annotation of a time delay for a transaction, which means that an initiator process can run ahead of time, compared to the target process. To accomplish this behavior, a so-called payload event queue (PEQ) is used, which stores the events accordingly. The performance speedup can be explained with a reduction of context switches, which reduce the simulation overhead. The time warp is organized through a so-called quantum keeper that keeps track of the local times for each module.
- **Approximately timed modeling** describes the concept of time-coupling, such that all processes run in lockstep with the global simulation time. Due to the fact that one transaction consists of various function calls, this approach is almost cycle-accurate. Approximately timed modeling is mainly used to determine the performance of a bus system or a system design.

The crucial part of the TLM-2.0 library is, however, the usage of **interfaces** and **sockets**, as well as the **generic payload**. Therefore they should be introduced briefly. Furthermore, design aspects of how to model a TLM-2.0 system should give an idea of how those components are typically used and partitioned.

Interfaces and Sockets

As previously mentioned, one of the TLM-2.0 library's core concepts is the use of sockets and interface methods. Therefore one has to distinguish between the initiator socket and the target socket, which are coupled to a forward and a backward path. The forward path can be used from the initiator socket to call an interface method of the target. The target socket has the possibility to call interface methods of the initiator. The socket is always a combination of a port with an export, which enables the communication between both of them. The communication always involves an instance of a so-called `tlm_generic_payload`

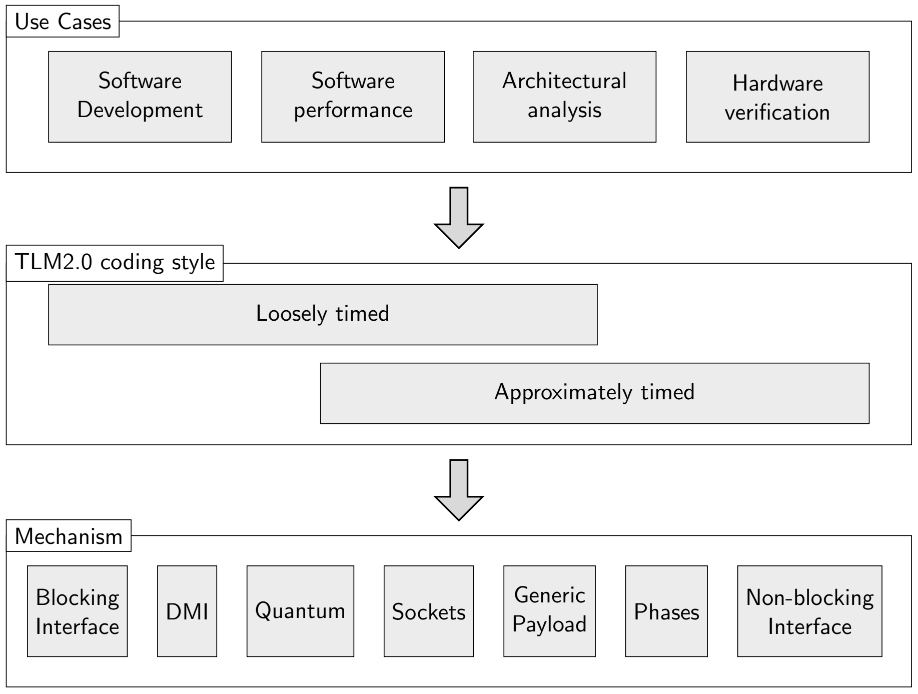


Figure 4.3: Architectural design of the SytemC TLM-2.0 standard [6].

object, which is passed to the transport interface method and indicates which operation should be executed on the receiver side.

Generic payload

The generic payload is a standard class, part of the SystemC TLM-2.0 library. Together with interfaces and sockets, it provides the capabilities to model a bus system, which works accordingly to the memory-mapped I/O principle. In general, the transaction object is a member of the initiator. Its reference is passed, when an interface method of the target is called. The transaction object has several attributes/members such as addresses, data and byte enable, which allow the implementation of single transfers, burst or the reply of the target.

4.1.4 System design with SystemC TLM-2.0

Generally, it is recommended to divide components of a TLM-2.0 system into different modules [21]. A typical partition would be in computation units, such as the previously described initiators and targets and the connection of those units, which is typically established by a bus system, as so-called interconnect or router. One simple design, which consists of these basic modules, is shown in figure 4.4 :

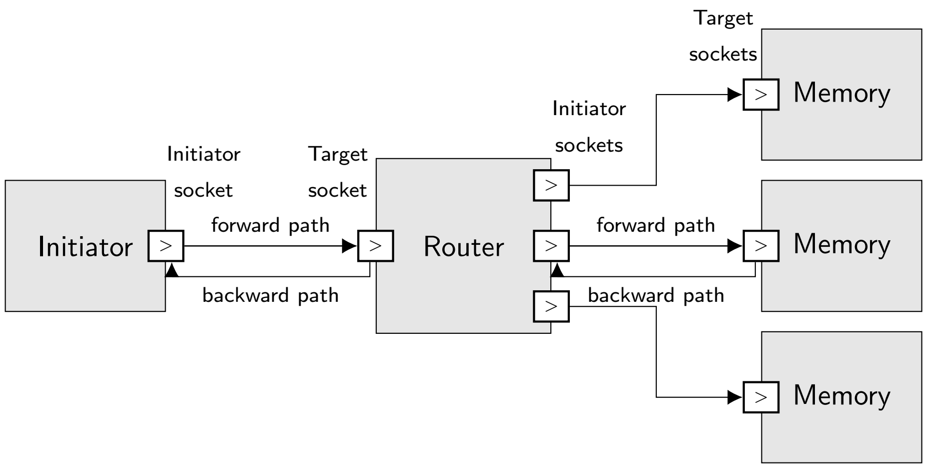


Figure 4.4: Design of a simple TLM-2.0 architecture, taken from [5]. It shows forward and backward paths for the communication between initiators and targets.

- **Initiators** are SystemC modules which are able to initialize transactions, by instantiating a transaction object and passing it to the target over the interconnection component. The initiator contains always an initiator socket.
- **Targets** obey an initiator and response in certain cases. The initiator is able to read from or write to the target via an interface method. A target has always at least one target socket.
- The **interconnect/router** module establishes the communication between initiators and targets. It calls interface methods and forwards the transaction object, by decoding target addresses. The interconnect module includes both, initiator sockets and target sockets.

Data processing can be realized as a pure C++ function, embedded in a so-called communication wrapper. The communication wrapper is then implemented as a SystemC module. Beside the computation component, it contains the TLM sockets and interfaces, which are used for the communication. One main advantage of this method is that the interfaces for the computation components are rather simple, which means that designers do not have to consider the complex TLM-2.0 communication mechanism once the implementation of the wrapper has been established.

4.1.5 SystemC AMS

The previous section has introduced SystemC's capabilities as a modeling language for virtual hardware prototypes. The original event driven simulation core of SystemC was nevertheless not designed to simulate analog/continuous time. As described in chapter 2, through the last years the demand for a tighter interaction between embedded hardware/-software systems and their environment has steadily increased. Therefore the SystemC AMS library was developed to cope with this issues. It is built upon the SystemC libraries and its simulation core [8].

SystemC AMS's architecture

The model abstraction of SystemC AMS is based on commonly used methods for the design of analog and mixed signal systems. SystemC AMS supports functional modeling at different abstraction levels, thus 3 different models of computations (MoCs) are used, namely Timed Data Flow (TDF), Linear Signal Flow (LSF) and Electrical Linear Networks (ELN).

As shown in figure 4.5, SystemC AMS makes a distinction between discrete-time and continuous-time behavior as well as non-conservative and conservative descriptions. Non-conservative descriptions are in general less complex and sufficient for system-level modeling. Therefore only TDF and LSF were used to model SysCore library components. The difference between those two MoCs is the time behavior, which is discrete for TDF and continuous for LSF. The so-called enabling technologies contain a linear DAE (Differential-Algebraic Equation) solver and a scheduler, which are used for the simulation of either LSF of TDF. Furthermore, the SystemC AMS makes use of a synchronization layer, it enables the simulation of models that are implemented with different MoCs.

In the following TDF and LSF are introduced by showing examples, which were implemented as part of the work on the SHARC IDE. Furthermore, this should give an idea about the difference in modeling and simulation within these approaches.

Timed Data Flow (TDF)

TDF modules are built upon the same modeling formalism as ordinary SystemC modules and therefore they are somehow similar in their implementation. A TDF module contains a `processing()` function, which can be compared with a process in bare SystemC. The only distinction is that TDF `processing()` functions are not called event driven as in bare SystemC. Signals are sampled in a time discrete manner, meaning periodically with a

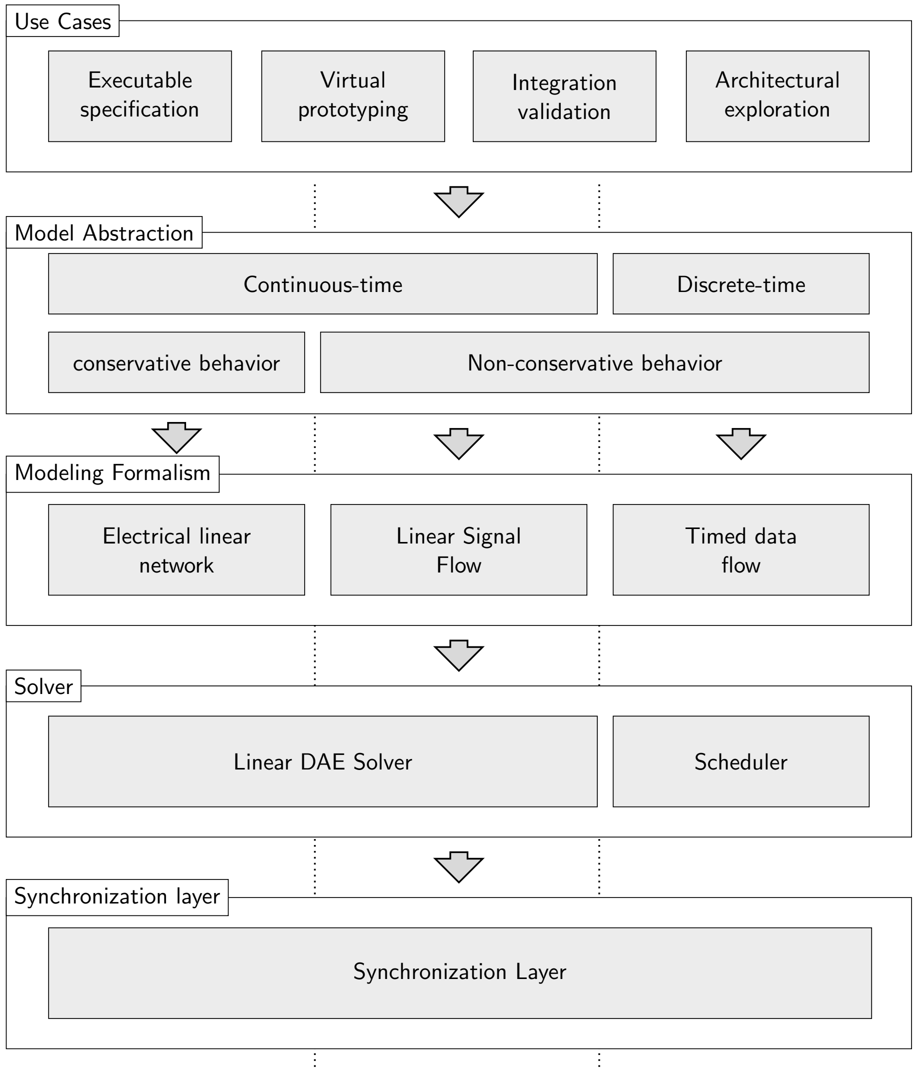


Figure 4.5: Schematic representation of SystemC AMS's core components and their relation [8].

fixed time step. The simulation core can define a fixed schedule for the function call before execution.

