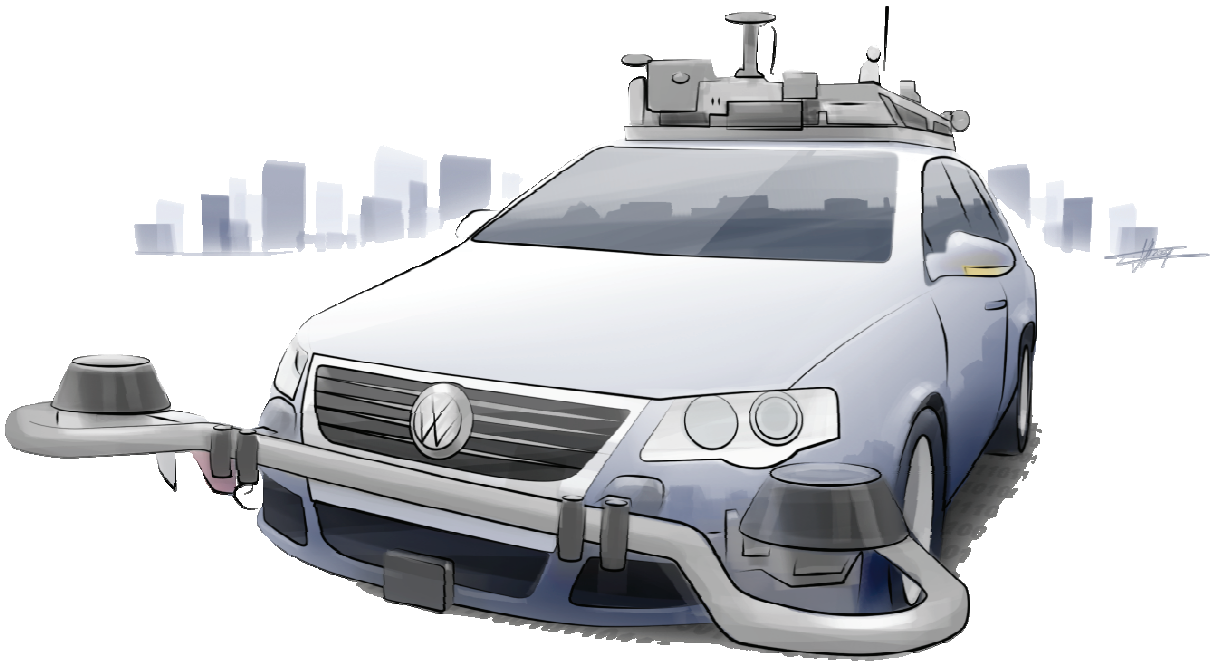
RWTH AACHEN

SE SOFTWARE ENGINEERING

Christian Berger

Automating Acceptance Tests for Sensor- and Actuator-based Systems on the Example of Autonomous Vehicles



Berichte der Aachener Informatik
Software Engineering
Hrsg.: Prof. Dr. B. Rumpe

Band 6

[Ber10] C. Berger
Automating Acceptance Tests for Sensor- and Actuator-based Systems on the Example of Autonomous Vehicles.
Shaker Verlag, ISBN 978-3-8322-9378-9.
Aachener Informatik-Berichte, Software Engineering Band 6. 2010.
www.se-rwth.de/publications

Automating Acceptance Tests for Sensor- and Actuator-based Systems on the Example of Autonomous Vehicles

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Wirtschaftsinformatiker Christian Berger

aus Braunschweig

Berichter: Universitätsprofessor Dr. rer. nat. Bernhard Rumpe
Professor Dr.-Ing. Thomas Form

Tag der mündlichen Prüfung: 19. Juli 2010

Die Druckfassung dieser Dissertation ist unter der ISBN 978-3-8322-9378-9 erschienen.

However impenetrable it seems, if you don't try it, then you can never do it.

Sir Andrew John Wiles (British mathematician, ∗ April 11, 1953)

Acknowledgments

This thesis was created during my work at the Software Systems Engineering Institute at Technische Universität Braunschweig, at the Department of Software Engineering at RWTH Aachen University, and at the Autonomous Ground Vehicles Group of the Center of Hybrid and Embedded Software Systems at University of California, Berkeley.

I am very grateful Prof. Dr. Bernhard Rumpe for carrying out the CarOLO project, for giving me the opportunity to realize this thesis, and for his continuous support. Furthermore, I am very grateful Prof. Dr. Thomas Form for his willingness being my second examiner and for his exceptional support for the CarOLO project. I thank Prof. Dr. Stefan Kowalewski who takes the chair of my graduation's committee and Prof. Dr. Thomas Seidl as well as Prof. Dr. Berthold Vöcking for examining me.

I would like to thank my colleagues from the Department of Software Engineering for intense and inspiring discussions, especially Ibrahim Armac, Christian Basarke, Dr. Hans Grönniger, Tim Gülke, Arne Haber, Thomas Heer, Christoph Herrmann, Dr. Anne-Thérèsa Körtgen, Dr. Holger Krahn, Thomas Kurpick, Cem Mengi, Claas Pinkernell, Holger Rendel, Jan Oliver Ringert, Martin Schindler, Frank Schroven, Steven Völkel, and Ingo Weisemöller.

For many interesting insights into automotive software engineering, I thank Dr. Christian Ameling, Dr. Arne Bartels, Dr. Holger Philipps, and Dr. Dirk Stüker from Volkswagen Corporation.

Furthermore, I thank Prof. Shankar Sastry and Jan Biermeyer for the invitation to visit their Autonomous Ground Vehicles Group at University of California, Berkeley, and Humberto Gonzalez for interesting discussions about autonomous ground vehicles.

For their support for the CarOLO project I would like to thank Dr. Jan Effertz, Fred W. Rauskolb III, and Carsten Spichalsky from Volkswagen Corporation, Prof. Dr. Peter Hecker, Prof. Dr. Marcus Magnor, Prof. Dr. Markus Lienkamp, Prof. Dr. Walter Schumacher, Prof. Dr. Ulrich Seiffert, and Prof. Dr. Lars Wolf. Furthermore, I would like to thank my former colleagues from the very exciting CarOLO project which inspired me for this thesis, especially my brother Kai Berger, Karsten Cornelsen, Michael Doering,

Joop Flack, Dr. Fabian Graefe, Kai Homeier, Felix Klose, Christian Lipski, Johannes Morgenroth, Tobias Nothdurft, Sebastian Ohl, and Jörn Marten Wille. For their support during our preparations for the 2007 DARPA Urban Challenge, I would like to thank the entire team of the Southwest Research Institute in San Antonio, Texas and Tim Kuser who supported us significantly during our visit to Victorville, California.

Moreover, I thank my students Andreas Donners, Stefan Kühnel, and Jonas Schwartze supporting the realization of this work.

And last but not least, I am very grateful my parents, my family, my friends and especially my lovely wife Evelin for her continuous encouragement and patience during my entire work–this thesis is dedicated to you!

Contents

Abstract

In projects dealing with autonomous vehicles which are driving in different contexts like highways, urban environments, and rough areas, managing the software's quality for the entire data processing chain of sensor- and actuator-based autonomous systems is increasingly complex. One main reason is the early dependency on all sensor's raw data to setup the data processing chain and to identify subsystems. These sensors' data might be extensive, especially if laser scanners or color camera systems are continuously producing a vast amount of raw data. Moreover, due to this dependency the sensors' setup including their respectively specified mounting positions and calibration information is also necessary to gather real input data from real surroundings' situations of the system. This is even more important before actually starting to integrate independently developed subsystems for carrying out tests for the entire data processing chain.

To reduce this dependency and therefore to decouple tasks from the project's critical path, an approach is outlined in this thesis which was developed to support the software engineering for sensor- and actuator-based autonomous systems. This approach relies on customer's requirements and corresponding customer's acceptance criteria as well as the decoupling of the software engineering from the real hardware environment to allow appropriate system simulations.

Based on the customer's requirements, formally specified scenarios using a domain specific language are derived which are used to provide surroundings and suitable situations for a sensor- and actuator-based autonomous system. From these formally specified surroundings, the required input data is derived for different layers of a sensor data processing system to generate actions within the system's context. This input data itself depends on a given sensor model to compute its specific raw data. Amongst others, on the example of laser scanners and camera systems, algorithms using modern graphical processing units are outlined to generate the required raw data even for complex situations.

To realize the aforementioned aspects, a development environment is necessary consisting of tools for modeling and working with instances of a domain specific language. Furthermore, a software framework is required which provides easily usable and mature solutions for common programming requirements like synchronization for concurrent threads

or communication in a high-level point of view. For relying on a consistent and homogeneous software framework for implementing the concepts, a highly portable and real-time-capable software framework for distributed applications was realized which was written entirely from scratch in strictly object-oriented C++. Moreover, this software framework also integrates the formally modeled input data derived from the specified requirements and the sensors' models to allow unattended system simulations to support the acceptance tests for subsystems or an entire system.

On the example of autonomous vehicles, the applicability of the approach and the software framework is demonstrated by implementing a vehicle navigation algorithm which uses a given digital map for finding the best route to a desired destination from an arbitrarily chosen starting point. This algorithm was developed considering the test-first-principle and is continuously evaluated by unattended and automatic software tests which are executed on a continuous integration system. Its implementation as well as its evaluation make use of the aforementioned concepts and algorithms. Therefore, the vehicle's surroundings were formally modeled together with its necessary sensors using the provided development tools and environment for performing and evaluating unattended system runs before the algorithm was put into operation on the real vehicle.

1 Introduction and Motivation

This chapter provides an overview and introduction for the development of complex software-intense embedded systems. The main focus is especially on sensor- and actuator-based autonomous systems which can be found in recent driver assistance systems or even in autonomous vehicles for example.

1.1 Introduction

The resulting quality of software-intense system development projects depends to a large extent on well understood customer's requirements. Thus, every development process starts with collecting and discussing requirements together with the customer and ends up in testing the final product against previously defined requirements for matching their fulfillment on the customer's demands.

Using these requirements, programmers and engineers develop the system according to specifications written in regular word processing systems or even in more complex requirements management systems like DOORS [83]. Moreover, specification documents are the contractual base for collaboration with third party suppliers. Thus, these documents build an important source for many other development artifacts, for example class diagrams or test cases.

Furthermore, requirements evolve over time due to technical limitations or legal aspects in the implementation stage or due to changes to former customer's circumstances. New requirements are added while others change. Ideally, the system to be developed should fulfill every single customer's requirement at any time or should iteratively fulfill selected customer's requirements at dedicated milestones.

Due to shorter production times or output-related project goals, the quality assurance must be carried out regularly especially for sensor- and actuator-based autonomous systems which require extensive tests. Even within a tight schedule, regular and automatically executed test suites which are intended to cover large parts of the system's source code

are a proper methodology for the software engineering to get a stable and high quality product nowadays.

1.2 Motivation

As already mentioned above, requirements found the source for any design decision or system specification. This important impact is indeed recognized by every project, but the continuous evaluation of their fulfillment is often difficult to monitor. This is caused by the specification's representation itself which is not directly part of a machine-processable software build *and* quality assurance process. The main reason is the form on one hand since specifications are either written in a natural language or entered in databases. On the other hand, a coherent and consistent methodology for formulating requirements to be part of the system development process itself does not exist in general.

Natural language-based specifications are simple and quick to formulate but they leave an enormous gap for misunderstandings and interpretations. Furthermore, modern revision systems to track changes cannot be applied easily to record and visualize modifications or changes compared to prior versions.

However, database systems perhaps with Wiki front-ends used for collecting and recording requirements allow a distributed collaboration and version tracking for all responsible authors, but in general they are not part of the system development and quality assurance process as well. Furthermore, entries in the database may be outdated, invalid, or contradictory without being noticed by the developer or even the customer.

Newer approaches like the Requirements Interchange Format (*RIF*) [82] simplify the potentially erroneous process of correct requirements exchange with suppliers by reducing misunderstandings on both contractual partners. Yet, this format cannot be part of the software build process at all due to its heterogeneous layout containing formal elements and arbitrary binary data. However, they improve the human collaboration between different project partners.

Moreover, requirements describe only *what* exactly a customer wants to get. But in general, requirements do not describe *how* and *when* the customer *accepts* the developed system, i.e. *which* are the customer's acceptance criteria. This means for the example of a driver assistance system for automatically parking the vehicle between other vehicles at the roadside: The main objective of the vehicle's parking system is evident, but different system engineers may have different opinions about *when* a vehicle is finally parked: For example, one engineer might prefer a smaller distance to the curb while another does

not. The customer's acceptance criterion for the system depends on his interpretation of a specific objective.