TDF models consist typically of connected TDF modules. Similar to RTL modeling, the connection is established with the definition of signals between input and output ports of a module. The created directed graph is known as TDF cluster, modules form the vertices and signals the edges. In that sense, a cluster defines a mathematical composition,

```
SCA_TDF_MODULE( tdf_stop ) {
public:
  sca_tdf::sca_in<double> in;

  tdf_stop(sc_module_name name): sca_module(name){}

  void initialize(){}

  void set_attributes(){}

  void processing() {
    if(in.read()!=0) {sc_stop();}
  }
};
```

Listing 4.1: The example is taken from the SHARC framework and highlights the typical structure of a TDF module.

including the internal functions of the instantiated TDF modules. The processing within a module depends always on the number of samples Input values are used to perform computation within the processing method, the obtained results are written to the output afterwards. The numbers of reads and writes are however predefined for the simulation. In TDF, delays which are commonly used to make the static scheduling feasible are defined during the simulation elaboration.

Listing 4.1 shows an implementation of a simple TDF module, more specific a sink. The module makes use of a TDF input port and contains the mandatory declaration of the constructor. The member functions `initialize()` and `set_attributes()` are optional and called by the simulation core in the elaboration phase. The method of interest is `processing()`, which encapsulates the implementation of the signal processing function (i.e. stopping if the input signal is not 0).

Linear Signal Flow (LSF)

In comparison to TDF, MoCs defined with LSF allow for an analytical solution to a given design. LSF modeling is done by setting predefined library components into relations. During the simulation, outputs are computed by a linear DAE solver. Again, ports and signals are used to interconnect components. A fully connected system is shown in figure 4.6, it is a so-called LSF cluster from which the simulation core can derive equivalent mathematical equations. Input and output ports allow the connections to other modules, which do not have to be necessarily modeled with LSF.

It might be necessary to assign a time step for the simulation. In case LSF models are

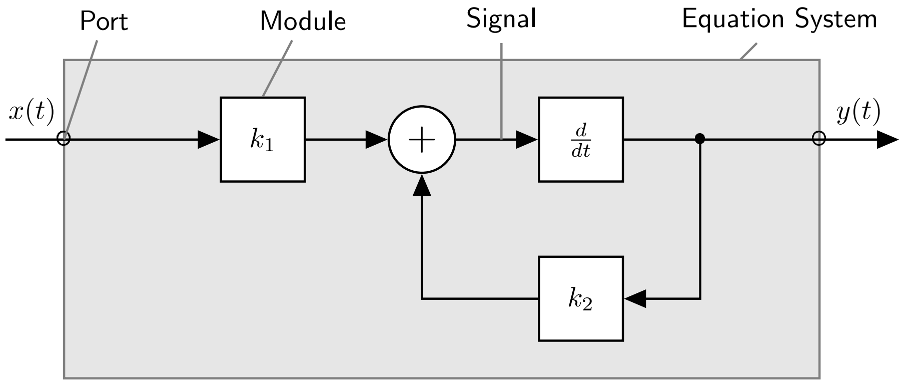


Figure 4.6: Example of a LSF cluster as described in [8]

connected to other TDF models, the timing information of the TDF ports are inherited. Inconsistencies between local and propagated times might lead to an improper connection and wrong simulation results. Due to this issue, it is crucial to define the simulation step for the entire system.

4.1.6 The use of SystemC AMS in SysCore

The basic simulation framework was implemented in [39] and is summarized at this point. The main and most complex part on that work is the parser, which is able to instantiate and connect pre-compiled SysCore modules, via `sca_lsf::sca_signals`. The parser uses the graphical UML description, which requires to solve dependencies recursively in case of compositions.

However, in order to extend the library, it is required to implement the `if_sim_obj_plugin` interface for the new component, such that the component can be registered and instantiated by the parser.

As already mentioned, LSF modeling requires the use of primitive modules, defined in the SystemC AMS library. Referring to the core components of SysCore, this approach is most sophisticated for the implementation of simple components, such as `add` or `delay`. In that case, the primitive LSF component is embedded in a class, which implements the required SysCore interface. An example for this approach is shown in listing 4.2.

```
class add: public sc_module, public if_sim_obj_plugin {
private:
  sca_lsf::sca_add add_;

public:
  sca_lsf::sca_in x1;
  sca_lsf::sca_in x2;
  sca_lsf::sca_out y;
  sca_core::sca_port<sca_lsf::sca_signal_if>* getPortByName(
    std::string name) {
    if (name == "x1"){
      return &x1;
    }
    if (name == "x2"){
      return &x2;
    }
    if (name == "y") {
      return &y;
    }
  return NULL;
  }

  void setParameter(std::string key, std::string value) {
  }

  bool checkParameters(){
    return true;
  }

  std::string getInstanceName(){
    return this->name();
  }

  add(sc_module_name name, std::map<std::string,std::string>*params=NULL):
    sc_module(name), add_("add"), x1("input1"), x2("input2"), y("output") {
    add_.x1(x1);
    add_.x2(x2);
    add_.y(y);
   }
};
```

Listing 4.2: The example highlights the implementation of a core component by wrapping a standard SystemC AMS component in the Syscore interface.

Embed TDF in LSF modules

In comparison to TDF, LSF does not allow to implement other functionalities than those provided by connecting primitive SystemC AMS library modules. This can be reasoned with the direct solving of the system of equations. To circumvent this issue and define components with higher complexity, LSF modules are commonly used as wrappers for TDF modules, which implement a functionality that can be solved numerically. The interaction between TDF and LSF requires the use of converter ports, which are part of the SystemC AMS library. The converted signal is thus of discrete-time.

The shown `stop` example in listing 4.3 highlights SystemC AMS's capabilities to embed components, written with other MoCs by using converter ports. In that case the `sca_lsf::sca_sink` converts the LSF to TDF signals. The generated `tdf_in` of type `sca_tdf::sca_signal<double> tdf_in` has a connection to the input of the `tdf_stop stop_` shown in listing 4.1.

4.1.7 Connecting SystemC TLM-2.0 with SystemC AMS models

A further crucial aspect of this part is the methodologies which establish hardware platform integration into SystemC AMS models. Meaning especially the communication interface between the existing SysCore approach and the virtual hardware prototype.

The authors of [12] describe an approach of how to connect SystemC AMS with TLM 2.0 models and vice versa. The representative example is the data exchange between different TDF sources and a loosely timed TLM-2.0 DSP, which uses a bus system for the internal communication.

Comparing the simulation behavior of SystemC AMS and SystemC TLM-2.0, there seems to be a problem when coupling both. As mentioned in 4.1.3, SystemC TLM-2.0 gains its speed-up by allowing its components to run temporally ahead of the simulation time. SystemC AMS on the other hand, and especially TDF is strictly timed and relies on a predefined static schedule. The authors [12] claim that the SystemC AMS simulation kernel would block the ordinary SystemC kernel all the time if a TDF model is simulated together with bare SystemC. To circumvent this issue, converter ports can be used to establish the connection between TDF modules and an ordinary SystemC signal. In case of an access to such a port, the AMS kernel triggers an interrupt, which causes a context switch to the SystemC simulation kernel, such that it can catch up with the AMS kernel, which is always running ahead of time.

In their paper [12], they described two different approaches, with the conversion from TLM to TDF and vice versa. Both approaches have the necessity for the implementation of a converter, which has to fulfill certain properties, depending on how data is processed.

```
class stop: public sc_module, public if_sim_obj_plugin {
private:
  sca_lsf::sca_tdf_sink in_lsf2tdf;
  tdf_stop stop_;
  sca_tdf::sca_signal<double> tdf_in;

public:
  sca_lsf::sca_in x;
  sca_core::sca_port<sca_lsf::sca_signal_if>* getPortByName(std::string
      name) {
    if (name == "x"){
      return &x;
    }
    return NULL;
  }

  void setParameter(std::string key, std::string value){
  }

  bool checkParameters(){
    return true;
  }

  std::string getInstanceName(){
    return this->name();
  }

  stop(sc_module_name name) : sc_module(name), in_lsf2tdf("x_to_in"), stop_
      ("stop"){
    in_lsf2tdf.x(x);
    in_lsf2tdf.outp(tdf_in);
    stop_.in(tdf_in);
  }
};
```

Listing 4.3: Interface implementation of the LSF stop plug-in. It converts LSF into TDF signals, the functionality is implemented in the `processing()` method of the `tdf_stop` module.

- **Converting from TDF to LT-TLM:** As already mentioned, the approach of interest is to monitor data produced in TDF with a platform built in TLM-2.0. For a real CPS, this would be typically realized with an ADC. The authors describe two possible scenarios for the implementation. One way would be to implement the TDF converter as the TLM initiator. Therefore the converter would bundle the data and send it to the platform with a TLM write command. In the second case the TLM side would be working as an initiator and send read commands to the converter. This could cause problems if no data is available. Therefore a buffer should be used on the TDF side to store received data tokens and provide it on read transactions. Further details to the implementation of this approach can be found in the section 5.1.2.
- **Converting from LT-TLM to TDF:** In that case, the TLM implementations sends write commands in different time spans to the converter. The converter should buffer the incoming data and stream it to the TDF reading side, which is continuously polling for data. If no data is available an interrupt is triggered, which yields the current SystemC simulation, such that TLM can catch up in its simulation.

Another approach which addresses this issue was described by Lonardi and Pravadelli [28]. Their paper focuses on how hardware emulators such as QEMU and OVP can be simulated together with developed SystemC components. Although they mention OVP's capabilities for TLM-2.0 communication, they claim that the approach of using sockets is too slow for a co-simulation with arbitrary SystemC components. Therefore they have implemented a virtual device bridge, written in SystemC, which uses shared memory and synchronization mechanisms for data exchange. In general, the bridge is used for both connections (QEMU and OVP). There are only a different implementations of a virtual device which connects the bridge with the simulator. This allows them to be independent of the abstraction level of the used SystemC model.

Although this approach might have worked for the purpose of the work in hand as well, it seemed less appropriate in comparison to the description of [12].

4.2 Open Virtual Platforms

Embedded software is often written in a desktop environment, on a general purpose operating system. This environment differs most often significantly from the target platform and parts of the designs have to be adjusted after integration on real hardware [35]. An approach which copes with this issue is the use of instruction set simulators for certain architectures. Since used SoC components are often developed by different vendors, it is difficult to write accurate models.

The Open Virtual Platforms initiative, founded by Imperas, provides a library of processor and peripheral models from different vendors. OVP can be used freely for non-commercial use [35] which allows designers to develop and simulate own SoC architectures. OVP focuses on performance enhancement, which can be explained by their software approach to model and simulate virtual platforms. It allows for the execution of a full system simulation even in real time. The main requirement on the system is the usage of instruction accurate binaries as well the implementation of all required interrupts. Such a design allows to build whole systems, taking the environment into account. Meaning, that the developed software is able to run under the same conditions as on the target platform. The main disadvantage can be seen in the fact that such models provide far less hardware details, which means that they are not synthesizeable like pure HDL models, as explained in 2.4.1.

OVP makes use of different technologies, which allow to build advanced homogeneous or heterogeneous multi-core platforms. Following these concepts should be described. The focus lies however on OVP's modeling capabilities, the definition of new peripherals and how developed platforms can be embedded into SystemC designs.

4.2.1 OVP's core technologies

The whole OVP simulation framework is pretty complex and many technologies are used to establish the full functionality, which is necessary to model and simulate heterogeneous platform models. OVP provides, on the one hand side a growing library of component models, which can either be downloaded as precompiled object code or source files and on the other side, it allows the user to create own models and new platforms. Therefore OVP has to define different APIs to guarantee integration capabilities. This section summarizes the main purpose of OVP's core components, and the respective APIs. Since architectures are always meant to consist of processors, peripherals, and their according connections, this section is structured in a similar way.

Platforms

OVPsim is the core element of the framework and implements the OpenPlatforms (OP), former innovative CPU manager (ICM), API which allows to load models of OVP's library (memories, processors, peripherals), application binaries and provides many other functionalities. OVPsim is a just in time (JIT) code morphing simulator engine, meaning that target instructions are dynamically translated to respective x86 host instructions, which can be simulated highly optimized and with the fastest throughput on the host system.

OVPsim is a run-time library. Due to this property, it can be used as stand-alone application or called from an external simulation environment such as the SystemC's simulation kernel.