Nowadays, proper software engineering methods to tackle the increasing complexity in embedded systems are still evolving [162]. The main reasons are on one hand the challenge to integrate complex third party software and on the other hand inappropriate development processes to handle complex software platforms; the latter are due to an insufficient evolvement over time compared to the evolvement of the software functions' complexity.

Moreover, today's development projects consist of many developers and the resulting quality suffers from insufficient tested software artifacts due to undiscovered software and system bugs. Finally, an approach to test the entire system after integration is missing, often due to inadequate resource allocation for software and system testing.

Especially for the automotive domain, an increasing complexity in Electronic Control Units (*ECU*) and caused by their integration in vehicle buses is remarkable in the last years [61, 171]. To overcome these problems, the Automotive Open System Architecture (*AUTOSAR*) approach was founded in 2002 by several Original Equipment Manufacturers (*OEM*), namely BMW, Daimler, and Volkswagen to standardize tools, interfaces, and processes. Using AUTOSAR, the exchange of ECUs or software components between different suppliers should be simplified. But AUTOSAR itself is not capable of handling directly customer's requirements and acceptance criteria because this is not its intended scope [77].

Newer approaches in software engineering have developed different methods for automatic software tests even in early stages of the development process. These methods are known as *unit tests* to specify expected results of a function or algorithm for a given set of input values [102]. But they only validate the algorithm's conformance to one specific sample or to specific boundaries, but they cannot prove the absence of errors or even verify its formal correctness.

Furthermore, unit tests are mainly used to figure out discrete and time independent algorithms. For evaluating continuous algorithms which require continuous input data from complex surroundings in terms of closed-loops-tests, the concept behind unit tests must be extended. *Mock objects* enable the specification of complex system behavior even if some parts are unavailable. Mock objects are used to substitute subsystems or parts of the entire system like database connections by imitating the sub-system's behavior and replies to function calls. Thus, their usage is usually limited to parts of an entire system.

For software and systems realized using the Java programming language, a stable and

mature set of tools for checking the software and system's quality is available. Furthermore, this tooling is well integrated in graphical Integrated Development Environments (*IDE*) like Eclipse providing all necessary information to the developer at a glance which is sponsored by all important software suppliers like Oracle, IBM, and Google.

Yet, besides a mature tooling comparable to Java systems, the fundamental source of requirements and their corresponding acceptance criteria are still lacking. For ensuring the system quality, the use of its specifying requirements is eligible. Modern development processes stay abreast of changes by defining iterative development cycles which integrate the customer to lower communication barriers.

1.3 Main Goals and Results

The early and intense integration of customer's requirements in the development process shortens response times and reduces misunderstandings. Nevertheless, a formal requirements specification which can be directly used as a consistent, traceable, and formal artifact as part of the software engineering and not only for project management is missing. Moreover, the development and evaluation of sensor- and actuator-based autonomous systems is getting more and more complex nowadays caused by an increasing amount of sensors' raw data. Furthermore, testing such systems is also getting more extensive due to more complex or dangerous situations which have to be set up in the systems' surroundings to provide the required input data.

Hence, for reducing the software's dependency on a mostly complete sensor's setup waiting for required input data before starting the actual development, an appropriate methodology is necessary to support the development, integration, and evaluation of these systems already at early stages. Depending on the system to be developed, the interactive development as well as the system's evaluation require a large and complex stationary and dynamic context which cannot be achieved by using the aforementioned mock objects.

A methodology which integrates the requirements and appropriate acceptance criteria for the software quality process is required for sensor- and actuator-based autonomous systems. The conceptual theory presented in this thesis is to use the customer's requirements *and* acceptance criteria to derive a formal *machine-processable* specification of the system's context of a System Under Development (*SUD*) to generate and perform interactive as well as unattended and automatable simulations for the SUD. The following goals have been achieved:

- Circumstances of a sensor- and actuator-based autonomous system development project are analyzed. Hereby, the design of a Domain Specific Language (*DSL*) to specify the system's context of an SUD based on the customer's requirements to describe an autonomous vehicle's behavior in its system's context is developed.
- Having formulated the system's context as basis for the system development, the definition of metrics for measuring the evolving requirements' fulfillment is subject for analysis. Hence, the customer's acceptance criteria are the basis for deriving metrics which are used to validate the developed system.
- Unit tests are the right choice to perform unattended and automatic regression tests. But for complex software systems especially based on algorithms which produce and process continuous input data, the sole use of unit tests or mock objects is insufficient. Thus, a software engineering methodology which includes the automatable system's evaluation of an SUD is presented which requires the definition and implementation of an unattendedly executable system simulation. The aforementioned metrics are used to establish a concept similar to unit tests.
- To realize the system simulation framework which should not be a stand-alone tool but shall instead use the same concepts and software environment like the regular system development for reducing the tools' heterogeneity, a highly portable and real-time-capable software framework written entirely from scratch in object-oriented C++ was developed which simplifies the development of distributed applications.
- Finally, a proof-of-concept for the software framework and the outlined methodology as well is given by performing a case study on the example of the development for an autonomous ground vehicle.

1.4 Thesis' Structure

The thesis is structured as follows. First, the context and history of autonomous vehicles is presented because they are the ongoing example for this thesis. Afterwards, the overall software engineering methodology to completely decouple and thus to virtualize the software and system development from any real hardware implementation is outlined. Following, the theoretical background for formulating a DSL to specify the system's context of autonomous ground vehicles in a machine-processable manner is presented. This DSL is used in a software framework called $\mathcal{H}$esperia which itself supports the development of distributed real-time applications. Furthermore, it can be used as a framework for

realizing embedded software by providing high-level interfaces and ready-to-use implementation patterns for common programming tasks. These tasks include synchronous and asynchronous communication between processes and systems, handling for complex data structures, or high-level programming concepts. Second, the entire system simulation framework which is a core part of the methodology is realized by using this framework and thus is directly incorporated into the framework.

Afterwards, the evaluation of the fulfillment of the customer's requirements using all parts as described before in so-called system simulations is outlined. Furthermore, interactive inspection as well as unattended monitoring of a running SUD are described before a proof-of-concept is given by presenting an example for developing an algorithm for an autonomous ground vehicle using the elaborated concepts. Finally, related work is discussed as well as an overall conclusion of this work with an outlook for future approaches is given.

1.5 Publications

This thesis mainly bases on several contributions describing aspects of this work. In the following, a list of these publications is given.

- In [131], some aspects of design patterns are presented and discussed for implementing safety in embedded systems. Embedded systems as an example from the automotive context are chosen. For this thesis, the presented design patterns had an impact on the concepts of the software framework $\mathcal{H}$esperia which is presented in Chapter 5.
- The work in [54] outlines design criteria and general considerations for the development of the autonomously driving vehicle "Caroline" which is explained in greater detail in Section 2.3. Furthermore, aspects of the international competition 2007 DARPA Urban Challenge are presented.
- The publication [10] presents a preliminary concept for testing intelligent algorithms which use continuous input data to produce discrete or continuous output data. This aspect is elaborated in Chapter 6 which presents an enhanced and integrated concept for testing algorithms from sensor- and actuator-based autonomous systems at various layers with different types of input data like sensor raw data or abstract high-level data.
- Further details of the previous contribution and its application in the project for developing "Caroline" are outlined in [11]. In that publication, first aspects about

the methodical integration of the system simulation concept in a development process which is based on the customer's requirements are presented. This concept is elaborated in Chapter 3.

- Conclusions and results from the application of the system simulation concept in the project for developing "Caroline" are presented at a workshop during the "International Conference on Robotics and Automation" in [12].
- The publication [9] along with [122] present results and conclusions from the "CarOLO" project and the vehicle "Caroline" and its performance in the international competition 2007 DARPA Urban Challenge. These contributions describe all aspects about the hardware setup, the software architecture, the algorithms used on the vehicle, and the concept of the quality assurance process.
- In [7], the automatically driving vehicle "iCar" for highways is presented. In this work, a strategical and tactical software component for deriving driving decisions and especially its quality assurance are outlined. Some more details about this vehicle are given in Chapter 2.
- The work in [18] explains and discusses an algorithm for generating synthetic sensor raw data using a GPU on the example of a single layer laser scanner. The algorithm itself is elaborated and embedded in the software framework $\mathcal{H}$esperia. The algorithm is outlined in detail in Section 6.4.6.
- In [17], the software framework $\mathcal{H}$esperia is presented. An in-depth description of the framework is given in Chapter 5.
- The publication [21] describes the so-called Berkeley Aachen Robotics Toolkit (*BART*) and its application for integrating and calibrating new sensors. The software toolkit BART combines the features of the software framework $\mathcal{H}$esperia with elements of the software framework IRT developed at the "Autonomous Ground Vehicles" group at the Center for Hybrid and Embedded Software Systems (*CHESS*) at University of Berkeley, California. In Chapter 8, the application of the software framework $\mathcal{H}$esperia for developing an algorithm to navigate a vehicle on a digital map is described.
- In [19], first ideas for using a simulation-based approach to support the software quality assurance- and evaluation-team are outlined. These ideas were developed and applied to the CarOLO project during the 2007 DARPA Urban Challenge.

2 Autonomous Ground Vehicles

The continuous example used in this thesis are Autonomous Ground Vehicles (*AGV*) as specific sensor- and actuator-based systems. This chapter introduces AGVs and gives a brief historic overview about their evolution. Finally, a technical overview describing a more generic system architecture for sensor- and actuator-based autonomous systems is presented to found the base for the further chapters.

2.1 History of Autonomous Ground Vehicles

The history of AGVs started on January 29, 1886 with patent number 37,435 given by the Kaiserliche Patentamt of the German Reich for the invention: Vehicle with gas engine [16]. That date can be regarded as the birthday of vehicles with combustion engines which changed fundamentally today's life.

The vision for driving autonomously was already presented 1939 at the World Fair in New York in the General Motors (*GM*) Pavilion [52]. In 1950 already, GM demonstrated a conceptual vehicle that was able to follow autonomously a buried cable emitting a Radio Frequency (*RF*) signal [164]. But only about 60 years later, GM's Chief Executive Officer (*CEO*) from 2003–2009 [66], Rick Wagoner, announced at the 2006 Consumer Electronics Show in Las Vegas the sale of autonomously driving vehicles by 2018 [33]. His announcement based probably on the success of the company's sponsorship for the Carnegie Mellon University in the 2007 DARPA Urban Challenge Competition [141, 147]. A more detailed description of the 2007 DARPA Urban Challenge and the participation of Technische Universität Braunschweig is provided in Section 2.3. Considering today's focus on low energy cars this goal might be changed or delayed in the near future.

The first automatically driving vehicle however was documented in 1961 from Stanford University. The robot named Cart was remotely controlled by a camera-based approach. It drove at an average speed of 0.0016m/s which meant 1m in ten to 15 minutes. The robot was extended by Carnegie Mellon University for speeds up to 0.016m/s. These

robots needed up to five minutes for computing a drivable trajectory from an acquired camera image [106].

Further work was carried out in 1977 by Tsukuba Mechanical Engineering Laboratories, Japan for the development of a robot guide dog. The robots MELDOG I–IV were equipped using ultra sonic and camera devices for obstacle detection [150, 151]. Those results led in 1979 to the development of a vehicle for camera-based road guidance. That car drove about 50m with a maximum speed of about 8m/s [159].

In the 1980s, DARPA funded a project for autonomous land vehicles. During this project, a vehicle capable to follow roads automatically using active Light Detection And Ranging (*LIDAR*) sensors was developed. Also, Carnegie Mellon University was among the project partners [4].

The first activities in Germany were carried out by the Universität der Bundeswehr München in the first half of the 1980s. The vehicle "Versuchsfahrzeug zur autonomen Mobilität und Rechnersehen", vehicle for autonomous mobility and computer vision (*VaMoR*) based on a Daimler-Benz van achieved in 1986 a maximum speed of about 25m/s on a separated highway. The car was able to drive autonomously for the lateral control [177].

Based on the success of the aforementioned VaMoR vehicle, from 1987 to 1994 the European Commission funded the Program for European Traffic of Highest Efficiency and Unprecedented Safety (*PROMETHEUS*) [26, 27]. Within this program, Daimler-Benz developed an autonomous road vehicle named "Vision Technology Application" (*VITA*) which was able to stay inside its own lane, to detect obstacles inside the current lane, to change the lane to the left or right neighbor lane initiated by the driver, and to detect intersections [160, 161]. Another vehicle resulting from the PROMETHEUS project was named "VaMoRs PKW", VaMoR's automobile (*VaMP*). That vehicle proved a long-run reliability for more than 1,000km with an average speed of 36m/s [15, 41].

Parallel to the PROMETHEUS effort in Europe, the vehicle "Rapidly Adapting Lateral Position Handler" (*RALPH*), developed by Carnegie Mellon University drove successfully from Pittsburgh, PA through eight states to San Diego, CA. The lateral control was operating autonomously and was implemented using a vision-based approach whereas the speed was controlled by a human safety driver. Its average speed was about 25m/s [117, 118, 119].

Daimler-Benz demonstrated 1994 the vehicle "Optically Steered Car" (*OSCAR*) which was an approach for vision-based adaptive intelligent cruise control [62]. In 1998, DaimlerChrysler enhanced that approach within a vehicle called "Urban Traffic Assistant"

(*UTA*) combining autonomous following a lead vehicle for highways and urban environments [57] as well as assistance in inner-city areas [65] using a vision-based approach. One main area of interest was the traffic sign and traffic light detection as well as pedestrian detection in urban environments [58, 59, 60].

From 1996 to 2001, the Italian government funded a project to realize autonomous driving using only passive sensors [29]. The vehicle named ARGO drove nearly 2,000km in June 1998 on Italian highways with a maximum distance of 54.3km without human intervention [28].

Besides independent AGVs, research also concentrates on the development of virtual vehicle platoons or automated highway systems. The European program Promote Chauffeur I from 1996 to 1998 and Promote Chauffeur II from 2000 to 2003 were carried out for virtual platooning for trucks. The main goal for those programs was the development of a virtual tow bar for saving fuel by lowering the distance between several trucks. The resulting truck platoon drove with a distance of 15m between each other at a speed of nearly 23m/s [137, 138]. The results of that program were further analyzed using a scenario-based approach subject to building a platoon, driving as part of a platoon, and leaving a platoon [79].

In California, the program Partners for Advanced Transit and Highways (*PATH*) [142] and in Japan, the program Intelligent Multi-mode Transit System (*IMTS*) [2] were set up to foster research in the similar area. The main focus for these programs is to increase the number of vehicles and the safety on highways and to lower environmental pollution.

In 1998, Volkswagen demonstrated its autonomously driving vehicle KLAUS. Using KLAUS, human exposure for test drivers could be significantly lowered [134, 173].

In 2004, DARPA announced and carried out the first Grand Challenge called “Barstow to Primm” named by the course which led from Barstow, CA to Primm, NV [139]. The course had a length of about 142 miles. The participation consisted of a qualification and a vehicle inspection prior to the final event. None of the 15 competitors completed the course and the farthest distance traveled autonomously was about 7.4 miles by Carnegie Mellon University’s team [139].

For enforcing a higher quality of the competition’s entries, DARPA announced the repetition in 2005 named “Desert Classic” and modified the entire qualification process [70]. The process consisted of a video submission demonstrating basic vehicle’s capabilities, on site inspections carried out by DARPA’s officials prior to the National Qualification Event (*NQE*). The NQE was meant to be the semifinal to select the competitors for the final event. From initially 195 competitors, only 43 teams achieved the semifinal. 23 teams

achieved the final event held on October 8, 2005 for completing the 131.6 miles course started at Primm, NV. Only five teams completed the entire course, four of them within the 10 hours time limit, from which the vehicle named “Stanley” of Stanford University won the race and the $2 million first prize [71].

Figure 2.1: Autonomously driving vehicle “Caroline” at the 2007 DARPA Urban Challenge in Victorville, CA. This vehicle was the contribution from Technische Universität Braunschweig for that international competition. The vehicle was a 2006 Volkswagen Passat station wagon which was modified for driving autonomously in urban environments. Detailed information about the vehicle, its modifications, and its performance during the competition can be found at [122].

In May 2006, DARPA announced a subsequent competition for AGVs, the 2007 DARPA Urban Challenge which also declared the goal to be reached by the competitors [170]. The qualification process was similar to the one for the previous challenge, consisting of a video submission, a demonstration at performers’ site, and a semifinal along with the final taking place in Victorville, CA at the former George Air Force Base from October 26 to October 31, 2007. 35 teams from initially 89 competitors achieved the semifinal and only eleven qualified for the final event held on November 3, 2007. The contribution of Technische Universität Braunschweig named “Caroline” shown in Figure 2.1 was among the eleven finalists and achieved as the best European competitor the seventh place [122]. Team Tartan Racing from the Carnegie Mellon University won the Urban Challenge and the $2 million first prize [141].

In spring 2008, Volkswagen together with regional research and development partners presented “Intelligent Car” at the Ehra proving ground as shown in Figure 2.2. The vehicle which based on a 2006 Volkswagen Passat station wagon demonstrated its abilities for automatic driving on highways which was demonstrated on the proving ground from Volkswagen in Ehra. For achieving that goal, the vehicle analyzes its surroundings and recommends passing maneuvers of slower vehicles. Its maximum speed was about 36m/s

[172].

Figure 2.2: Vehicle “Intelligent Car” from Volkswagen at the company’s own proving ground in Ehra. This vehicle was able to drive automatically on highways by continuously evaluating its surroundings to derive suitable driving propositions for safely passing slower moving vehicles for example. Further information about the vehicle can be found at [7]; image credit: Volkswagen Corporation.

The first European variant of a program similar to the challenges organized by DARPA was held in 2006 by Bundeswehr (German Federal Armed Forces). Unlike the American challenges, this program which is called European Land Robot Trial (*ELROB*) is not a competition but a demonstration of the capabilities of the current robotics research. The program is split into a Military ELROB (*M-ELROB*) and a Civilian ELROB (*C-ELROB*) demonstration which alternate yearly. In 2009, C-ELROB took place from June, 15 to June, 18 in Oulu, Finland [56].

The Netherlands are organizing an international challenge on cooperative driving. The goal within this challenge is to interact with other traffic participants to optimize the overall traffic flow. Currently, rules and judging of the participants are discussed. The event will probably take place in 2011 [81].

2.2 Architecture of Sensor- and Actuator-based Systems

This section describes the general architecture of sensor- and actuator-based autonomous systems from a functional point of view in general which can be also found in AGVs as mentioned in several publications [41, 122, 155, 172]. Furthermore, necessary software

and system components for proper system function are outlined as well. First, necessary definitions and terms are given before afterwards a brief overview over one concrete example is presented.

2.2.1 Terms and Definitions

In the following, a list of definitions for the following chapters is given.

- *Automatically driving vehicle.* An automatically driving vehicle is able to drive for itself under supervision of the driver. A recent popular example is an automatic parking system or "intelligent Car" [7, 172].
- *Autonomously driving vehicle.* Contrary, an autonomously driving vehicle is a robot trying to reach a predefined goal for itself without interventions of a driver. An example is "Caroline" as described shortly in Section 2.3 or the vehicle described in the case study in Chapter 8.2.1.
- *System.* This term is the short term for sensor- and actuator-based autonomous system which can be for example an automatically or autonomously driving vehicle with all components including its software.
- *Subsystem.* Hereby, either a software or hardware component from a system is denoted.
- *System's context.* The system's context denotes the environment for which the system is designed for. This includes elements from its surroundings as well as technical dependencies for correct operation. The short term *context* is used interchangeably.
- *Surroundings.* Surroundings of an autonomously driving vehicle are all visible elements like pedestrians, trees, or lane markings as well as logical elements like speed limits for example as described in Section 4.3.
- *Rigid body.* A surroundings' element which does not change its shape when translations or rotations are applied, i.e. the distances between all points forming its shape do not change. For the sake of simplification, collisions with other rigid bodies and their impact are explicitly excluded. The terms *body* and *element* are used interchangeably as well.
- *Real-time.* By this term, a context is denoted which is not entirely controllable, mostly caused by an independent time source or uncontrollable elements.

2.2.2 General System Architecture

An autonomously or automatically driving vehicle is a system interacting with the system's context by *continuously* detecting its surroundings, assessing and interpreting the detected environment for deriving an action, and performing an action. In recent literature, a sensor- and actuator-based autonomous system is also be regarded as a Cyber Physical System (*CPS*) [94].

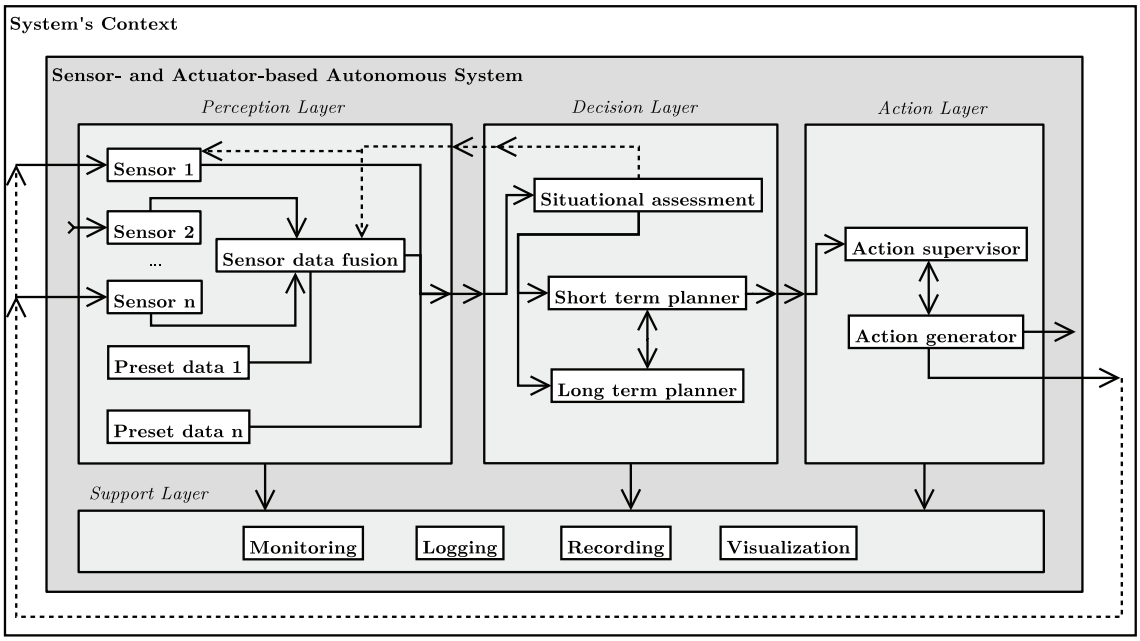


Figure 2.3: General system architecture of a sensor- and actuator-based autonomous system consisting of a *Perception Layer*, a *Decision Layer*, and an *Action Layer*. The leftmost layer is responsible for gathering information from the system's context by processing incoming data from all available sensors. Furthermore, this layer applies algorithms for fusing and optimizing potentially uncertain sensors' raw data to derive a reliable abstract representation. This abstract representation is the input data for the next layer which interprets and evaluates this abstract model to derive an action. This action is passed to the rightmost layer which calculates required set points for the control algorithms. To support the development as well as to supervise the data processing chain which is realized by the aforementioned three stages the support layer provides appropriate tools for visualizing, capturing, and replaying data streams. The overall data processing chain realized by this architecture is *closed* by the environmental system's context as indicated by the dotted arrows.

The general system architecture for sensor- and actuator-based systems is shown in Figure 2.3. It consists of three major parts which form a data processing chain to modify incoming data: *Data Perception Layer*, *Decision Deriving Layer*, and *Action Layer*. These parts describe the data flow for a sensor's datum through the system for causing some action

in the system's context while the data's complexity might decrease during processing. Beyond, an additional layer is depicted for supporting these major parts for example by simply recording the data flow.

The first layer is the *perception layer*. This layer is the adapter for continuously acquiring data from any source into the system. As shown in the diagram data may be acquired from outside the system denoted by *Sensor 1* like data from a GPS. The acquisition of internal data from the system itself like an odometer is indicated by *Sensor 2*. This data may be improved and combined with another sensor's data using a *sensor data fusion* as explained in greater detail in [75] for example. Furthermore, preset data like a digital map can be provided by this layer.

The following layer is named *decision layer* which is fed by the perception layer. In this part, all continuously incoming data is situationally interpreted to feed back information to the perception layer for optimizing the data acquisition. Mainly, the *Situational assessment* provides processed data to a short and an optional long term planner to generate a decision for the next layer. Furthermore, both planners influence each other to avoid short term decisions which do not accomplish the system's overall goal. Otherwise, if none short term decision could be derived, the long term plan must be adjusted regarding the changed situation.

The last processing stage is called *action layer* which receives its data from the decision layer. In this layer an abstract decision is split up into set points fed to controllers for the system's actuators by the *Action supervisor*. The *Action generator* represents all controllers and performs the actual low-level controlling by monitoring feedback variables to compute new actuating variables. Obviously, this generator modifies the system itself but may also influence the system's context by transmitting data as indicated by the white outer frame around the system. Shortcuts from the perception layer into the action layer which bypass the decision layer can also be found in sensor- and actuator-based autonomous systems to reduce the latency for example. However, in Figure 2.3 these shortcuts are not explicitly shown.