Processors

The library supports various processor architectures such as ARC, ARM, MIPS, PowerPC, Xilinix and others. Processor models have to be written with the virtual machine interface (VMI) API and can then be compiled to a shared object. The created ISS can be loaded by OVPsim. The implementation of a processor requires some major components such as an instruction decoder, an external hardware interface, and model members to support different variants of a processor family. Other features that can be implemented are disassemblers, instruction morphers and debug interfaces. Sometimes it is convenient to use features of the host systems instead of implementing it on the target system, therefore semihosting can be used. An illustrative example might be the redirection of a Universal Asynchronous Receiver/Transmitter (UART) `write` call to the simulator `log` or `stdout`.

However, for the work on this thesis, only predefined processor models have been used. For further details about how to implement processors with OVP [26] is recommended for the intendend reader.

Peripherals (OVP peripheral modeling guide)

Peripheral models are generally executed in their own virtual environment, the peripheral simulation engine (PSE). The behavioral features of a peripheral have to be modeled with the BeHavioral Modeling (BHM) API. It provides threads, events, simulated delays, simulator control as well as diagnostic control.

An entire peripheral model has to consist of different definitions that can be automatically retrieved from a TCL file using OVP's tool iGen. The generated source files `pse.igen.c` and `pse.attrs.igen.c` contains the initialization and interface definitions of the peripheral.

The functional behavior of the model is implemented in a corresponding `user.c` file, which uses threads to model concurrent behavior. Threads, created in a PSE, run in a safe environment and cannot be seen from outside and run in general until a `wait()` of execution occurs in the simulation. Events are typically used for the synchronization of threads.

Communication

The Peripheral Platform Modeling (PPM) API can be seen as an extension to the BHM API, which allows the interaction of components in a modeled platform. It includes reading and writing memory regions, interrupts and task communication. Platform connection is established by using the ports of the model.

- **Net connections** come with a ppmNetHandle, which is updated during construction and can be used for writes and reads on the connection.
- The **bus slave connection** establishes a connection by mapping local memory to the address space of the bus. This establishes memory-mapped I/O read and write callbacks, which can be applied to local memory regions. One API command of interest is the `ppmCreateRegister`, it allows the definition of registers that can be accessed by the slave port.
- The recommended way to establish a **bus master connection** is to use the API methods `ppmReadAddressSpace` and `ppmWriteAddressSpace`. The main advantage is that the simulator can track all bus cycles.

4.2.2 OVP models in SystemC TLM-2.0 platforms

For the creation of OVP SystemC TLM-2.0 platforms, it is necessary to instantiate the used models within TLM-2.0 wrappers. Those wrapper files implement a certain interface, depending on the used model (processor, MMC, peripheral). In general, they implement the necessary `sc_module` class as well as the OVP specific interfaces.

Processor

Processor models generally run their instruction set simulators in a SystemC thread, and execute the number of instructions per Quantum without affecting the SystemC time. Some of the instructions are propagated to other components in the platform within this thread.

For the usage of OVP models, SystemC has to instantiate a TLM Platform object, that keeps the quantum period. The quantum is used to define how long a processor model has to wait before running again. The quantum size is always a trade-off between a more accurate representation of reality and better simulation performance. In general, the quantum period should be smaller than the shortest time delay modeled in any peripheral with which the processor interacts. Meaning that the processor must be scheduled in between the delay.

Peripherals

Peripherals are typically connected to the bus as slaves. In the simulation there exist different ways of how to activate a peripheral:

- One first approach is the propagation of a TLM-2.0 transaction by another model, which needs to be handled with a read or write on the bus.

- Writing to a net propagates the signal directly to the connected peripheral model and triggers an interrupt.

Interrupts

Interrupts are modeled with analysis ports in the TLM-2.0 interface. The analysis port has the property to propagate new values directly, whereas the net implementation of SystemC would require the scheduler. This would lead to large and unrealistic delays in the simulation, which have to be avoided.

4.2.3 iGen - tool control language

As already mentioned, OVP provides a proprietary language to describe components and platform architectures called TCL. Semantically and syntactically correct TCL scripts can be interpreted by OVP's model generator iGen, which generates according C or SystemC code, which represents the model.

Generating Platforms

For the approach of this thesis, especially the capabilities to generate platform code was crucial. The directory structure of the Imperas Library was designed upon the VLNV standardization from the Spirit consortium. The main aim is to establish a unique identification for models by providing the parameters vendor, library, name, and version. In a similar way, the iGen retrieves necessary information of the instantiated models within the platform. An example for an iGen description is outlined in listing 4.4, the corresponding graphical description can be seen in figure 4.7. The example should give a basic idea of how elements can be instantiated and connected within a TCL platform description. In section 5.1.1 further attempts are made to establish mappings between a graphical UML description, as used in SHARC, and TCL commands.

Platform descriptions as shown in 4.4 can be directly converted to SystemC TLM code passing following parameters to `igen.exe`.

```
> igen.exe --batch model.tcl --writetlm model.cpp
```

```
ihwnew -name simpleCpuMemory
ihwaddbus -instancename mainBus -addresswidth 32

ihwaddprocessor -instancename cpu1 -vendor ovpworld.org
                -library processor -type or1k -version 1.0
                -semihostname or1kNewlib -variant generic
ihwconnect -bus mainBus -instancename cpu1 -busmasterport INSTRUCTION
ihwconnect -bus mainBus -instancename cpu1 -busmasterport DATA

ihwaddmemory -instancename ram1 -type ram
ihwconnect -bus mainBus -instancename ram1 -busslaveport sp1 \
           -loaddress 0x0 -hiaddress 0xffffffff
```

Listing 4.4: Outline of the Platform.tcl script providing information for the conversion with iGen [27].

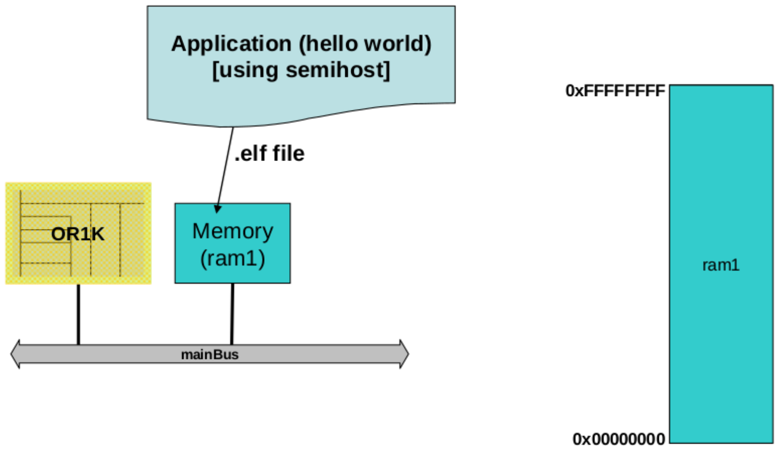


Figure 4.7: A bare metal platform consists only of a processor and a memory, which communicate over a bus system. The figure shows the graphical equivalent to the TCL description in listing 4.4 [27].

4.3 Graphical modeling languages

The conceptual design for technical systems is nowadays mainly done within the first design phases and uses the power of graphical modeling languages. The advantage lies in the fact that processes and components can be explained in a standardized manner, and further details can be disregarded at first. This usually leads to a better overall understanding of the system, whereby first semantics are provided from the developer's side.

4.3.1 UML

During the 1990s object-oriented programing (OOP) got very popular. This phenomenon can be explained by the increasing complexity of software projects and upcoming side effects during the development process. One issue that provides OOP is that software can be structured in a way which corresponds more to the human way of thinking. The hierarchical outline of different objects is more intuitive and allows encapsulations of components. Within the OOP developing process, it is common to visualize class dependencies and various work flow scenarios of the program usage. All in all, this led to proprietary dialects, which ended up in the standardization, which is known as Unified Modeling Language [10]. One of the main goals of UML is to provide exchangeable model information between different parties, which are involved in the design process. This standardized semantic makes UML powerful enough to synthesize code from given graphical model descriptions. The scope of UML has steadily increased over time and is currently in version 2.5 [10], which provides a very extensive range of different diagram types and modeling opportunities. In the first place, one distinguishes between a structural representation of components and the illustration of the functionality for a developed design. The first group contains, for instance, class and component diagrams, whereby the second includes, among others, sequence or use case charts. However, the reader should keep in mind that only one special purpose can be outlined within a certain diagram type. The representation on a very high level of abstraction mainly leads to an improvement of the overall design process, since people without detailed technical knowledge can be integrated into the development as well. Furthermore, basic requirements for the software can be kept in sight easier.

XMI and UML profile mechanism

One of the corner stones for the wide success of UML is the XML metadata interchange (XMI) format, which allows the specification of metadata for the usage of the extensible markup language (XML). In this case, each metamodel, that follows the requirements of the meta object facility (MOF), can be integrated into XMI [34]. MOF can be imagined as a specification language for modeling languages and XMI offers the possibility to provide compatibility among those defined modeling languages. This makes it, for instance, possible to show UML models in different editors. Furthermore, XMI allows the definition of profiles, suitable for a particular purpose respectively domain. Therefore stereotypes extend the vocabulary of UML in order to create new model elements, derived from existing ones. This led to other modeling languages such as SysML or MARTE which are described in the sections 4.3.2 and 4.3.3. Additionally, this extensibility mechanism was used to provide further information about OVP's hardware models, which got integrated into the SHARC IDE. A detailed description of the required extensions can be found in section 5.1.1.

Composite Structure Diagram

UML plays a crucial role in the SHARC IDE since the composite structure diagram is used for the system design. The central components are the so-called parts and the linking connectors', which are represented in a classifier container. In case of the electric Vehicle example, the main eVehicle container represents the classifier and the used subcomponents are parts, which are connected with connectors. The graphical representation of the example is depicted in figure 3.2.

The same concept can also be applied to the test instances. In this case, the testbench is the classifier, and the eVehicle is incorporated as an attribute respectively part, among other corresponding components, necessary for the verification, as shown in figure 3.4.

4.3.2 SysML

Since UML has proven to be a good modeling language in terms of software development, different teams of experts from leading industrial companies have investigated many efforts to create similar semantics for the description of system components at a high level of abstraction. The developed approach has also been standardized by OMG and is called SysML which is an acronym for System Modeling Language [33]. SysML is based on some achievements of UML2 but does not implement it entirely, which is illustrated in figure 4.8. This offers the possibility to integrate parts of SysML in UML and vice versa. But SysML offers even an extension to the diagram and modeling elements.

One major difference is that UML is designed to model software components. In comparison to that SysML offers more modeling perspectives, which relate primarily to the product architecture and its functionality [7]. This means that models can be designed with respect to the physical environment and its constraints.

Relying on the principles of ISO 26262, system design has to be done with respect to a predefined safety goal. Safety Goals can be seen as a requirement for the item, which may be described textually in the SysML requirements diagram. In the context of the SHARC

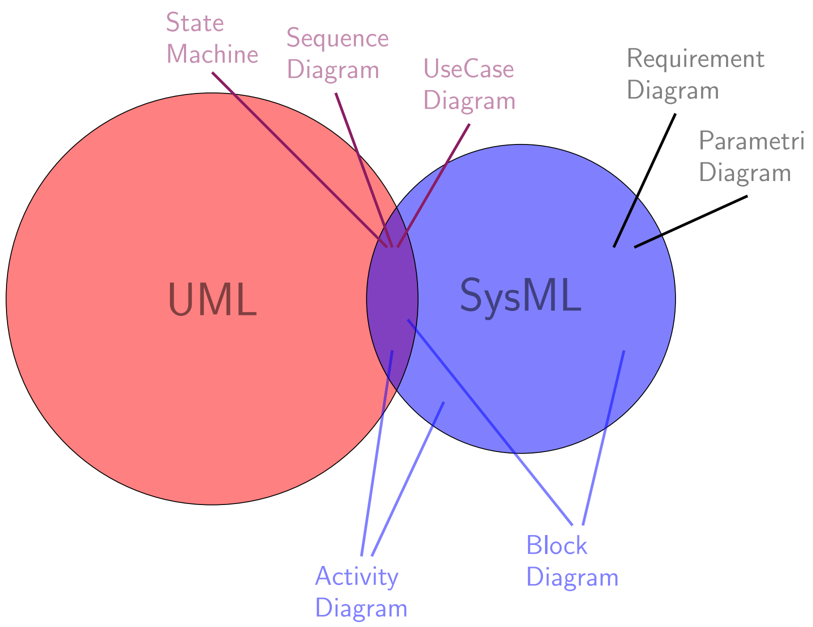


Figure 4.8: Distinction between SysML and UML and their similarities [7].