All aforementioned elements and layers are assisted by the *support layer*. The layer itself is not part of the data processing chain but maintains the chain by logging events, recording or simply visualizing the data flow, and monitoring the system's overall state. Due to the system's interactions with the reality in perhaps critical applications like an Adaptive Cruise Control (*ACC*) system, the support layer must not interfere the actual data processing chain. Thus, its operation must be non-invasive and non-reactive.

The system architecture for a sensor- and actuator-based autonomous system is common for many AGVs. Thus, this architecture of a continuously data processing system interact-

ing with its surroundings is the base for all further chapters. In the following, an example for an AGV implementing the aforementioned model is presented.

2.3 Example: Autonomously Driving Vehicle “Caroline”

In this section, an overview of the autonomously driving vehicle “Caroline” is presented. “Caroline”, named according to the founder of Technische Universität Carolo-Wilhelmina at Braunschweig, Carl I., is a 2006 Volkswagen Passat station wagon. It is able to drive autonomously in urban environments and competed with only ten other teams in the 2007 DARPA Urban Challenge final event [140].

(a) Caroline’s sensor setup for the front side: Several sensors with different measuring principles and overlapping viewing angles and ranges are used to perceive reliably its surroundings. To detect stationary and dynamic obstacles like parked or moving cars, active sensors like laser scanners and radars are used; for sensing lane markings or drivable areas cameras with different resolutions were used.

(b) Caroline’s trunk which carries the processing units: Automotive computers were used running a Debian Linux with a special real-time Linux kernel.

Figure 2.4: Overview of Caroline’s sensors and trunk (based on [122]).

In Figure 2.4, the general setup of the vehicle is shown. On the left hand side, the sensors for environmental perception are depicted, whereas the right hand side shows the computing units in the trunk. “Caroline” was equipped with a redundant sensor configuration to improve the perception’s reliability on one hand and to enhance the abstract environmental model produced from the sensor data fusion by using different measuring principles for overlapping fields of view on the other hand.

Depending on the desired perception of the surroundings’ elements, different types of sensors were used. “Caroline” detected the road and drivable area in front of the vehicle, lane markings from the own and the left and right neighbor lanes, and stationary and

dynamic obstacles. At the rear side, sensors to perceive stationary and dynamic obstacles only were mounted.

Starting at the lowermost surroundings' element to detect, "Caroline" used a color camera combined with two one layer laser scanners to detect the shape and the course of drivable terrain up to about 50m at 10Hz. Using this information, the vehicle could drive on the road by centering itself between curbs or on similar colored and shaped ground [20]. For using the correct side of a road, potentially available lane markings were detected using a setup of four color cameras looking up to 35m with a Field of View (*FOV*) of 58° at 10Hz as well [99].

For detecting stationary and moving obstacles, only active sensors with varying measuring principles for different distances were used. To detect obstacles at the farthest distance, a beam LIDAR sensor for up to 200m with a FOV of 12° was mounted below the license plate. For a FOV of 240° up to 60m, two scanning four layers laser scanners operating as a fusion cluster were mounted on the left and right front section. Between both sensors, a radar sensor covers the FOV of 40° up to 150m. From all these sensors, the raw or pre-processed data by the sensor's own ECU was fused using a sensor data fusion to generate an abstract environmental model from the vehicle's surroundings [45].

Following the perception layer, the decision layer analyzed and interpreted the pre-processed data as next stage in the data processing chain. "Caroline" used an approach based on [128] for generating weighted curvatures using its situational assessment. The highest rated curvature was chosen to compute a trajectory to be driven by the action layer. For controlling the vehicle in rule-based situations, this approach was enhanced with an interceptive system taking control at intersections or for performing a U-turn for example.

Using trajectories from the previous stage, the action layer supervised the vehicle's overall state and computed the necessary set points for the actual controlling algorithms. Furthermore, preset complex maneuvers like performing a U-turn could be commanded from the decision layer by issuing a command providing an appropriate U-turn polygon to be used as the area to carry out a U-turn [174, 175].

The vehicle's support layer consisted not only of simple logging and supervising components for run-time. Moreover, a graphical run-time visualization [98] to be used also as front-end for the simulation of "Caroline" during development was provided [11]. Further information in greater detail can be found in [9, 122].

3 Methodology for Automating Acceptance Tests

In this chapter, a methodology for *automating acceptance tests* during the system development is presented. Furthermore, preconditions, requirements, and necessary concepts for implementing a supporting tool chain are derived which are elaborated in the following chapters.

3.1 General Considerations

Referring to [31], [120], and [162], related research challenges especially in automotive software and systems engineering–which can also be applicable for similar domains for system development relying on sensors and actuators–for this work include nowadays:

- *Languages, models, and traceability for requirements engineering.* Approaches for handling requirements must include the customer's point of view. However, since most requirements are provided in natural language, they cannot be directly part of the software and system development process by definition.

  Furthermore, as required by process audits to achieve a certain level of certification, not only requirements which might stem from customers but also change requests caused by identified faults in the software or due to changed circumstances must be tracked down to single lines of code. Hence, a machine-processable integration of these requirements into the system and software development process is desirable to support these tasks.

- *Middleware to enable communication between heterogeneous systems.* Today's vehicles are equipped with different communication infrastructures, for example CAN-, Flexray-, or LIN-buses; even newer ones are evaluated in research departments of various OEMs. Moreover, Road Side Units (*RSU*) enabling communication between a vehicle and its infrastructure need to be covered by technical

approaches which require a reliable, safe, and uncompromised communication between sender and one or many receivers. Furthermore, due to changing hardware platforms and thus changing ECUs, software shall be robust and thus embrace these changes with only a minimum of additional system tests.

- *Continuous integration of evolving software artifacts.* Well supported by tools for the Java programming language, this aspect is currently quite time-consuming for sensor- and actuator-based autonomous systems due to insufficient tools. This is caused by the dependency on sensor's input data to stimulate the data processing chain. Furthermore, sometimes complex situations must be set up in the system's context to evaluate an SUD's actions.

In the following, the generic system architecture for sensor- and actuator-based autonomous systems is analyzed to identify requirements and preconditions related to the V-model development process to derive a development methodology which is independent from the real existing hardware already at early stages. Moreover, this methodology shall allow an automation of acceptance tests by deriving the system's context of an SUD based on the customer's requirements in a machine-processable manner to integrate it directly into the system- and software development process for addressing the aforementioned issues.

3.2 Virtualizing a Sensor- and Actuator-based Autonomous System for the V-model-based Development Process

In Figure 3.1, the V-model alongside with the previously discussed general system architecture for sensor- and actuator-based autonomous systems is depicted. The development process starts on its highest level with an analysis of the customer's requirements. For the development of a sensor- and actuator-based autonomous system, these requirements describe in general the customer's *observable behavior* of the system in its intended system's context as shown in Figure 2.3. From a different point of view, these requirements describe the "interface" and "behavior" of the system.

When developing sensor- and actuator-based autonomous systems, the software development depends not only on the real hardware environment including actuators and sensors with their corresponding mounting positions and fields of view, but also on the system's context causing various and often unlimited different situations which must be handled

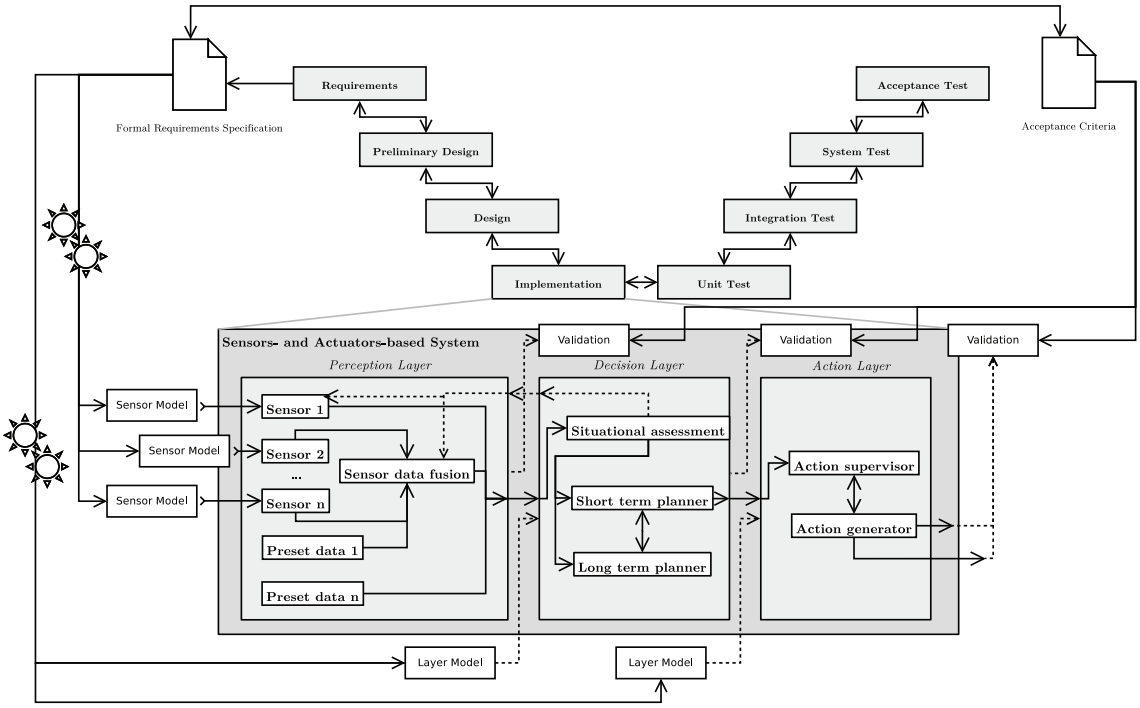


Figure 3.1: Methodology which uses the customer's requirements from the topmost level of the V-model as the basis for a formal specification of the system's context. This specification is used to generate the required input data for the different layers of a sensor- and actuator-based autonomous system. Moreover, completing the customer's requirements, its acceptance criteria are used to specify metrics for evaluating individual layers or the entire data processing system.

safely and in time by the SUD. Thus, several limitations unveil for the software engineering which are selectively listed in the following:

- *Dependency on the sensors may delay the software development.* Due to the fact that the sensors' raw data is necessary to develop appropriate software components to evaluate and interpret the system's context for generating an abstract model of its surroundings, the development of these components is delayed until situation-dependent raw data can be produced even if the software architecture is decomposed into layers with interfaces. Thus, the availability and mounting of sensors is on the project's critical path.

- *Early and continuous integration of software components is hardly possible.* Followed by the aforementioned issue, the continuous and early integration of independent software artifacts is hardly possible. Thus, issues which arise from the component's interaction or from architectural faults cannot be fixed early and cause delays later on.

- *Test automation is only possible in a limited manner.* Caused by the dependency on sensors' raw data, automation for unit tests and thus an automated continuous integration system is only available for selected parts of the software system. Thus, a report about the quality and robustness for the entire system is only possible when it is integrated and tested in reality which is impossible for each single modification to the entire source code.
- *Dependency on continuous input data from the system's context.* To feed required input data into the data processing chain of a sensor- and actuator-based autonomous system, its actions within its intended system's context cause on one hand reactions in the surroundings. On the other hand, these reactions of all elements from the system's context must be returned into the data processing chain in terms of a closed-loop. However, the closing of the data processing chain can be realized at different layers of the chain as shown in Figure 2.3. Therefore, various abstractions from the system's context are required for the different layers; for example, the necessary input data for a sensor data fusion module needs to be more complex compared to the input data which is necessary for a high-level planning algorithm which may only require an aggregated abstraction from the system's context.

In the following, a methodology is outlined which addresses these challenges for the software engineering in the development of sensor- and actuator-based autonomous systems.

3.2.1 Preconditions and Limitations

For applying recent methodologies from the software engineering domain including test automation to all parts of the system's software, several preconditions must be met. These are listed in the following.

- *Formal specification of the system's context.* For developing a sensor- and actuator-based autonomous system which relies on stimuli from its intended system's context, input data must be available to provide information about its surroundings. To get this data in a reliable and consistent manner, the SUD's system's context including stationary and dynamic elements must be specified formally to provide scenarios. These scenarios are re-usable artifacts which are machine-processable in the software development process and can be derived from the customer's requirements.
- *Specification of customer's acceptance criteria.* For carrying out acceptance tests to evaluate the behavior of the SUD, metrics which reflect the customer's acceptance

criteria must be specified. These metrics are used to evaluate the SUD's behavior on stimuli from the stationary and dynamic system's context.

- *Decoupling the system and software development process from the real hardware environment and the system's context.* To realize an automatic validation of the SUD, the software and system development must be operable in a purely virtualized manner. Thus, the SUD itself must be available in a completely virtualized way.

  Therefore, the CPS's interface to its real world must be virtualized. Hence, a model of each sensor and actuator must be available to decouple the SUD from its surroundings. Furthermore, a fail-safe, reproducible, and convenient framework to support the active software development as well as the automated validation must be available which is able to provide the required input data for a data stream processing system using sensors to understand its surroundings and actuators to interact with its surroundings.

- *Enabling the testability.* To apply software testing methods on the SUD, the system itself and especially its components which may be supplied by third party contributors must be testable. Thus, a testable architecture must be available which supports the testing of individual layers or even subsystems.

However, a purely virtualized system and software development process can extend and partly substitute real system tests but cannot completely substitute them. The main reasons are the simplifications and assumptions which were made in developing a model to feed data into the SUD. Instead, the main goal behind a software development process which bases on a virtualization of the SUD is to reduce the dependencies to the real hardware environment and the system's context to increase the overall SUD's software quality by enabling the usage of well established software engineering methodologies like test automation and continuous integration. Thus, an increase in efficiency for evaluating the system's quality is possible by using automatable software tests.

3.2.2 Formal Specification of the System's Context

For using the customer's requirements as part of the software development process to specify the SUD's system's context as shown in Figure 3.1, a formal representation in form of a so-called formal language is necessary. Nowadays, these languages which help to address selected issues of a chosen domain are called DSLs contrary to so-called General Purpose Languages (*GPL*) which do not cover a specific domain but rather provide more widely applicable concepts. A DSL consists of a set of specific words which are called

the *alphabet* of the language or which are more commonly known as *keywords*. Furthermore, these keywords must be composed in a formally specified way which is defined in a so-called language's grammar to create valid sentences which meet the grammar.

The main advantage of grammar's compliant sentences is that any tools which use them for further data processing can safely assume that the provided input data is valid if and only if the sentence is compliant to the grammar. Thus, the content of a given sentence can safely be evaluated if it is formally correct; this fact is called *syntactical correctness*.

However, syntactical correctness is necessary but not sufficient i.e. any sentence which is compliant to a given grammar must not be automatically meaningful. Therefore, further concepts like the Object Constraint Language (*OCL*) which is part of the Unified Modeling Language (*UML*) can be used to specify constraints for parts of a sentence. Another way to evaluate the semantic validity of a sentence is to perform a post-processing step after evaluating the sentence's syntactical correctness during the so-called parsing step. Therefore, all atomic pieces of a sentence are evaluated using a specific program which is often part of the *parser*.

To create a suitable DSL, the domain has to be carefully analyzed by a domain expert first. This analysis yields meaningful atomic pieces of the domain with potential attributes which describe varying parts. Moreover, relations between these pieces are defined and further specified. The resulting artifact is for example a UML class diagram which reflects identified elements from the analyzed domain. The diagram itself is the base to derive necessary non-terminals and terminals for the grammar. In Chapter 4, an analysis for the domain for sensor- and actuator-based autonomous systems on the example of autonomously driving vehicles is carried out which yielded a grammar to describe the system's context.

The resulting grammar is not only the base to define the system's context for autonomous vehicles. It is also the Single Point of Truth (*SPOT*) to generate the required input data for the different layers of a sensor- and actuator-based system as shown in Figure 3.1 on the left hand side of the general system architecture for sensor- and actuator-based systems. Therefore, further algorithms which are described in greater detail in Chapter 6 evaluate the given grammar's instances to provide the required input data.

3.2.3 Specification of Customer's Acceptance Criteria

As already mentioned unit tests are nowadays an established methodology [130] to define an executable specification for an expected algorithm's behavior. Therefore, the considered algorithm is structurally analyzed to derive necessary test cases; however, during the

actual test process the algorithm is regarded as a black box. The analysis is necessary to identify the required test data which is fed into the algorithm to compare the algorithm's results with the expected output data. Moreover, using continuous integration systems like [38] the test execution can be automated to report regularly on the software's quality unattendedly.

Inspired by the aforementioned methodology this concept shall be applied to the final step of the V-model for automating the acceptance tests. Therefore, the necessary input data is generated from the detailed and formal specification of the system's context either to feed data into the sensor models or into a model for an entire data processing layer as shown in Figure 3.1. Contrary to the structural analysis which is necessary to specify appropriate unit tests for an algorithm as sketched above, the customer evaluates the resulting system mainly by validating its behavior according to the customer's own acceptance criteria.

Additionally to the formally specified system's context which is derived from the customer's requirements, the customer's acceptance criteria must be gathered to define appropriate acceptance tests; recent requirements management tools like the Volère Requirements Specification [167] enforce to define so-called *fit criteria* to support the evaluation of the requirement's fulfillment. Adapting this concept to the area of sensor- and actuator-based autonomous systems the final evaluation consists of measuring various aspects of the system's behavior in its intended context. On the example of autonomous vehicles, these criteria include minimum and maximum distances to stationary and dynamic obstacles, timings, and legally correct behavior in different situations. During the 2007 DARPA Urban Challenge competition, a specific document was released [39] which describes very detailed the evaluation criteria which were applied manually by the DARPA judges during the competition.

These criteria which measure the system's performance in its intended system's context can also be realized alike the evaluation of an algorithm using unit tests. Therefore, comparable to the system's evaluation in reality the system must be supervised continuously and any applicable evaluation criteria must be applied continuously to the SUD regarding its system's context.

3.2.4 Decoupling the SUD's Evaluation from the Real Hardware Environment and System's Context

To realize an evaluation for a sensor- and actuator-based system as shown in Figure 3.2, its system's context is necessary. As mentioned before, this system's context is derived

from the customer's requirements specification which is provided using a DSL. Thus, a formal, repeatable, and consistent representation of an SUD's surroundings is available.

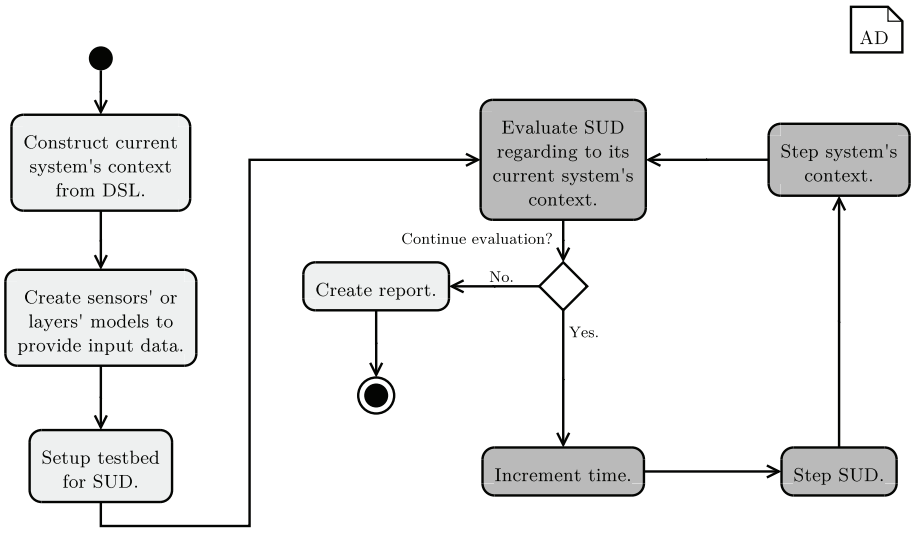


Figure 3.2: Activity diagram showing the required steps to evaluate automatically a sensor- and actuator-based autonomous system: First, on the left hand side the steps for setting up the test environment are shown. Afterwards, the iterative evaluation of the SUD's behavior is depicted by the gray steps which are continuously executed until the evaluation is finished due to fulfilling all customer's acceptance criteria for the specified test or due to violating a certain criterion. Finally, a summarizing report is created.

To feed appropriate input data into the SUD, a model of its sensors which simulates a specific sensor's measurement principle or at least an abstraction from its sensors by using a so-called layer's model is necessary. The latter represents an entire layer encapsulating all available sensors by providing already pre-processed data which reflects an abstract representation of the SUD's surroundings. Thus, the SUD's interface to its surroundings is specified.

As the next step, the so-called testbed is setup for the SUD. The testbed is a run-time environment for controlling the scheduling for the SUD and its system's context to ensure a repeatable and consistent execution for all SUD's components and its surroundings, supervising the entire communication, and providing a system-wide time source.

Following, the actual evaluation loop is executed. Therefore, the initial SUD's state related to its system's context is evaluated. Then, the system-wide time is incremented to calculate the next states for $t = t + 1$ for the SUD on one hand and its system's context on the other hand. Within this time step, the SUD may calculate next values for the actuating variables or process incoming data using its situation assessment module for example.

After modifying the SUD, its execution is suspended to generate the data for the SUD's surroundings for the current time step based on the SUD's behavior. Therefore, specific models which are using the system's context to calculate and provide the necessary data for the different layers of the SUD are required. Finally, the current situation is evaluated to start over again.

The aforementioned evaluation reflects not only a purely virtualized way to evaluate a software system for sensor- and actuator-based autonomous systems but it also resembles the way the real system may be tested: First, the surroundings and a given situation must be prepared; then, the SUD must be set up followed by an initial evaluation of the current test setup. Then, the SUD is executed with the given parameters to operate in the prepared system's context and situation while it is continuously supervised by a test team for example. Afterwards, a final report for the test is created. Hence, this real test scheme for a sensor- and actuator-based autonomous system was the template for purely virtualized evaluation.

Compared to methodologies like Software-in-the-Loop (*SiL*) and Model-in-the-Loop (*MiL*), this methodology bases on the SPOT principle realized by a DSL to specify the system's context. Thus, instances from this DSL are used as the only source to generate automatically all necessary and specific input data to close the loop for the various layers of the data processing chain. Furthermore, the DSL and the concepts outlined in the following chapters are embedded into a common framework to provide a seamless integration of the validation concepts for acceptance tests which themselves are inspired by unit tests. Finally, due to the integration into a framework, the methodology is self-contained in general and, thus, scales with the number of users. Thus, it can be used interactively on one hand to evaluate an SUD, but it can also be unattendedly automated on the other hand by any existing Continuous Integration System (*CIS*) as described for example in Section 8.2.5.

3.2.4.1 Interceptable and Component-based Applications

An SUD may consist of several only loosely coupled independent components to tackle the customer's desired systems using the *divide and conquer* principle. Furthermore, to support the scheduler for the SUD and its surroundings as described in the following, all components must be structured similarly in a software architectural sense. Therefore, the data processing part of a component which is a part of the data processing chain can be identified from the component's software architecture in a pre-defined way, and thus, the component can be intercepted easily.

Moreover, to support a consistent setup of the testbed for all independently running SUD's components, centralized configuration data must be deployed to all components nonetheless whether they are running on only one computing node or distributedly on several computing nodes. Furthermore, only those data which is particularly meant for a specific component should be deployed to avoid side-effects or dependencies on other unrelated configuration data.

3.2.4.2 Non-Reactive and Supervisable Communication

Comparable to a consistent internal structure of all running SUD's components, their communication must be precisely specified and typed as well to inspect incoming data at run-time to avoid misleading behavior inside a component. Furthermore, to avoid interfering SUD's components or evaluating components, directed point-to-point communication between a priori known communication partners must not be used for the core data processing components. Instead, a fast and bus-like communication which allows non-reactive inspection both for monitoring components and for evaluating components shall be chosen.

However, bus-like communication may cause packet drops caused by an increasing communication load. Assuming components which communicate with a high frequency for updating their previously sent data with new information, packet drops may be neglected if they appear rarely and thus do not reduce the data's quality of all awaiting components. However, this issue depends on the final applications' partitioning on the intended hardware platform. Real-time aspects for data processing and exchange depend on an actual implementation. Therefore, these aspects are discussed in Section 8.1.

Furthermore, not only the actual data processing step inside an SUD's component must be interceptable but also its incoming and outgoing communication to enforce a deterministic scheduling. However, bus-like communication is undirected in principle and hence, a software abstraction must be seamlessly integrated into all SUD's components to intercept and control all communication.

3.2.4.3 Scheduler for SUD and its System's Context

As indicated by Figure 3.2 the evaluation of a running SUD inside its predefined system's context needs a reliable scheduler. The main task for the scheduler is to control all SUD's components and all components which are used for evaluation to ensure a deterministic execution order. Furthermore, the scheduler selects one component which needs to

be executed and performs one single and constant execution step $\bar{\Delta}t$ for the main data processing part of the selected component.