IDE these concepts were used in terms of verification, and safety goals were mapped to certain test scenarios.

4.3.3 MARTE

There are various modifications and domain specific extensions to UML, an example which is used in the design of embedded electronics is MARTE. MARTE is an acronym for modeling and analysis of real time and embedded systems and has its own OMG standard [32]. In comparison to UML, MARTE allows to model also the computational technology, which affects the design of cyber-physical systems in an essential manner. Core concepts in this sense are the definition of execution times, the possibility to describe hardware resources (such as processors, memories, networks or I/Os) or real-time specific software solutions (threads, processes, mutexes). According to [40], the extensive use of MARTE allows the validation of designs, with regard to their performance before the actual implementation.

For the work in hand especially MARTE's capabilities for platform description are crucial, thus stereotypes for hardware descriptions should be further described since they are used to describe the virtual hardware platforms in section 5.1.1.

- **Processing resources:** A commonly used stereotype in this context is the HwProcessor, which defines attributes such as the used instruction set architecture, cache type, number of cores or MIPS rate of a processor. HwASIC or HwPLD stereotypes are also among those kind of classes.

- **Storage resources:** The HwMemory can be used to further describe storage such as RAM or ROMs. The standard stereotype has different attributes such as memory size or throughput.
- **Communication resources:** The HwBus stereotype is meant to be used for the description of a bus-type communication. It is derived from the HwMedia stereotype and has special attributes such as bandwidth, addressWidth or wordWidth.

4.4 Integrated development environment

As already mentioned, there are a variety of UML editors, which are to a good portion proprietary and tailored for certain areas of application.

The goal of SHARC is to provide a platform which offers opportunities for extensions, an integrated graphical UML editor as well as the possibility to develop core components in SystemC. Therefore Eclipse has been selected, which combines all requirements through its plug-in mechanism. The advantages and features of the Eclipse IDE are the subjects of this section. Furthermore, the used UML editor Papyrus is introduced as well.

4.4.1 Eclipse IDE

Eclipse is a widespread framework for the creation of Integrated Development Environments and is based on an original IBM development. The main objective was the reduction of the increasing incompatibility between various IDEs. In that sense, the design was mainly built with the aim of modular expandability and a unified basic framework. Eclipse was published as an open source project in 2001 and provides the basis for more than 200 projects nowadays, dealing with various aspects of the software development [13]. A wide distribution is mainly accomplished from the Eclipse-based Java IDE. The Eclipse Foundation coordinates further development of the basic platform and guarantees the support for official software projects under the Eclipse Public License (EPL).

Many ingredients of the IDE are of general purpose. This typically includes the user interface, extensibility through plug-ins or help components. These constituents have been extracted and got bundled as framework, which allows the development of stand-alone applications, having a unitary structure. The resulting outcome is now in its 4th version and known as rich client platform (RCP).

There are several aspects that speak for an RCP-based development. Firstly, the available basic components for creating graphic applications. Secondly, of course, the modularity, which allows to separate large applications into smaller parts. Since the mentioned modularity is a fixed part of Eclipse, it also offers tools for the development of plug-ins as well. A significant advantage is also the community size, which includes a various amount of well-know companies such as IBM or Google. These facts guarantee the sustainable development of Eclipse and made it de facto a kind of standard [50] over the past decade.

Core Components of Eclipse

The Eclipse platform is composed of the basic components shown in figure 4.9, which provide the functionality to extend the RCP with plug-ins:

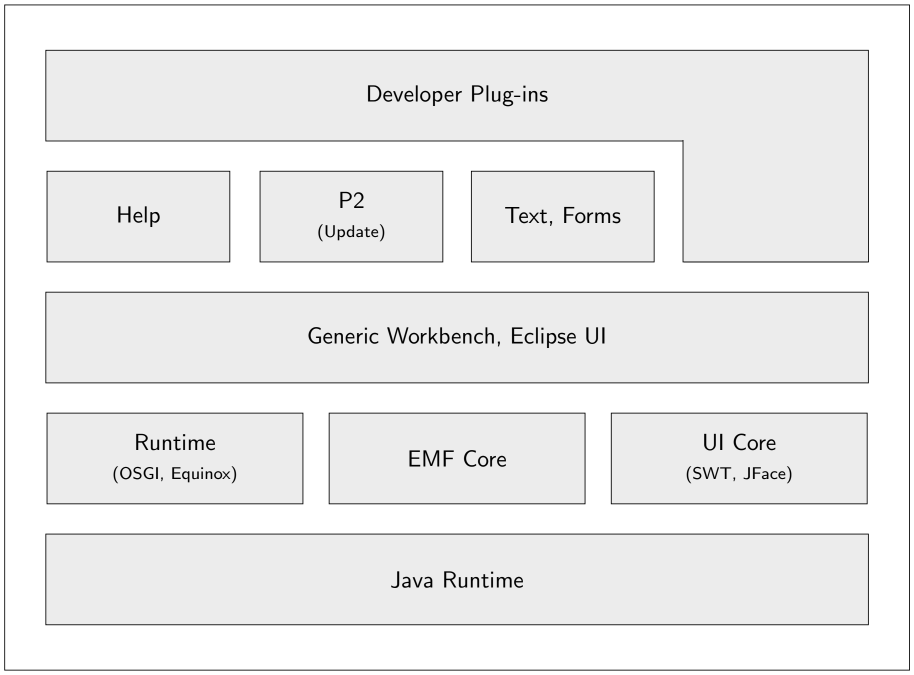


Figure 4.9: Structure of the Eclipse integrated development environment [50].

- The **Java Runtime Environment** is a prerequisite for platform-independent execution of Eclipse.
- **OSGi(Open Services Gateway initiative)** relies on the Java runtime environment (JRE) and is primarily used to group individual services in so-called bundles'.
- **Equinox** is based on OSGi and implements the extension registry, which allows the

IDE expansions.

- **EMF (Eclipse Modeling Framework)** enables the opportunity to create classes from an XMI model description.
- **SWT (Standard Widget Toolkit)** is a graphical widget toolkit which accesses the GUI libraries of the underlying operating system. Thereby JFace can be seen as an extension for this toolkit to accomplish the Model View Control approach.
- **Eclipse UI** instantiates a generic workbench that provides different views, editors, and perspectives.
- Components such as **Help, P2, Text and Forms** serve basic functionalities for RCP developers and can be used by default.
- The above-mentioned components enable the interface for creating own **Developer plug-ins** which allow the modularity of Eclipse.

Eclipse plug-ins

The extension of Eclipse through a plug-in requires two important configuration files:

- The `plugin.xml` file describes the extension of the IDE within its extension mechanisms. An important role is therefore assigned to the usage of extension points since they allow the expansion of those APIs, which define their origins.
- The `MANIFEST.MF` defines the OSGi configurations for the runtime environment of the plug-in. However, this file also provides the definition of extension points for own developed APIs.

4.4.2 Papyrus UML editor

Since UML, SysML and MARTE have proved themselves as modeling languages, there exist a wide range of open, proprietary graphical modeling editors. An implementation available under the EPL is Papyrus. Papyrus provides the possibility to model UML, SysML as well as MARTE, which made it to the reasonable choice for the developed SHARC IDE. Papyrus is built upon the model development tools (MDT) project, which meets all requirements to implement a graphic editor for creating UML diagrams. The plug-in is firmly connected to EMF and uses the opportunity for representation with the graphical modeling framework (GMF) [15].

Chapter 5

Implementation

The following chapter shows a concept of how to combine the two development methods model-driven engineering and model-based designs. Thus it demonstrates how OVP components can be visualized in order to integrat and simulat them along with other components.

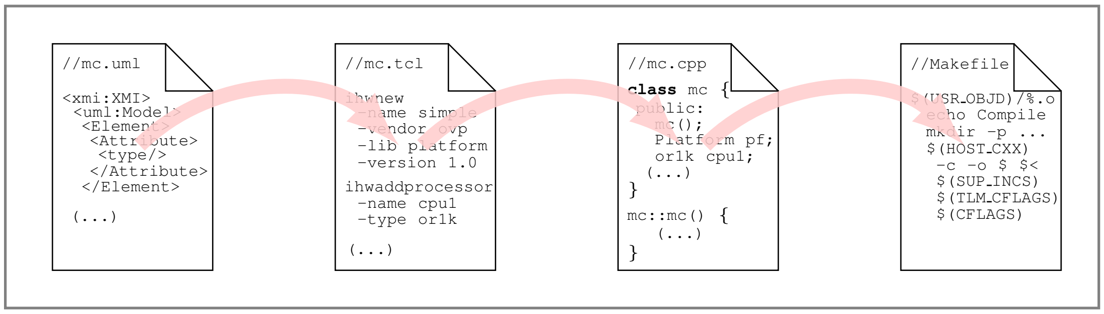
The first part provides a mapping between UML resp. MARTE and the corresponding counterparts of the proprietary description language TCL, which can be converted into corresponding OVP platforms by the introduced tool iGen. Furthermore, it will be shown how a generated stand-alone platform can be embedded in the simulation framework. This requires primarily an interface between SystemC AMS and SystemC TLM-2.0.

These procedures are concretized by means of an application example in which the battery model of an electric vehicle is monitored with a corresponding virtual hardware platform. In this case, the derived microcontroller receives signals from the environment and further communicates with another hardware platform. In doing so, the software solution which is executed on the hardware is covered as well. The simulation of the standardized Worldwide harmonized Light vehicles Test Procedure (WLTP) shows the corresponding behavior of the electric vehicle in the simulation.

5.1 Toolchain for virtual hardware prototyping in SHARC

Relying on the ideas of MDE, it is an outlined goal to model virtual hardware prototypes in UML, similar to what has been shown in previous works [53]. In contrast to the original SHARC framework, the individual hardware components such as CPUs, memories or peripherals should not be implemented in SystemC by the user, instead, it should be possible to make use of models in the corresponding OVP library. This approach changes also the integration of individual components into the simulation. OVP's tool iGen makes it possible to convert TCL platform description into SystemC TLM-2.0 source code, which can be compiled and executed as a stand-alone platform. Due to that, platforms should be described in a graphical manner first, fulfilling the requirements to contain enough information for the creation of an according TCL file. For the graphical platform description, we rely on the capabilities of UML resp. MARTE to describe embedded system properties and map according OVP components to the available profiles. Additional properties, which exceed MARTEs capabilities are defined by additional UML stereotypes.

As already mentioned, a generated platform can only be simulated on its own. Thus it has no interaction with the physical environment in which it is embedded. To cope with this issue, two additional properties have to be fulfilled and the automatically generated source code has to be altered. Firstly, each SysCore plug-in has to implement the `if_sim_obj`, which is required by the SysCore parser to instantiate and connect the models. The second crucial part is the conversion from SystemC AMS to SystemC TLM signals, which is necessary for the data exchange. Compiled to a shared object, the virtual platform can be simulated together with other SysCore components. This approach should guarantee a seamless virtual platform integration into the existing simulation framework and enable the co-simulation of a virtual hardware platform alongside the physical environment. To highlight the individual steps, the described toolchain is shown in figure 5.1.



```
//mc.uml

<xmi:XMI>
 <uml:Model>
  <Element>
   <Attribute>
    <type/>
   </Attribute>
  </Element>

 (...)
```

```
//mc.tcl

ihwnew
 -name simple
 -vendor ovp
 -lib platform
 -version 1.0
ihwaddprocessor
 -name cpu1
 -type or1k

(...)
```

```
//mc.cpp
class mc {
 public:
  mc();
  Platform pf;
  or1k cpu1;
 (...)
}
mc::mc() {
   (...)
}
```

```
//Makefile

$(USR_OBJD)/%.o
 echo Compile
 mkdir -p ...
 $(HOST_CXX)
  -c -o $ $<
  $(SUP_INCS)
  $(TLM_CFLAGS)
  $(CFLAGS)
```

Figure 5.1: The toolchain to convert a graphical UML platform description to a SHARC SysCore plug-in. The steps contain the necessary graphical definition, the derived TCL conversion, an extension of the generated source code and the compilation to a shared object.