Moreover, the scheduler must ensure that a component which is selected for execution does not run forever; therefore, the selected component has a predefined maximum execution time $\bar{t}_{\text{exec,max}}$. Whenever one component misses this predefined deadline the entire evaluation is canceled because the scheduler cannot identify the reason for the failing component. This behavior is a fail-safe behavior because it cancels the evaluation at the earliest possible time t_{fail} which does not cause further failures or misleading behavior. Furthermore, it may be possible to deduce the failure reason from any non-reactively captured data during the evaluation.

The strategy to select a component for execution is inspired by the *Round-robin* scheduling method which implements a First-In-First-Out (*FIFO*) queue to select the next available component. Thus, a constant run-time frequency for each component together with the aforementioned execution deadline $\bar{t}_{\text{exec,max}}$ ensures that all executable components will be selected in a deterministic order with a maximum execution time.

3.2.5 Structure for the Following Chapters

The resulting methodology consists of three parts: Formal specification of the system's context, definition of metrics for evaluating an SUD's behavior inside its system's context, and an appropriate software framework which not only supports the aforementioned both concepts but also supports the software development itself. Hence, this methodology is elaborated in the following chapters.

Therefore, a thorough domain analysis which is described in Chapter 4 is carried out for the system's context of sensor- and actuator-based systems on the example of autonomous vehicles. Afterwards, a software framework to support the development of distributed real-time applications is designed and outlined in greater detail in Chapter 5. This framework is intended to found the basis for interactive and unattended system simulations by using a DSL which describes the system's context as outlined in Chapter 6. The actual SUD's evaluation is described in greater detail in Chapter 7. The software solution is applied exemplarily on an autonomous vehicle whose results are described in Chapter 8.

4 Modeling the System's Context

In this chapter, a DSL and its implementation for Java and C++ for modeling the system's context mainly for AGVs is outlined. Therefore, the mathematical basis is discussed first. Afterwards, an analysis of the AGV's surroundings is carried out to derive a DSL for modeling stationary and dynamic elements. Finally, the DSL's implementation and its drawbacks for Java and C++ are discussed.

4.1 General Considerations and Design Drivers

According to [24], a model is a simplified representation from sometimes complex relations, actions, or behaviors of a real system and is intended to serve specific and domain-dependent purposes. The important remark is that the model is an abstraction of the original system by defining a set of assumptions to satisfy the desired purpose; thus, the model is only similar and not the same as the original system. Using a model, information from the original system should be deduced or predictions should be derived to allow further inspections for the system.

For modeling in the domain of automotive software development, MATLAB and Simulink [154] are today's first choice by many developers. Using these tools for specifying a system, even stepwise interpretable models at early stages in the software development process could be defined. However, these models are rather inapplicable for large models describing elements in the system's context along with its visual appearance. Furthermore, for designing stationary and dynamic elements with an associated behavior that could be event-based itself, MATLAB is rather inappropriate because to its limited concept of referencing variables for example.

Below, the most important design drivers for a domain specific language for modeling an AGV's system's context are listed:

- *Scenario-based modeling.* The topmost modeling element is a scenario defining a concrete setup for the system's context as well as a goal to be reached by the system

itself within the modeled context. For example: “Scenario: Correct handling of 4-way stops.”

- *Separation of modeling.* Stationary and dynamic elements of the system’s context should be modeled separately for allowing flexible reuse.
- *Mathematical relations as common base.* Scenarios describe complex relations between different elements from the system’s context like positions and rotations, timings, and velocities. All these attributes arise from consistent and unique mathematical relations between element’s attributes from the system’s context. Therefore, for using models from the system’s context in the software development to generate input data for example, they must rely on a mathematical base.

Next, mathematical considerations for modeling three-dimensional elements of the reality are given. Afterwards, a domain specific language to define an AGV’s system’s context is introduced by analyzing the AGV’s surroundings.

4.2 Mathematical Considerations

Before the surroundings of stationary and dynamic elements can be modeled, their mathematical representation and valid manipulations must be defined. In this section, manipulations for rigid bodies which are used in the surroundings’ representation to simplify the modeling and computation are discussed. Further information for the concepts which are outlined briefly in the following can be found at [30].

First, every rigid body has a unique representation in the model. Since the model itself is derived from the reality, its assumed modeling space is $\mathbb{R}^3$ with Cartesian coordinates from the orthonormal basis as shown in Equation 4.1. A rigid body’s elementary representation P_B in $\mathbb{R}^3$ consists of a translation P_{B_T} relative to the coordinate system’s origin and a direction vector P_{B_D} describing its rotation around all three axes: $P_B = (\vec{P_{B_T}}, \vec{P_{B_D}}) = ((x_B, y_B, z_B), (d_{X_B}, d_{Y_B}, d_{Z_B}))$, where P_{B_T} itself denotes a fixed and immutable point over time in the rigid body itself, for example its center of mass.

$$\mathbb{R}^3 = (\vec{e_X}, \vec{e_Y}, \vec{e_Z}) = \left(\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \right) = E. \tag{4.1}$$

For modifying the body P_B without changing the body’s shape, translations, rotations,

and reflections are the only allowed manipulations respecting P_B's orthonormal basis. Since manipulations applied to the body P_B in modeled surroundings must always be continuous, it is evident that reflections however are not permitted.

4.2.1 Manipulations in $\mathbb{R}^3$

A translation applied to P_B is an addition of an arbitrary translation vector $\vec{t_B} = (x_{t_B}, y_{t_B}, z_{t_B})$ to $\vec{P_{B_T}}$. The rotation of P_B is defined as an application of a given angle $\theta \in [-2\pi; 2\pi]$ around the X-, Y- or Z-axes from the orthonormal basis specified as rotation matrices R_{θ_X}, R_{θ_Y}, or R_{θ_Z} as shown in Equation 4.2. It can be easily shown that $\det(R_{\theta_X}) = \det(R_{\theta_Y}) = \det(R_{\theta_Z}) = 1$ holds for any θ. Furthermore, all column vectors from the rotations R_{θ_X}, R_{θ_Y}, and R_{θ_Z} found an orthonormal basis themselves.

$$
\begin{aligned}
R_{\theta_X} &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta_X) & -\sin(\theta_X) \\ 0 & \sin(\theta_X) & \cos(\theta_X) \end{pmatrix}, \\
R_{\theta_Y} &= \begin{pmatrix} \cos(\theta_Y) & 0 & -\sin(\theta_Y) \\ 0 & 1 & 0 \\ \sin(\theta_Y) & 0 & \cos(\theta_Y) \end{pmatrix}, \\
R_{\theta_Z} &= \begin{pmatrix} \cos(\theta_Z) & -\sin(\theta_Z) & 0 \\ \sin(\theta_Z) & \cos(\theta_Z) & 0 \\ 0 & 0 & 1 \end{pmatrix}.
\end{aligned} \tag{4.2}
$$

For performing a rotation θ around an arbitrary axis denoted by $\vec{w} = (w_X, w_Y, w_Z)$, the problem can be formulated as a rotation around a known axis X, Y, or Z. In Equation 4.3, the rotation is reduced to the rotation around the X-axis.

$$
\begin{aligned}
R_{\theta_w} &= R^{-1} \cdot R_{\theta_X} \cdot R = R_{\theta_{(x_w,y_w,z_w)}}, \qquad (4.3)\\
&with\\
R &= \begin{pmatrix} \vec{w}^T \\ \vec{s}^T = (\frac{\vec{w}\times\vec{e_X}}{\|\vec{w}\times\vec{e_X}\|})^T \\ (\vec{s}\times\vec{w})^T \end{pmatrix},\\
R_{\theta_{(x_w,y_w,z_w)}} &= \begin{pmatrix} C+Tx_w^2 & Tx_wy_w - Sz_w & Tx_wz_w + Sy_w \\ Tx_wy_w + Sz_w & C+Ty_w^2 & Ty_wz_w - Sx_w \\ Tx_wz_w - Sy_w & Ty_wz_w + Sx_w & C+Tz_w^2 \end{pmatrix},\\
&with\\
C &= \cos(\theta_{(x_w,y_w,z_w)}),\\
S &= \sin(\theta_{(x_w,y_w,z_w)}),\\
T &= (1-C).
\end{aligned}
$$

If $\vec{w}$ and $\vec{e_X}$ are parallel, w.l.o.g. $\vec{e_Y}$ can be used for defining the aforementioned equations. For simplifying rotations, Eulerian angles can be used separately for X-, Y-, and Z-axes using rotation matrices R_{θ_X}, R_{θ_Y}, and R_{θ_Z}, respectively. All these matrices can simply be multiplied regarding the order of rotation.

To express a rotation and translation in one single matrix, homogeneous coordinates are necessary. Transforming the aforementioned matrices and vectors into homogeneous coordinates is trivial as shown in Equation 4.4. Furthermore, mappings like perspective projections as shown in Equation 4.5 are easily possible.

$$
\begin{pmatrix} r_{(1,1)} & r_{(2,1)} & r_{(3,1)} \\ r_{(1,2)} & r_{(2,2)} & r_{(3,2)} \\ r_{(1,3)} & r_{(2,3)} & r_{(3,3)} \end{pmatrix} \; and \; \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} \mapsto \begin{pmatrix} r_{(1,1)} & r_{(2,1)} & r_{(3,1)} & t_x \\ r_{(1,2)} & r_{(2,2)} & r_{(3,2)} & t_y \\ r_{(1,3)} & r_{(2,3)} & r_{(3,3)} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad (4.4)
$$

$$
\underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{orthogonal\ projection} \cdot \underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{d} & 1 \end{pmatrix}}_{perspective\ transformation} \cdot \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{d} & 1 \end{pmatrix}}_{perspective\ projection} \cdot \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} \qquad (4.5)
$$

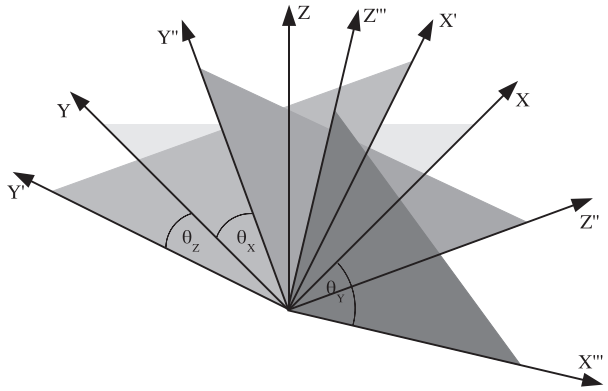


Figure 4.1: Three-dimensional coordinate system with rotations around all three axes. The triangles with the different gray-tones are additionally drawn helping to identify the rotations.

The Cartesian coordinate system used for modeling is shown in Figure 4.1 based on a right-hand-coordinate system. It is also the base for $\mathcal{H}$esperia as described in Chapter 5.

4.2.2 Quaternions

As already mentioned before, rotations can be expressed using rotation matrices around X-, Y-, and Z-axes representing Eulerian angles. However, this representation can be erroneous when one axis aligns with another during rotating causing the loss of one degree of freedom as shown in Equation 4.6. In literature, this problem is known as *Gimbal lock* [96].

$$
\begin{aligned}
R &= R_{\theta_Z} \cdot R_{\phi_X} \cdot R_{\psi_Z} \qquad (4.6)\\
&= \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi) & \cos(\phi) \end{pmatrix} \cdot \begin{pmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{pmatrix} \\
&\underset{\phi=0}{=} \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot E \cdot \begin{pmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} \cos(\theta+\psi) & -\sin(\theta+\psi) & 0 \\ \sin(\theta+\psi) & \cos(\theta+\psi) & 0 \\ 0 & 0 & 1 \end{pmatrix}
\end{aligned}
$$

For avoiding the Gimbal lock problem, a *quaternion* $q \in \mathbb{H}$ can be used. Quaternions are so-called hyper-complex numbers $q = a + bi + cj + dk$ with $i^2 = j^2 = k^2 = ijk = -1$. They provide the homomorphism $(\mathrm{SO}(3), \| \cdot \|)$ by interpreting a quaternion as a rotation θ around an arbitrary axis as $\vec{w}$: $q = (cos(\frac{\theta}{2}), sin(\frac{\theta}{2})\vec{w})$. To compute a rotation in $\mathbb{R}^3$, the quaternion multiplication as shown in Equation 4.7 is used.

$$\begin{aligned} \vec{p}' &= q \cdot \vec{p} \cdot \overline{q} \\ with & \\ \cdot &\equiv \text{quaternion multiplication} \\ \overline{q} &\equiv \text{conjugate of q: } \overline{q} = (cos(\frac{\theta}{2}), -sin(\frac{\theta}{2})\vec{w}) \end{aligned} \tag{4.7}$$

Using quaternions as an alternate representation for rotations, the Gimbal lock problem cannot arise since rotations around several axes are expressed as concatenated quaternion multiplications. For expressing a rotation of point $\vec{p}$ around an arbitrary rotation axis $\vec{w}$ by θ in $\mathbb{R}^3$, the quaternion correspondence as shown in Equation 4.8 applies [73, 96, 108].

$$\begin{aligned} R_{\theta_{(w_x,w_y,w_z)}} \cdot \vec{p} &\mapsto q(\frac{\theta}{2}, \vec{w}) \cdot \vec{p} \cdot \overline{q}(\frac{\theta}{2}, \vec{w}) \\ &= (\cos(\frac{\theta}{2}) + \sin(\frac{\theta}{2})\vec{w}) \cdot \vec{p}(\cos(\frac{\theta}{2}) - \sin(\frac{\theta}{2})\vec{w}) \\ &= (\cos(\frac{\theta}{2}), \sin(\frac{\theta}{2})w_x, \sin(\frac{\theta}{2})w_y, \sin(\frac{\theta}{2}w_z) \cdot \vec{p} \cdot \\ &\quad (\cos(\frac{\theta}{2}), -\sin(\frac{\theta}{2})w_x, -\sin(\frac{\theta}{2})w_y, -\sin(\frac{\theta}{2})w_z) \\ &= \begin{pmatrix} 1-2(y^2+z^2) & -2vz+2xy & 2vy+2xz \\ 2vz+2xy & 1-2(x^2+z^2) & -2vx+2yz \\ -2vy+2vz & 2vx+2yz & 1-2(x^2+y^2) \end{pmatrix} \cdot \vec{p} = \vec{p}' \\ with & \\ & v = \cos(\frac{\theta}{2}),\ x = \sin(\frac{\theta}{2})w_x,\ y = \sin(\frac{\theta}{2})w_y,\ z = \sin(\frac{\theta}{2})w_z \end{aligned} \tag{4.8}$$

4.2.3 The WGS84 Coordinate System

For locating positions on Earth, two-dimensional World Geodetic System 1984 (*WGS84*) coordinates are widely used today [109]. Even the well-known GPS service bases on these

coordinates. This coordinate system considers the Earth's curvature on its surface. The aforementioned Cartesian coordinate system however assumes a planar surface without a curvature on its surface.

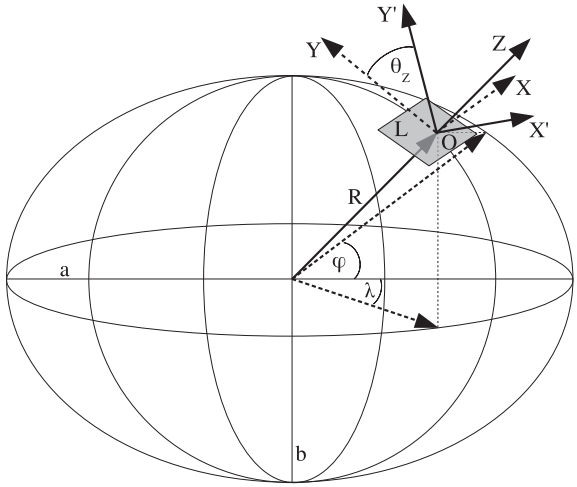


Figure 4.2: Cartesian coordinate system based on WGS84 origin. a denotes the the equatorial diameter, b denotes the North/South diameter. The triple (R, φ, λ) which consists of the radius and a spherical coordinate in WGS84 denotes a coordinate on the Earth's surface pointing to the origin O of a Cartesian coordinate system L. The normal vector for this coordinate system is described by the aforementioned triple.

In Figure 4.2 the geometric model of the Earth is shown wherein a denotes the equatorial diameter, b denotes the North/South one, and its ratio $\frac{a-b}{a}$ describes the Earth' flattening. Using the WGS84 model, $a = 6,378,137m$ and $b = 6,356,752.314m$ apply.

For mapping a Cartesian coordinate system on a curved surface in the spherical coordinate system (R, φ, λ), the following model can be assumed. Let O be the perpendicular point of plane L located orthogonally to R representing the plane's normal vector. Additionally, the Cartesian coordinate system itself can be rotated around its Z-axis using θ_Z. For getting a correspondence to spherical coordinates, $\theta_Z = 0$ is mapped to a Cartesian coordinate system, whose Y-axis directs from South to North.

For mapping spherical coordinates onto the required Cartesian coordinate system using $O = (R, \varphi, \lambda)$ as origin, a projection of the Earth's surface is necessary. The simplest projection is a cylindrical projection using a cylinder wrapped around the Earth at the equator. However, the major disadvantage is the increasing imprecision towards the poles. More precise is the *poly-conic projection* as shown in Figure 4.3. Hereby, several cones, which are tangent to Earth at different latitudes, are chosen instead of a cylinder. The main advantage is the precise mapping of the Earth's surface area [49].

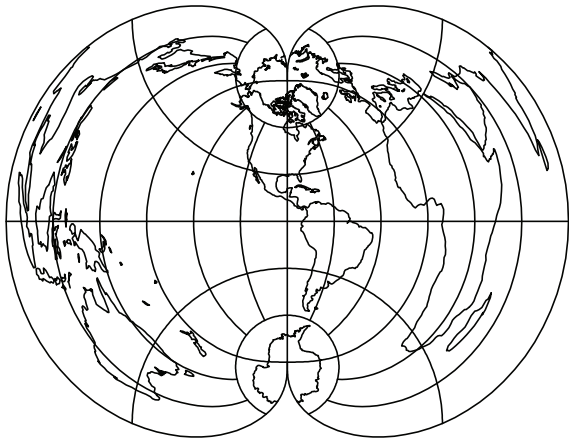


Figure 4.3: Poly-conic projection of the Earth (based on [49]).

4.3 Domain Analysis: Surroundings of Autonomous Ground Vehicles

The surroundings of AGVs depend obviously on the range of applications and vary from simple driver assistance functions up to autonomous driving. In the following, the use case for autonomous driving is analyzed.

First, the surroundings can be subdivided into private and public areas. The difference in classification is the restricted access to the latter, the location of the AGV's control, and legal aspects for operating such a vehicle. Second, the surroundings' elements can be classified into stationary and dynamic elements.

4.3.1 Traits of Private Areas

Private areas are clearly separated from public access. Furthermore, the area of operation for AGVs is either entirely known or also restricted from public access. Therefore, risks for life and health are restricted to the private area's personnel only who can be instructed properly to reduce the risk of accidents.

Moreover, the current state of all AGVs is known by a central control center at any time which might be a real person or an institution as well. Thus, this center is able to track and stop any AGV in dangerous situations. Harbor facilities or factories are main areas for the operation of automatically or fully autonomously operating vehicles [76, 149].

Furthermore, rules for operating these AGVs to minimize the risk of accidents are available by the carrier and restrict the absolute liability to the private area's personnel and

material. These rules can be supervised by government to ensure conformance to laws.

Another important property is the possibility to measure the performance of AGVs. Therefore, it is possible to modify the surroundings or vehicle's technology to improve their performance and reliability over time. Moreover, it is possible to upgrade the entire fleet or parts of the fleet if newer technology is available.

Finally, the surroundings in private areas consists of denumerable elements with a predefined behavior at any time. Thus, the environment can be well modeled using a formal specification.

4.3.2 Traits of Public Areas

Contrary to private areas are restrictions and unpredictable issues of public areas. The main and important difference is the access for everyone. This is the reason why the potential state space for an AGV is unbounded and thus difficult to predict and control.

Furthermore, controlling AGVs is either decentralized by the vehicles themselves or only centrally supervised by a control station. Moreover, the traffic consisting of vehicles controlled by human drivers and AGVs at the same time is hardly interpretable or predictable for computer algorithms. The main reason is that AGVs cannot make "eye-contact" with a human driver to obtain information about the driver's future intentions. Therefore, a description of the current state may be incomplete on one hand and may be unpredictable on the other hand. The complexity of this context cannot be completely modeled using a formal specification.

4.3.3 Surroundings' Elements

There are different elements in the surroundings of an AGV to be considered in a description. These elements subdivide into a ground, as well as stationary and dynamic elements with a corresponding visual representation in the real world. Furthermore, there are elements without a visual representation like the right-of-way rule at intersections. These elements are described in the following sections.

4.3.3.1 Ground

The surrounding's ground is an important element because AGVs are ground-based and change their position and rotation directly depending on the ground's shape. Furthermore, the ground may be used by a perception system to compute the vehicle's current position

in the real world. Furthermore, the ground's property influence the vehicle's movements due to its adherence. Some algorithms also use the ground's characteristics to detect the own road or lane [20, 99].

4.3.3.2 Stationary Elements

The main aspect of stationary elements is their immobility. Moreover, their absolute position in the real world is known or can be easily determined. Examples for stationary elements are houses, trees, or traffic signs. Stationary elements are not only necessary to detect for collision avoidance but may also be used for Simultaneously Localization And Mapping (*SLAM*) algorithms [42] to navigate an AGV through unknown areas where satellite based localization can not be used like in buildings or factories.

4.3.3.3 Dynamic Elements

Dynamic elements are all objects changing their position and rotation over time. These elements may interact with AGVs to cause a proper reaction. Examples for dynamic elements are other cars, pedestrians, or bicyclists. Besides, these objects are most complicated to detect since all today's sensors detect only contours or shapes with a certain quality. Furthermore, object classifiers to map contours or shapes base either on assumptions or rule sets. For example, a contour-based sensor which is pointing in a vehicle's driving direction would classify a large and moving contour in front of the own vehicle as another car or truck. Spots or smaller contours next to a lane could be a pedestrian. Furthermore, a currently non-moving object on the sidewalk could be classified as a stationary object like a tree until it starts moving. Today, human experience in a machine processable representation for improving this classification problem is missing.

4.3.3.4 Logical Elements

Logical elements describe either elements like roads, lanes, or speed limits which are relevant to the sensor- and actuator-based autonomous system's context. Furthermore, they describe relations between other logical elements and may be used to specify traffic regulations. This information is provided before or during the sensor- and actuator-based autonomous system's run-time and may change over time.

4.4 Modeling the Surroundings of Autonomous Ground Vehicles

Based on the aforementioned overview of the AGVs' surroundings, a DSL is developed in the following. Moreover, as demanded by the design criteria mentioned above, the DSL is split into one for modeling stationary and one for modeling dynamic elements which are described separately.

4.4.1 Modeling of Stationary Elements

In this section the DSL which is used to provide and exchange formally consistent data for the stationary system's context is described. The overview of this part of the AGV's surroundings is depicted in a UML class diagram in Figure 4.4.

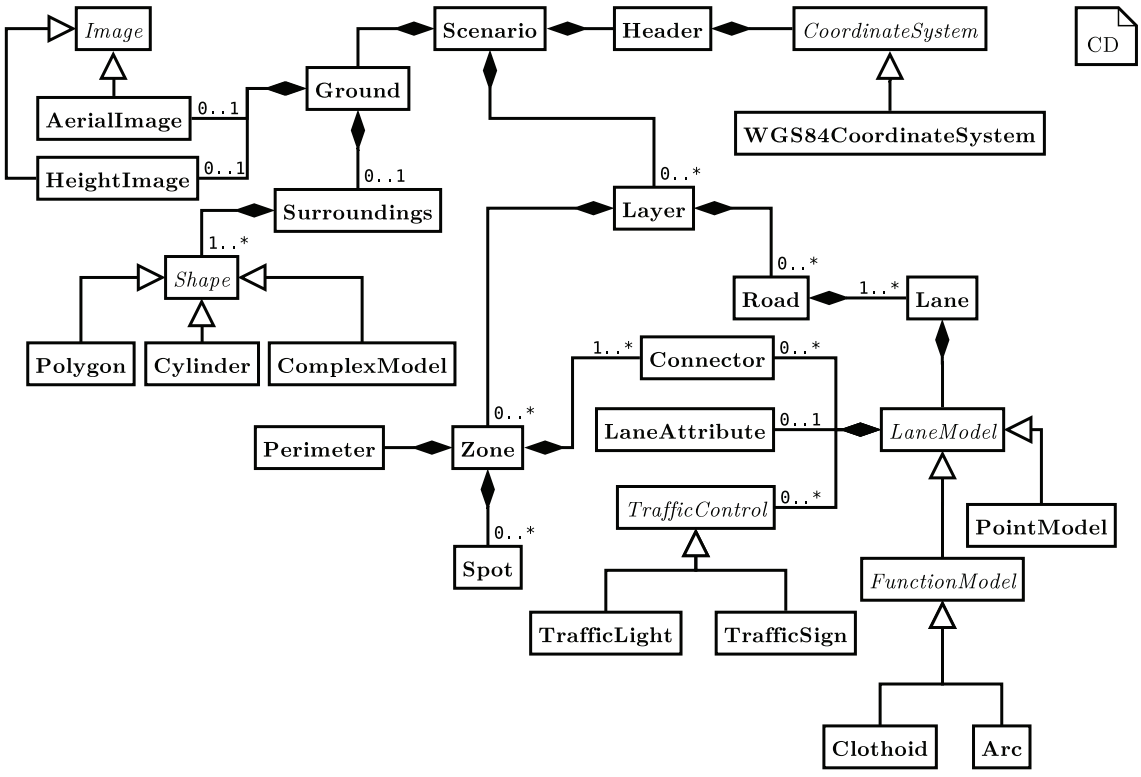


Figure 4.4: UML class diagram describing the system's context's stationary elements. For the sake of clarity only multiplicities other than 1 are denoted. The complete DSL which is derived from this UML class diagram including all attributes can be found in Section A.