5.1.1 Graphical representation of virtual hardware prototypes

The graphical representation is the first part of the toolchain and has the aim to provide enough information to derive a TCL file, that can be converted to respective source code.

To ensure consistency with earlier work on SHARC, UML and the corresponding editor Papyrus was used. Therefore individual Eclipse plug-ins have been modified and extended as described in the following section. In particular, 3 different parts had to be defined and implemented.

Firstly the definition of new UML stereotypes, which contain properties to establish the compliance with OVP models and the TCL description language. The second step was to define UML types for the new OVP hardware components, such as `CPU`, `MEMORY` or `BUS`. The third part is determined by the necessary iGen Commands to create a platform.

UML Stereotype definition for TCL compliance

With MARTE, OMG has defined a UML profile with the capabilities to model real-time and embedded systems. MARTE defines different stereotypes, which allow to assign additional hardware specific properties and information to components. These properties can then be converted to a TCL description. Unfortunately not the entire semantic of TCL can be covered with pure MARTE stereotypes. To cope with this issue, additional stereotypes have been defined in the `CISC-SHARC.profile`. An overview of those definitions is shown in table 5.1 resp. 5.2:

- `VLNV` is necessary for the unique identification of models by providing its vendor, the library, a name and a version.
- The stereotypes `memorymaps` is required for the definition of a memory region, used by the component.
- The `businterface` is applied to a port and provides information about the bus connection.
- `ADC` is used to add certain properties, such as the reference voltage to an ADC model.

Type definition for the component library

As already explained, one of the main concepts of SHARC is composition, which means that a defined UML class can be embedded in another UML class as UML property. The newly instantiated property references to the original class with the definition of a UML type. Thus the predefined hardware components, which are inserted as UML properties need an underlying UML class, which assigns basic attributes, such as ports, to it. The UML classes of the standard components are defined in a central place in a UML model (`ComponentTypes`), which is loaded at runtime. Due to this, the UML `ComponentTypes` model has been extended with new hardware components as shown in table 5.3. To make this UML model accessible for the user, meaning importable as registered package in the

Name	**Attribute**		**Metaclass**

`VLNV`	`vendor`
	`library`
	`name`
	`version`
		`<string>`
		`<string>`
		`<string>`
		`<string>`
			`property`

`MemoryMaps`	`name`
	`baseAddress`
	`range`
	`width`
		`<integer>`
		`<string>`
		`<integer>`
		`<integer>`
			`property`

`businterface`	`name`
	`interfaceMode`
		`<interface_types>`
		`<string>`
			`port`

`ADC`	`vrefhigh` `vreflow`	`<real>` `<real>`	`property`

Table 5.1: The table shows the defined stereotypes to establish TCL compliance. A stereotype consists of a name, its attributes and the metaclass to which it can be applied.

Name	**owned literals**

`interface_types`	`system`
	`master`
	`slave`

Table 5.2: The table shows the defined enumeration, which is an attribute of the bus interface.

model explorer it has to be defined as a UML library. The library is integrated into the IDE via an extension-point in the `plugin.xml` file.

To handle the assignment of a predefined `Core Component` or `Stereotype` to an inserted property, the existing `cisc.helper.advice` Eclipse plug-in was extended in this context as well.

TCL commands for the definition of virtual hardware prototypes

This part contains the description of how the defined stereotypes and classes have been used to accomplish compliance with the TCL platform descriptions. The first step was to determine the necessary commands for the platform definitions in TCL. Basically there exist whole different types of commands, but in general, the set has been reduced such that it is still sufficient for the needs of our application. In this case, 3 different kinds of commands are of importance. Firstly, commands, used for the instantiation of elements, which are shown in table 5.4. The crucial part is, in this case, the definition of the extended

Type	Ports	applied Stereotype	Property
CPU	DATA	CISC::SHARC::BusInterface	InterfaceMode:Master
	INSTRUCTION	CISC::SHARC::BusInterface	InterfaceMode:Master
	inter0	MARTE::GCM::FlowPort	direction:in
	inter1	MARTE::GCM::FlowPort	direction:in
BUS	DATA	CISC::SHARC::BusInterface	InterfaceMode:Slave
	INSTRUCTION	CISC::SHARC::BusInterface	InterfaceMode:Slave
	Initiator	CISC::SHARC::BusInterface	InterfaceMode:Master
ADC	bport1	CISC::SHARC::BusInterface	InterfaceMode:Slave
	AdIn	MARTE::GCM::FlowPort	direction:in
	Interrupt	MARTE::GCM::FlowPort	direction:out
UART	bport1	CISC::SHARC::BusInterface	InterfaceMode:Slave
	DirectRead	MARTE::GCM::FlowPort	direction:in
	DirectWrite	MARTE::GCM::FlowPort	direction:out
	Interrupt	MARTE::GCM::FlowPort	direction:out
RAM	sp1	CISC::SHARC::BusInterface	InterfaceMode:Slave

Table 5.3: Definition of the UML classes to describe hardware components. The table entries contain their ports and the respective applied UML stereotypes. The `MARTE::GCM::FlowPort` contains information about the direction of the signal, whereas the `CISC::SHARC::BusInterface` contains information about the internal bus connection.

element in the OVP library, which can be achieved by using the VLNV pattern. Secondly, there are commands to connect those elements with each other. As shown in table 5.5, the command has always a dependency to the involved parts, which are connected. For the purpose of the implementation, it was sufficient to connect masters and slaves to the bus, as well as directly connect two components through a net. Last but not least, it might be necessary to parameterize instantiated components through the use of the `ihwsetparameter` command, as shown in table 5.6.

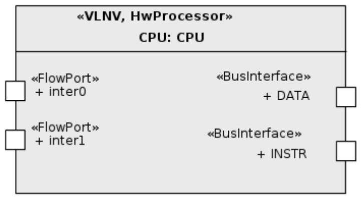
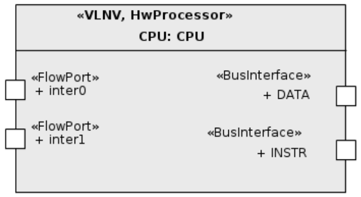
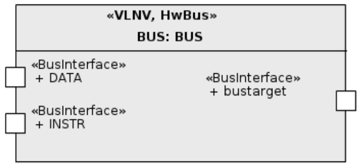
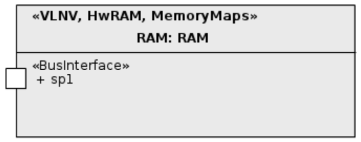
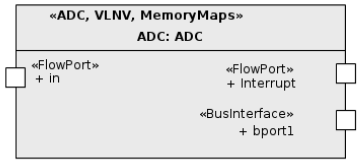
UML	command	mandatory parameters	Note
 «VLNV, HwProcessor» CPU: CPU «FlowPort» + inter0 «FlowPort» + inter1 «BusInterface» + DATA «BusInterface» + INSTR	`ihwnew`	`-vendor` `-library` `-type (name)` `-version`	creates a new hardware design
 «VLNV, HwProcessor» CPU: CPU «FlowPort» + inter0 «FlowPort» + inter1 «BusInterface» + DATA «BusInterface» + INSTR	`ihwaddprocessor`	`-instancename` `-vendor` `-library` `-type (name)` `-version`	many more parameters (i.e. semihost)
 «VLNV, HwBus» BUS: BUS «BusInterface» + DATA «BusInterface» + INSTR «BusInterface» + bustarget	`ihwaddbus`	`-instancename` `-addresswidth`	standard passive element for connection
 «VLNV, HwRAM, MemoryMaps» RAM: RAM «BusInterface» + sp1	`ihwaddmemory`	`-instancename` `-vendor` `-library` `-type (name)` `-version`	address range is given through connection
 «ADC, VLNV, MemoryMaps» ADC: ADC «FlowPort» + in «FlowPort» + Interrupt «BusInterface» + bport1	`ihwaddperipheral`	`-instancename` `-vendor` `-library` `-type (name)` `-version`	adds a peripheral to the design

Table 5.4: The table introduces the TCL commands to instantiate components as well as their respective representation in UML.

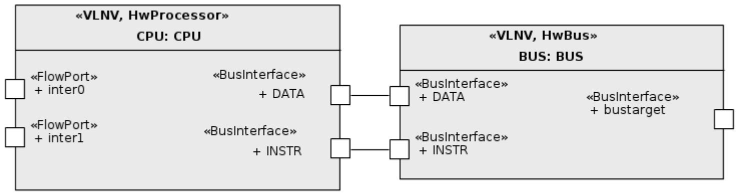
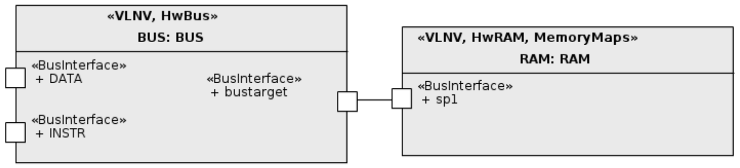
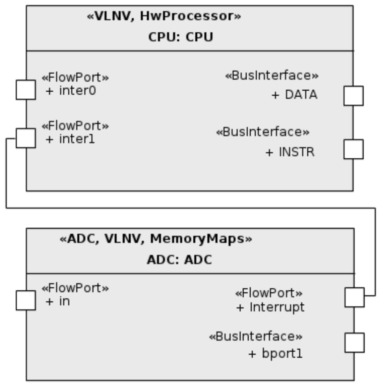
UML	command and parameter combination
 «VLNV, HwProcessor» CPU: CPU «FlowPort» + inter0 «FlowPort» + inter1 «BusInterface» + DATA «BusInterface» + INSTR «VLNV, HwBus» BUS: BUS «BusInterface» + DATA «BusInterface» + INSTR «BusInterface» + bustarget	`ihwconnect -bus -busmasterport`
 «VLNV, HwBus» BUS: BUS «BusInterface» + DATA «BusInterface» + INSTR «BusInterface» + bustarget «VLNV, HwRAM, MemoryMaps» RAM: RAM «BusInterface» + sp1	`ihwconnect -bus -busslaveport -loaddress -hiaddress`
 «VLNV, HwProcessor» CPU: CPU «FlowPort» + inter0 «FlowPort» + inter1 «BusInterface» + DATA «BusInterface» + INSTR «ADC, VLNV, MemoryMaps» ADC: ADC «FlowPort» + in «FlowPort» + Interrupt «BusInterface» + bport1	`ihwconnect -net -netport`

Table 5.5: The table introduces the TCL commands to connect components and the respective representation in UML

command	mandatory parameters	Note
`ihwsetparameter`	`-handle`	handle of the resp. instance
	`-type (bool, float, ...)`	internal parameter type
	`-name`	internal parameter name

Table 5.6: The table introduces the TCL commands to parameterize an instantiated component.

SHARC palette

In order to instantiate the newly created components with the right stereotype, the corresponding palette has been extended by the 5 newly defined OVP components. Figure 5.2 shows the extended version of the palette, elements can be easily inserted to the editor via drag and drop.

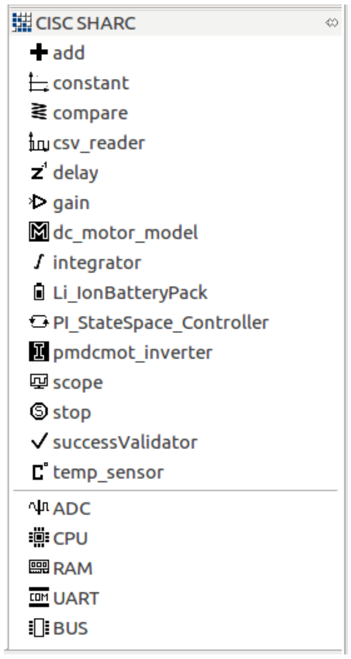


Figure 5.2: Extended palette to describe a virtual hardware prototype

Conclusion

From this point, the graphical representation contains enough information to generate a respective TCL file, which can be converted to respective source code. An example of such a platform design can be seen in figure 5.5, which is part of the outlined eVehicle example. However, for the integration of a platform into the SHARC simulation, the respective `if_sim_obj.h` interface has to be implemented. Further, the respective communication mechanisms between the generated platform and the environment have to be established. These two approaches are further addressed in the following.

5.1.2 Integrating an OVP platform into the SysCore simulation framework

This part is concerned with the second part of the toolchain shown in Figure 5.1. The starting point is a TCL file of the platform that should be created. First, the reader will find an explanation of how the retrieved SystemC source code has to be altered in order to simulate it together with the other SysCore plug-ins. This simple simulation should again be enhanced by the possibilities to interact with the environmental model. To do so, the corresponding signals from SystemC AMS have to be converted to SystemC TLM-2.0 and vice versa. In particular, the two peripherals ADCs and UARTs have been used for communication, thus they are further addressed.

Implementing the the SysCore interface

iGen has some options to convert TCL files to SystemC TLM-2.0 source code. The generated output consists of 3 files which can be compiled and executed as a standalone application. However, the main aim is to create a SysCore plug-in, which requires the class definition, the setting of peripheral attributes, and the constructor. All information can be found in the corresponding `platform.sc_constructor.igen.h` file.

The fundamental structure of SysCore can be found in chapter 3. One requirement for a developed plug-in is the implementation of a predefined `if_sim_obj` interface which accomplishes the structure of a factory pattern. This approach was also shown in listings 4.2 and 4.3.

Thus, the generated code has to be enhanced with the required functions. This is basically `getPortByName(std::string name)`, necessary to retrieve port informations for the connection to other core components. The name of the port in UML is used by the parser to retrieve the respective LSF port from the component and to connect them. Furthermore the generated `platform.cpp` file gets redefined. It has to implement the functions `getName(void)` and `getObj(...)`, which are called from the SysCore parser, when instantiating the used plug-ins resp. shared objects.

In the next part, the necessary modules for the connection of a platform with its environment are described, meaning the connection between SystemC AMS and SystemC TLM.

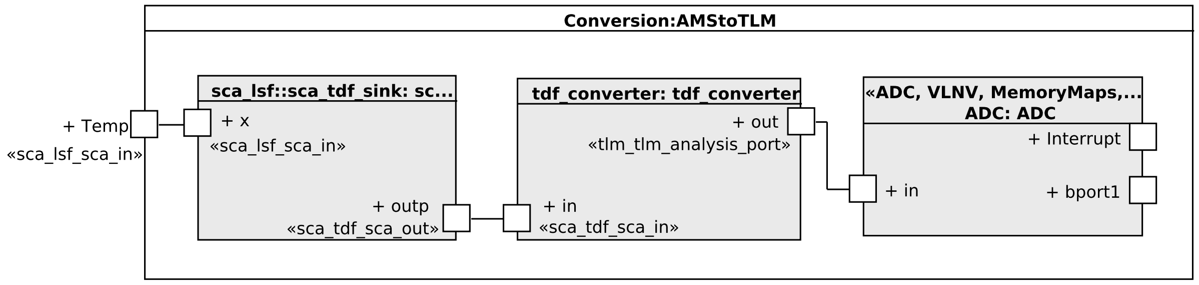
Data exchange between analog and digital domain

In chapter 4 it has been claimed that the SystemC kernel is capable to exchange data between models written in SystemC AMS and SystemC TLM by the use of so-called converter ports. To make use of these capabilities and to realize data exchange, the created platform class has to be extended by new ports and signals, as well as its corresponding constructor.

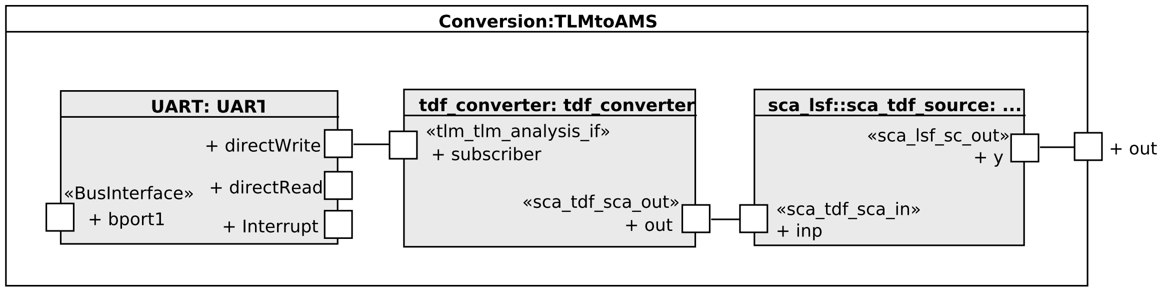
In this context, one should keep in mind, that SysCore uses signals of type `sca_lsf::sca_signal` to connect plug-ins, which means that they are basically of continuous time. The authors of [12] explained that data exchange between SystemC AMS and TLM can only be achieved in case of using discrete-time signals. Due to this, it is necessary to convert the incoming signals from LSF to TDF by using converter ports and doing it vice versa in case of outgoing signals. The implementation depends on the signal direction and thus a distinction is made:

- **Input**: Referring to section 4.1.5, a converted TDF signal is required to be handled in the respective `processing()` method of a TDF module. As already described in 4.2.2 OVP's TLM implementations make use of `tlm::tlm_analysis_port`s, which have the advantage that they are not restricted to the SystemC scheduler. Thus they can publish signal changes directly. One further advantage is that such ports do not require to be bound, which makes it possible to leave them unconnected as well. Within the `processing()` method, the incoming TDF signal is written to a out port of type `tlm_analysis_port`. The `out` port is an attribute of the TDF module and can further be connected to a port of an OVP hardware model. This approach makes LSF signals readable for the used OVP peripheral models. Figure 5.3(a) illustrates the channel.

- **Output**: For the output the way of conversion is in the opposite direction, meaning a TLM signal coming from an OVP peripheral should be converted to a LSF signal. The main difference is that `WritePorts` are of type `tlm_analysis_port`. These ports require to be bound to a subscriber, which implements the `tlm_analysis_if`. To accomplish this requirement, the respective TDF module got extended with a subscriber. The subscriber implements a `write()` function, which stores the incoming values. Further on, these values are accessed in the `processing()` method of the respective TDF module and written to a TDF out port, from which the signal can be converted to an LSF signal. A graphical representation can be seen in figure 5.3(b).



(a) Necessary elements to convert a SystemC AMS LSF signal into a readable signal for the used ADC.



(b) Necessary elements to convert signals coming from the virtual platform into a AMS signal.

Figure 5.3: Ways of how to convert signals to establish communication between the virtual hardware prototype and the surrounding environment.

5.1.3 Use of OVP library models

OVPs library of models and platforms is supported by many different vendors. However, only a few of them provide real source code for the user. Thus certain models were selected for the use within this thesis. The simplest platform would only consist of a memory, a communication bus, and a processor.

Memory and Bus are implemented independently from any vendor. For the processor, OVP makes use of the OR1K processor, with the OpenRISC architecture. A further description of the processor can be found in the following section.

In order to convert electrical signals into a processable form, the model of a Kinetis ADC was used. For this model, the source was further modified. For the communication to other platforms, a UART is used as well, which was taken from the Kinetis library. In the following description the registers to access data are highlighted.

Processor model

The used OpenRISC processor is an open source project, supported and further developed by the OpenCores community. Its underlying design is meant to be a 32/64-bit load and store RISC (reduced instruction set computing) architecture [24]. Within its development, different implementations have been made, reaching from an RTL model, written in Verilog, to the high-speed ISS, which is available through the use of OVPsim. In order to execute software on the defined processor architecture, the application has to be compiled accordingly. A compiler toolchain, for the OpenRISC type, is supplied by the Imperas library, which can be downloaded from their official website [35].

UART model

The UART (Universal Asynchronous Receiver/Transmitter) is a simple, digital circuit that can be used for serial communication between two devices. Thus it is usually installed directly on modern microcontrollers [56]. Data is in general transmitted as a data stream consisting of a start bit, data bits, an optional parity bit and a stop bit. Transmissions take place at a predetermined rate, the so-called baud rate. To ensure the transmission, receiver and transmitter must have the same parameters.

Furthermore, a buffer is usually used, which prevents an overflow between CPU and UART. In this case, further control registers are used, which provide information about the state of transmission or reception of data. Specifically, a UART of the EDA company Kinetis [41] was used in this work. The registers shown in 5.7 are important for sending and receiving data.

Name	Width	Controll bits	Description
UART_AB_C2	8	TIE(5) TCIE(4) RIE(3) ILIE(2) TE(1) RE(0)	This register is used to enable the configure whether the UART.
UART_AB_S1	8	IDLE(3) RDRF(2) TC(1) TDRE(0)	This register indicates, whether data can be send to the UART or not.
UART_AB_D	32	-	The UART_AB_D register is a 8-bit register representing the data stream.

Table 5.7: Overview of the most important registers of the UART model.

ADC model

For the realization of cyber-physical systems, ADCs are used to build an interface between the analog and the digital domain. The task of an ADC is the conversion of a continuous-analog signal into a discretized/digital one, which means that both, value and time, are only represented within fixed intervals. Along the necessary sampling within a timed interval, quantification and encoding are the main tasks of an ADC. Basically, the range for the input signal is divided into predefined quantification intervals, such that each input signal can be assigned to a specific slot, depending on its analog value. Each interval has its own code word, which is typically an ascending binary value, the assignment is known as encoding.

Depending on the resolution of the ADC, which is stated with N, the values range between 0 and $2^N - 1$. If the input voltage is 0V, the output of the ADC is 0 too. In case of reaching the reference voltage on the input, the value on the output is $2^N - 1$. The authors of [18] provide details about the physical properties of ADCs, such as speed or resolution, and distinguish between three different realization concepts for ADCs, which are Direct-conversion ADC/flash ADC, Successive-approximation ADC and integrating ADC. System designers have to be aware of the limitations of the different types and have to select ADCs depending on the needs of the underlying application. Nevertheless, respective ADC implementations in OVP's library are in general on a higher level of abstraction, which is mainly the realization as an algorithmic model. This means the assignment of a certain analog value to one of the quantification intervals and retrieving the respective code word.

From the software developers view, 2 things are of certain interest: First the conversion of the ADC codeword into a respective approximation of the analog value and the way of how to retrieve the converted values from the data registers. The first point is mainly the same for any ADC and can be realized as given in equation 5.1 and 5.2.

$$Q = \frac{V_{refHi} - V_{refLo}}{2^N} \tag{5.1}$$

$$V_{in} = ADCvalue \cdot Q + V_{refLo} \tag{5.2}$$

The second point depends mainly on the realization of the ADC and how to address the respective data and control registers, which vary from vendor to vendor. The most important registers of the used *Kinetis* ADC are described in [41] and shortly summarized in table 5.8.

Name	Width	Controll bits	Description

ADCx_CFG1	16	ADLPC(7)
		ADIV(5-6)
		ADLSMP(4)
		MODE(2-3)
		ADICLK(0-1)
			This register is mainly used for clock configurations of the ADC, as well as setting the mode for the codeword resolution, which can be 8, 10, 12, 16 bits.

ADCx_SC1n	16	COCO(7)
		AIEN(6)
		DIFF0(5)
		ADCH(0-4)
		TRGn0(4)
		FRn1(1)
		FRn(0)
			This register is mainly used to enable the interrupt, which indicates that a conversion has been completed.

ADCx_R	32	-	The ADCRnm register is a 32-bit register holding the A/D conversion results. Depending on the selected mode the results are written to different registers.

Table 5.8: Overview of the most important registers of the ADC model.

5.2 Example platform for a battery management system

In the following section, the described toolchain will be used in an illustrative example. In concrete terms, it is about the monitoring of the battery temperature in the described eVehicle example. Beginning with a general outline of the enhanced model, this part shows further on the graphical representation of a platform and the corresponding TCL code. The two created platforms were integrated into the simulation as described previously, which makes it possible to communicate with their environment. Conclusively the simulation of the created model with a standardized input (WLTP) provides information about the functionality of the written software, which can be further executed directly on the right virtual hardware platform.

5.2.1 Enhancing the eVehicle example by using temperature observation

The first step was determined by the definition of an application for the developed toolchain. In this case, we further extended the used eVehicle example, which is shown in figure 3.2. The original model was used to proof the modeling capabilities of SHARC as described by [39]. It consists mainly of 4 components, which are the inverter, its motor, a state space controller and the battery. The engine of the eVehicle is modeled as a dc motor, which calculates the torque on the shaft according to the given voltage. Further on, the load has a direct impact on the consumed current, which is drained from an advanced lithium-ion battery pack. Within the battery model, the discharging behavior, as well as temperature changes are modeled. Through this information, the module voltage is calculated.

Relying on this rather simple model, it is shown how the described concepts are applied on the existing modeling and simulation framework. Thus an extended BatteryPack was designed, which consists of a resistance thermometer to convert the measured temperature into a respective voltage signal. The retrieved values should be further processed on the generated platform. In addition, noticed temperature changes should be transmitted to a second microcontroller, which might have other communication interfaces or triggers the transition into a safe state if the temperature is too high. A model of this enhanced system is described in figure 5.4.

5.2.2 Definition of the hardware platform used in the BMS

UML, MARTE and the discussed extensions were used to model a microcontroller for the outlined purpose. It consists of an ADC, which converts the temperature voltage from the modeled temperature sensor into a digital equivalent. A direct interrupt connection between the ADC and the 32 bit OpenCore CPU allows direct notification of incoming values, which triggers the program interrupt handler. Through the use of a stereotype, properties, such as the reference voltages of the ADC, can be modeled. In terms of bus communication, the use of TLM guarantees fast transactions and in terms of design, it is sufficient to use only the CPU as a bus master. The memory model is rather simple and requires only the memory block stereotype, which contains information of the memory region that has to be allocated for data processing and program storage. One further

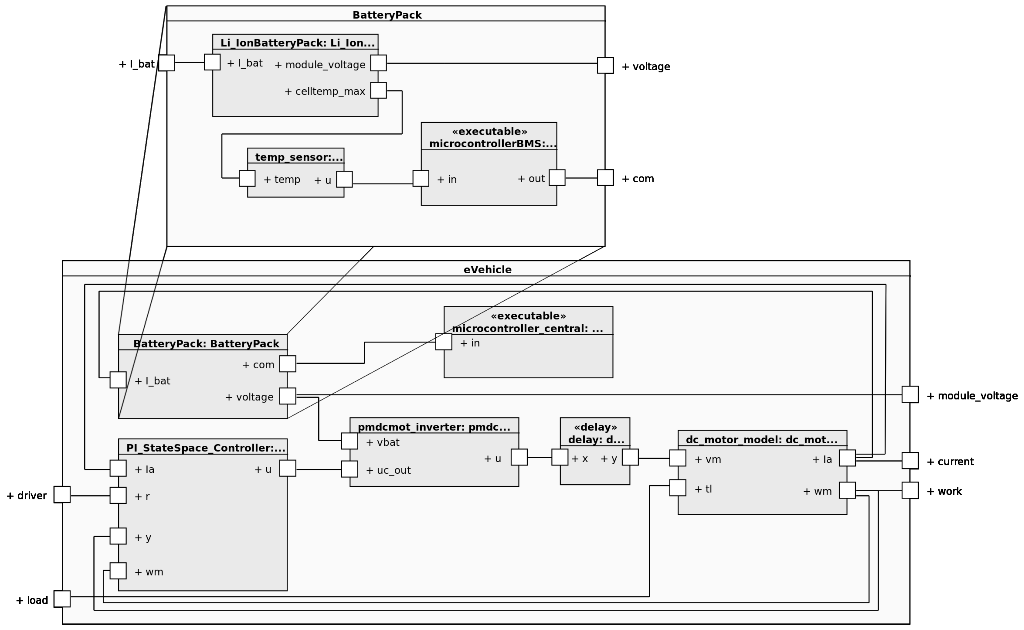


Figure 5.4: Conceptual design of how to enhance the existing eVehicle by the use of virtual hardware prototypes.

peripheral besides the ADC is an Assembled UART, used to directly communicate temperature changes to a second platform. Figure 5.5 shows the fully connected platform, which contains the information to retrieve the according TCL file.

5.2.3 Software running on the defined platform

The developed applications show the capabilities of the created simplified OVP platform. C respectively Assembler have proofed themselves as low-level programming languages and must be compiled to a respective elf file in order to be executed on the underlying OpenCore architecture. The application, which runs on the BMS, waits in a simple endless loop for an interrupt to occur. Due to the configurations of the ADC, interrupts are triggered in case of a successful conversion from an analog to a digital value. This invokes further the interrupt service routine, which is implemented as a function in the application. In this function, the CPU registers are written to memory and the converted value can be read from the `ADC_R` register. Temperature values are transmitted to a second platform, which has a serial connection to the UART of the BMS. In case of a change in the ADC code word, the actual temperature is calculated through the linear dependencies given in equation 5.2. The retrieved voltage comprises information about the temperature, which

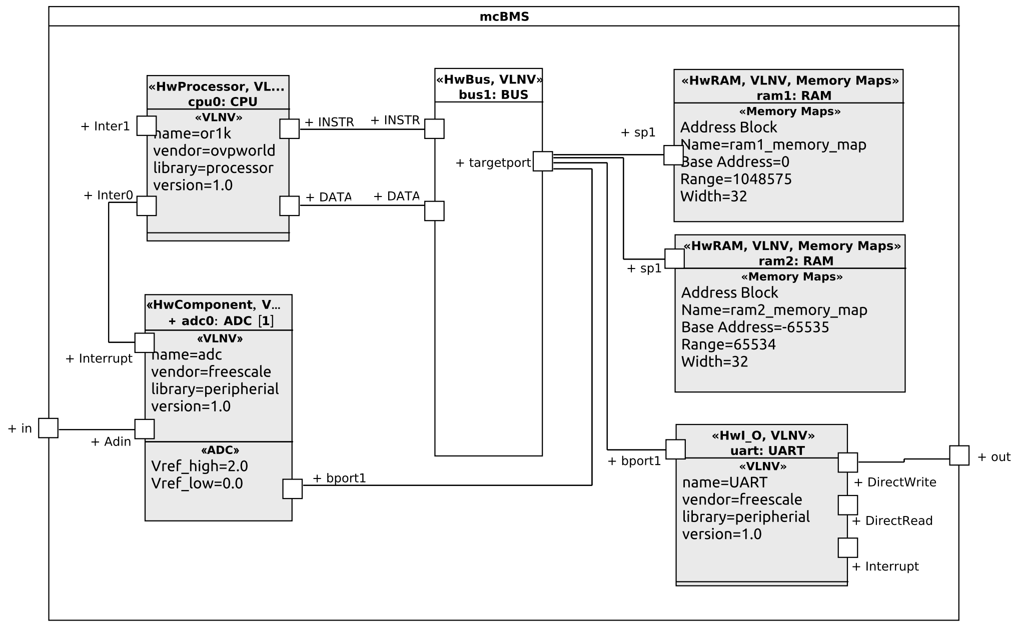


Figure 5.5: Graphical description of the virtual hardware platform by the use of MARTE.

is sent to the other platform by using the serial connection. The application of the other platform is pretty similar and waits for interrupts, which occur when the UART receives a new value.

5.2.4 Simulation with standardized input

The enhanced version of the eVehicle, as shown in 5.4, was used to highlight the described approach to couple the simulation between physical and digital world. Without changing the internal components of the eVehicle, such as motor or controller, the extended version is also driven by the speed of the wheels in [rpm] and the road load in [Nm]. Both affect internal signals of the eVehicle and especially the behavior of the used battery model.

Figure 5.6 shows the simulation results with a timestep of 2[ms] and a duration of 1800[s], which finished in 1011.87[s] real time on a Linux 32-bit machine. The driving speed was defined by the Worldwide harmonized Light vehicles Test Procedure (WLTP), which is a standard maneuver to determine the level of vehicle energy consumption. WLTP defines the speed in km/h, thus it was necessary to convert those values into rpm. The road load depends on multiple factors, such as roll resistance, aerodynamics or the road grade. Since those dependencies were not considered in the outlined eVehicle model, the used value for the simulation is constant 2.5 [Nm]. Even if this value seems inappropriate for a realistic example, it should be mentioned that the focus was set to highlight the integration of the virtual hardware platform.

The functionality of the co-simulation can be seen in the communication signal between the virtual platforms. Due to the resolution of the ADC, temperature changes are only detected in leaps. The peaks in the communication signal outline those leaps and highlight the data transmission from the BMS to the second platform. Although interrupts are continuously triggered by the ADC, transmissions are only performed in case of changing values, which is the desired software solution.

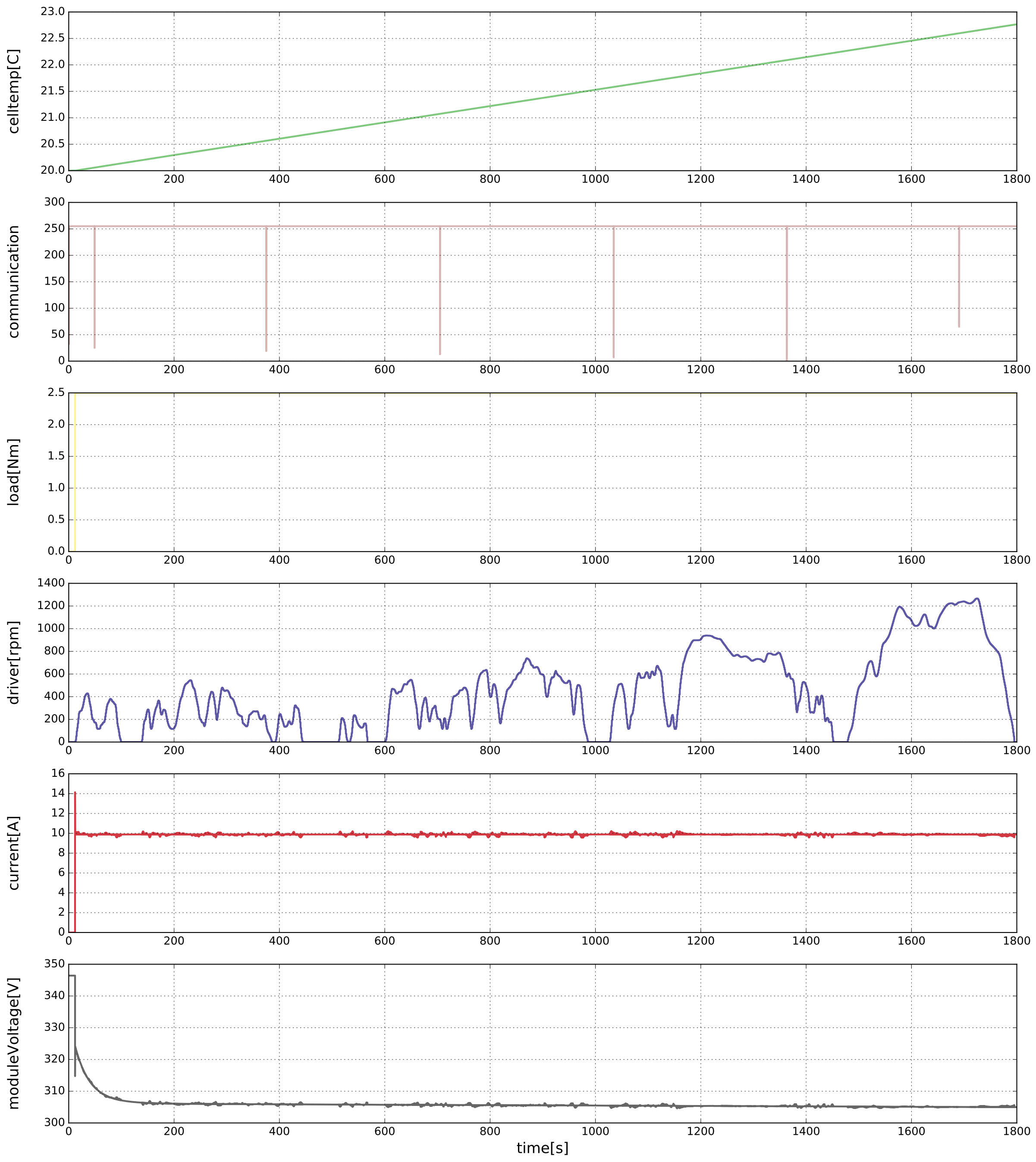


Figure 5.6: Simulation result with the WLTP driving maneuver. The peaks in the communication trace show data transmission from the BMS to the second virtual prototype.

Chapter 6

Conclusion

Within this thesis, we have shown a possibility was presented to design cyber-phyiscal systems in an integrated development environment. In particular, it has been shown that a unification of the development paradigms of model-based design and model-driven engineering leads to an improvement of the entire workflow in which various parties with different fields of expertise knowledge can participate. Glancing at a joint development, which is for instance required by ISO 26262, it is vital to create all these models in a common IDE. Thereby the creation of holistic systems is realized by a graphic description. The decisive advantage over other development methods is the capability to create system models, which may already be simulated in the design phase. The most important aspect of this work is the possibility to integrate virtual hardware prototypes directly into the existing simulation and to create an interaction between the physical and the digital world. Components are therefore not only of functional nature but allow the execution of code, which can run on the used hardware. The development of source code in earlier stages allows shorter development cycles which lead to an advantage over other competitors. Such an approach provides a uniform picture of the overall design, from which further knowledge about the interaction of individual components can be derived.

The possibilities of the simulation were shown by means of a concrete example. An existing eVehicle model was modified such that the built-in battery model got monitored by a generated virtual prototype. In the shown example, the generated platform, communicates with the environment via the interface of an ADC. Furthermore, the abstract communication between platforms was shown by the use of a UART. However, in real world scenarios various ECUs guarantee the functionalities of items. Thus, the interconnection between those components is done via a network in more complex systems. CAN or Ethernet communication are common standards that require an underlying OVP implementation, which is only implemented with limitations. Other aspects that can be addressed are the execution of real-time operating systems or the use of more complex platforms.

In this work, the necessary steps were defined to convert UML/MARTE descriptions into integrable platforms. These concepts could be further improved and realized in corresponding Eclipse plug-ins. Furthermore, the obtained simulation results could be used to verify the Safety Goal, as described in 2.

In summary, it can be said that the creation of complex systems always requires expertise in different disciplines. The simulation of hardware prototypes requires knowledge of the underlying description by the framework. In this case, this was specified with the use of OVP and can in some cases be adapted to individual needs. The cooperation of development teams is vital and becomes more and more important. With this work, a concept was developed to design cyber-physical systems in a single IDE, such that various development teams can work together and share their expertise.

Bibliography

[1] Accellera. *Universal Verification Methodology (UVM) 1.1 User's Guide.* Accellera, 2011.

[2] IEEE Standards Association. IEEE Standard for Standard SystemC ® Language Reference Manual. Standard, IEEE 1666-2011, 2012.

[3] Modelica Association. FMI history. `https://www.fmi-standard.org/history`, 2011. [Online; accessed 04-August-2017].

[4] Ran Avinun. Concurrent Hardware/Software Development Platforms Speed System Integration and Bring-Up. Technical report, Cadence Design Systems, Inc., 2011.

[5] John Aynsley. Getting Started with TLM-2.0. `https://www.doulos.com/knowhow/systemc/tlm2/tutorial__3/`, 2008. [Online; accessed 04-August-2017].

[6] John Aynsley. *OSCI TLM-2.0 language reference manual.* Open SystemC Initiative, 2009.

[7] Laurent Balmelli. An Overview of the Systems Modeling Language for Products and Systems Development. *Journal of Object Technology*, 2007.

[8] Martin Barnasconi and Christoph Grimm. *SystemC AMS extensions User's Guide.* Open SystemC Initiative, 2010.

[9] David C. Black and Jack Donovan. *SystemC: From the ground up.* Kluwer Academic Publishers, 2004.

[10] Steve Cook. *OMG Unified Modeling Language (OMG UML), Version 2.5.* Object Management Group, 2015.

[11] Philippe Cuenot, Patrick Frey, Rolf Johansson, Henrik Lönn, Yiannis Papadopoulos, Mark-Oliver Reiser, Anders Sandberg, David Servat, Ramin Tavakoli Kolagari, Martin Törngren, and Matthias Weber. *11 The EAST-ADL Architecture Description Language for Automotive Embedded Software*, pages 297–307. Springer, Berlin, Heidelberg, 2010.

[12] Markus Damm, Christoph Grimm, Jan Haase, Andreas Herrholz, and Wolfgang Nebel. Connecting SystemC-AMS Models with OSCI TLM 2.0 Models using Temporal Decoupling. In *Forum on Specification and Design Languages*, pages 25–30, 2008.

[13] Eclipse Foundation. FAQ Where did Eclipse come from. `https://wiki.eclipse.org/FAQ_Where_did_Eclipse_come_from%3F`, 2006. [Online; accessed 17-August-2017].

[14] John Fitzgerald and Kenneth Pierce. *Co-modelling and Co-simulation in Embedded Systems Design*, pages 15–25. Springer, Berlin, Heidelberg, 2014.

[15] Sébastien Gérard, Cédric Dumoulin, Patrick Tessier, and Bran Selic. *19 Papyrus: A UML2 Tool for Domain-Specific Language Modeling*, pages 361–368. Springer, Berlin, Heidelberg, 2010.

[16] Cláudio Gomes, Casper Thule, David Broman, Peter Gorm Larsen, and Hans Vangheluwe. Co-simulation: State of the art. *CoRR*, abs/1702.00686, 2017.

[17] Thorsten Gröetker, Stan Liao, Grant Martin, and Stuart Martin. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[18] Harald Hartl, Edwin Krasser, Gunter Winkler, Wolfgang Pribyl, and Peter Söser. *Elektronische Schaltungstechnik: Mit Beispielen In PSpice*. Pearson Studium, 2008.

[19] Martin Hillenbrand. *Funktionale Sicherheit nach ISO 26262 in der Konzeptphase der Entwicklung von Elektrik/Elektronik Architekturen von Fahrzeugen*. Steinbuch Series on Advances in Information Technology, 2012.

[20] ISO. Road vehicles – Functional safety. Norm ISO 26262, ISO, Geneva, Switzerland, 2011.

[21] Frank Kesel. *Modellierung von digitalen Systemen mit SystemC: Von der RTL- zur Transaction-Level-Modellierung*. Oldenbourg Wissenschaftsverlag, 2012.

[22] Aamir Mehmood Khan. *Model-based design for on-chip systems*. PhD thesis, Université de Nice, 2009.

[23] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. Experimental Security Analysis of a Modern Automobile. In *2010 IEEE Symposium on Security and Privacy*, pages 447–462, 2010.

[24] Damjan Lampret. *OpenRISC 1000 Architecture Manual*. Opencores, 1.1 edition, 2014.

[25] Edward A Lee. Cyber Physical Systems: Design Challenges. In *Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, pages 363–369, 2008.

[26] Imperas Software Limited. OVP processor modeling guide. Technical report, Imperas Software Limited, 2015.

[27] Imperas Software Limited. iGen Platform and Module Creation User Guide. Technical report, Imperas Software Limited, 42.

[28] A Lonardi and G Pravadelli. On the Co-simulation of SystemC with QEMU and OVP Virtual Platforms. In *22nd IFIP WG 10.5/IEEE International Conference on Very Large Scale Integration*, pages 110–128, 2014.

[29] James Martin. *Rapid Application Development*. Macmillan Publishing Co., Inc., Indianapolis, IN, USA, 1991.

[30] Shabtay Matalon. Vista Virtual Prototyping. Technical report, Mentor Graphics, 2015.

[31] Francisco Mendoza Cervantes. *A Problem-Oriented Approach for Dynamic Verification of Heterogeneous Embedded Systems*. PhD thesis, KIT Scientific Publishing, Karlsruhe, 2013.

[32] OMG. *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, Version 1.1*. Object Management Group, 2011.

[33] OMG. *OMG Systems Modeling Language (OMG SysML), Version 1.3*. Object Management Group, 2012.

[34] OMG. *XML Metadata Iterchange (XMI) Specification, Version 2.4.2*. Object Management Group, 2014.

[35] Open Virtual Platforms. Open Virtual Platforms - The source of Fast Processor Models Platforms. `http://www.ovpworld.org/`, 2017. [Online; accessed 4-August-2017].

[36] Ragunathan Rajkumar, Insup Lee, Lui Sha, and John Stankovic. Cyber-physical Systems: The Next Computing Revolution. In *Proceedings of the 47th Design Automation Conference*, pages 731–736, 2010.

[37] Roger Aarenstrup. *Managing Model-Based Design*. MathWorks, Inc., 2015.

[38] W. W. Royce. Managing the Development of Large Software Systems: Concepts and Techniques. In *Proceedings of the 9th International Conference on Software Engineering*, pages 328–338, 1987.

[39] Markus Schuss. Design and Implementation of a Distributed Simulation Framework based on Cloud Computing. Master's thesis, Graz University of Technology, 2016.

[40] Bran Selic and Sbastien Grard. *Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2013.

[41] Freescale Semiconductor. *Kinetis Peripheral Module Quick Reference.* Freescale Semiconductor, Inc., 2010.

[42] David J. Smith and Kenneth G.L. Simpson. *Safety Critical Systems Handbook.* Butterworth-Heinemann, Oxford, 2011.

[43] Herbert Stachowiak. *Allgemeine Modelltheorie.* Springer, 1973.

[44] Statista. Size of the Internet of Things (IoT) in retail market in the United States from 2014 to 2025. *Dossier Internet of Things (IoT)*, 2017.

[45] Thilo Streichert and Matthias Traub. *Elektrik/Elektronik - Architekturen im Kraftfahrzeug.* Springer, Berlin, Heidelberg, 2012.

[46] Sang C. Suh, U. John Tanik, John N Carbone, and Abdullah Eroglu. *Applied Cyber-Physical Systems.* Springer, New York, 2014.

[47] Cadence Design Systems. Cadence and MathWorks Provide System-Level Simulation Solutions for Mixed-Signal IoT and Automotive Applications. `http://www.orcad.com/about/news/cadence-and-mathworks`, 2016. [Online; accessed 04-August-2017].

[48] Janos Sztipanovits. Composition of Cyber-Physical Systems. In *Engineering of Systems 14 th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems ECBS 2007*, pages 3 – 6. IEEE Computer Society, 2007.

[49] Techradar. How the space race changed computing. `http://www.techradar.com/news/computing/how-the-space-race-changed-computing-665069`, 2010. [Online; accessed 10-August-2017].

[50] Lars Vogel. *Eclipse 4 RCP: the complete guide to Eclipse application development.* vogella series, 2013.

[51] James D. Weiland, Neha Parikh, Vivek Pradeep, and Gerard Medioni. Smart image processing system for retinal prosthesis. In *Proceedings of the Annual International Conference of the IEEE Engineering in Medicine and Biology Society, EMBS*, pages 300–303, 2012.

[52] Mark Weiser. The computer for the 21st century. *Mobile Computing and Communications Review*, pages 3–11, 1999.

[53] Ralph Weissnegger, Markus Schuss, Christian Kreiner, Kay Römer, Markus Pistauer, and Christian Steger. Bringing UML/MARTE to life: A Model-Based Simulation-Framework for Safety-Critical Systems. In *Proceedings*, 2016.

[54] Ralph Weissnegger, Markus Schuß, Martin Schachner, Kay Römer, Christian Steger, and Markus Pistauer. A Novel Simulation-based Verification Pattern for Parallel Executions in the Cloud. In *Proceedings of the 21st European Conference on Pattern Languages of Programs*, pages 20:1–20:9, 2016.

[55] Ralph Weissnegger, Christian Kreiner, Markus Pistauer, Kay Römer, and Christian Steger. SHARC - Simulation and Verification of Hierarchical Embedded Microelectronic Systems. In *8th International Conference on Ambient Systems, Networks and Technologies*, volume 109, pages 392 – 399, 2017.

[56] Wayne Wolf. *Computer as components.* Morgan Kaufmann Publishers, 2012.