In that figure, the stationary system's context is shown. The root element is named `Scenario` and consists of a header, a ground, and optional one or many layers. The

`Header` describes meta data about the concrete model itself like creation date, version, and its relation to the reality using a WGS84 referenced origin. The origin itself is the logical point $(0; 0; 0)$ creating a three-dimensional Cartesian coordinate system called World Coordinate System (*WCS*) for all other modeled elements. Using Cartesian coordinates, re-modeling of real environments using aerial images is easily possible. Aerial images as well as height maps describing the elevation of a region is provided by `Ground`.

The class `Ground` consists furthermore of a description of the stationary surroundings called `Surroundings`. This class contains at least one shape of the type `Polygon`, `Cylinder`, or `ComplexModel`. While the former both are evident, the latter type allows the reuse of existing complex 3D models created by popular 3D modeling programs like Blender [127] provided in the well-known Wavefront format. All these elements are positioned and rotated three-dimensionally in the model relative to the defined origin. Thus, modeling of realistic environments is possible.

On top of `Ground`, several `Layers` may be specified. A layer is a logical element allowing to define an order for `Roads` and `Zones` with a predefined height. Thus, a layer itself contains a height and a name. Moreover, using layers the definition of bridges is possible. Layers themselves contain `Roads` and optional `Zones`.

A road is the container element for one or more `Lanes`. A `Lane` is the concrete description of a drivable path. Thus, several attributes are necessary. The most important is `LaneModel`. This class describes the underlying mathematical model for the lane's shape. The easiest shape is the definition using absolute points related to the model's origin; this model is called `PointModel`. Besides, `Arc` can be used to define a circle with a predefined radius.

A more complex definition are clothoids as shown in Figure 4.5 [111]. Clothoids are the base for the design of roads for German highways for example. The advantage of clothoids is the linear change of curvatures for allowing a smooth driving dynamic due to continuous changes between curves along the path. However, the integrals themselves cannot be solved directly but must be approximated numerically. Their definition and a third order approximation which was used in [111] can be seen in Eqs. 4.9 and 4.10.

$$\begin{pmatrix} x \\ y \end{pmatrix} = \int_0^x \begin{pmatrix} \cos\left(t^2\right) \\ \sin\left(t^2\right) \end{pmatrix} dt. \tag{4.9}$$

$$c(x) = d_\kappa (x - x_0)^3 + \kappa (x - x_0)^2 + \tan(\phi)(x - x_0) + y_0 \tag{4.10}$$

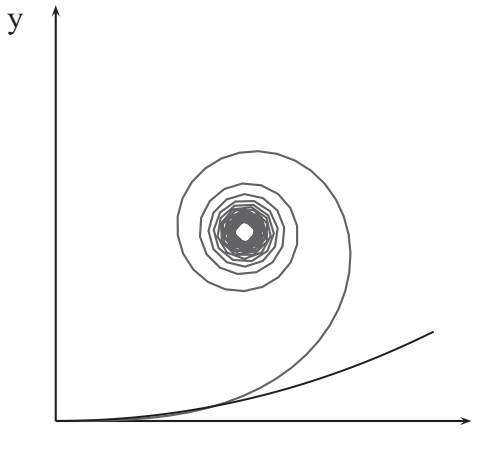


Figure 4.5: Fresnel integral together with an approximation using a 3rd order polynomial with $d_\kappa = 0.0215$ and $\kappa = 0.215$ which are estimated values to approximate the beginning of the Fresnel integral. For road segments which be shall modeled with a 3rd order polynomial instead of clothoids, a segment-wise approximation using several 3rd order polynomials with different lengths and coefficients are necessary. An algorithm for a segment-wise approximation of the Fresnel integrals is presented in [111].

Besides the lane's shape, additional information regarding its logical connections to either other lanes or zones are defined using `Connectors`. Furthermore, several `TrafficControls` can be assigned to a lane. Traffic controls are for example a traffic light, a stop sign, or speed limits. These controls are valid for the entire lane they are assigned with.

Additionally, `Lanes` can be connected to `Zones`. A zone defines a region without predefined drivable areas like lanes inside roads and is for example a parking-lot, defined by its `Perimeter`. Within this zone, several special `Spots` can be defined.

In a concrete instance of this model for stationary elements, all elements are named and identifiable using hierarchical identifiers starting at 1. For example, an addressable waypoint of a point-shaped lane that might be identified by 2.3.1.4 can be found on layer 2, road 3 and lane 1. Using this consistent nomenclature, navigatable routes can be defined easily by listing consecutive points.

4.4.2 Modeling of Dynamic Elements

After defining the stationary, immutable surroundings, dynamic elements can be added which is described in this section. Modeling of dynamic elements according to the model shown in Figure 4.6 can be used to extend the stationary surroundings with dynamic objects to define a *situation*. Therefore, a situation is always technically associated with exact one model of the stationary surroundings using its `SituationHeader`.

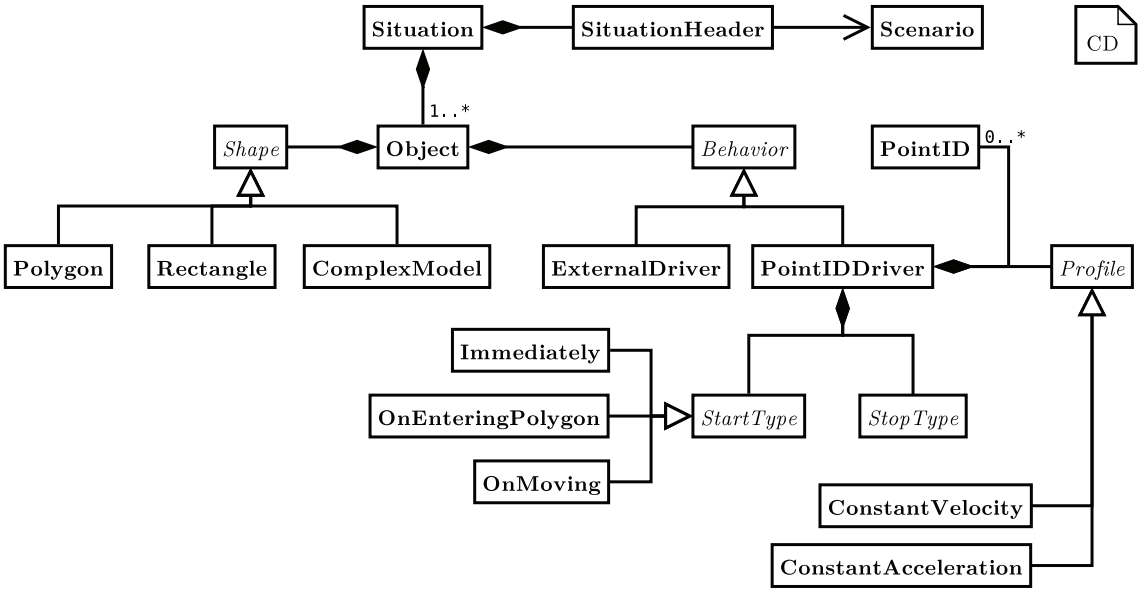


Figure 4.6: Class diagram describing the system's context's dynamic elements. For the sake of clarity only multiplicities other than 1 are denoted.

As shown in Figure 4.6, the dynamic system's context consists of at least one `Object`. Every object has an associated appearance, which is either based on polygons or modeled by using `ComplexModels`. Here, the latter are the same as already mentioned in Section 4.4.1. Furthermore, every object has an initial position and rotation, a name, and a unique identifier as well.

Besides its shape, a behavior must be assigned to an object to define its role in a concrete situation, which can be `ExternalDriver` or `PointIDDriver`. The former indicates that this dynamic object is controlled externally by humans or by a high-level trajectory planner. The latter behavior instead can be used by the simulator as outlined in the following.

The `PointIDDriver` indicates that the associated object is controlled by an algorithm to generate its position data during a simulation. An object with this behavior simply follows a predefined list of identifiable way-points from the stationary surroundings using an associated driving `Profile` like constant velocity or constant acceleration. Moreover, `StartType` and `StopType` for `PointIDDrivers` can be defined. The former describes the event for starting an object. This includes an immediate start at $t = 0$, a start when any other object starts moving to model situations at intersections for example, or a start, when any other object enters a specific polygon and thus triggering an event. The `StopType` defines the object's behavior, when it has reached its final way-point. This includes no further action or a restart of its associated route.

4.5 Scenario-based Modeling

For simplifying the creation and exchange of scenario data, the Compressed Scenario Data Format (*SCNX*) was designed to define *scenarios* which combine the stationary and dynamic elements as described above for modeling different traffic situations. The SCNX collects all necessary data describing completely a scenario and combining of several files into one single, compressed file. Furthermore, the Wavefront format for describing complex objects like houses, trees, or vehicles using three-dimensional models, which consists itself of several separated files for the shape, its material, and textures, was combined into one single Compressed Object Data Format (*OBJX*) file, which can be easily embedded into SCNX files. Using an SCNX file, consistent scenario-based modeling can be achieved.

After defining meta-models for surroundings' models, also known as abstract syntax and referred to as DSL in the following, its realization is discussed in the next sections. First, its realization for Java using *MontiCore* [74, 93] is presented. Then, the realization for C++ to be used for time-critical applications on ECUs is shown.

4.5.1 Scenario-based Modeling Using MontiCore

The framework MontiCore is developed at the Software Engineering Group at RWTH Aachen University and Technische Universität Braunschweig for supporting the agile design, evolution, and implementation of DSLs or even GPLs. MontiCore offers traditional grammar-based as well as today's meta-modeling concepts for defining the abstract and concrete syntax for a language in one single representation. Therefore, it provides a language similar to the Extended Backus-Naur Form (*EBNF*) which is used to generate a suitable lexer based on Another Tool for Language Recognition (*ANTLR*) [116] and a parser as well as classes for constructing an Abstract Syntax Graph (*ASG*) to be used for processing instances of a concrete grammar using modern object-oriented concepts like visitors [64]. MontiCore itself is realized using the programming language Java.

MontiCore was chosen to quickly implement and evolve conceptual elements of the modeling language. Furthermore, a graphical editor for the stationary and dynamic AGV's surroundings was developed [145] using the Eclipse Rich Client Platform [153] which also bases on Java. In Listing 4.1, an excerpt from the grammar for modeling stationary elements is shown. The complete grammar as well as the grammar for dynamic elements can be found in Section A.

```
. . .
```

```
Road = "Road" "RoadID" RoadID:Number ("RoadName" RoadName:↩
AlphaNum)? Lanes:Lane+ ";";

Lane = "Lane" "LaneID" LaneID:Number
        LaneModel:LaneModel ";";

LaneModel = LaneAttribute:LaneAttribute
            Connectors:Connector*
            TrafficControls:TrafficControl*
            (PointModel | FunctionModel);
...
PointModel = "PointModel" IDVertex2+ ";";

FunctionModel = "FunctionModel" (Clothoid | Arc) ";";

Clothoid = FunctionModel:"Clothoid"
           "dk" dk:Number
           "k" k:Number
           "Start" Start:IDVertex2
           "End" End:IDVertex2
           "RotZ" Rotation:Number;
...
```

Listing 4.1: Excerpt from MontiCore grammar for stationary surroundings.

The grammar shown in Listing 4.1 is abbreviated but shows some core elements for modeling the stationary surroundings. In line 2, the definition of a road is given. Besides its identifier and optional name, lanes are associated and defined in line 4. The mathematical basis for these lanes is essential and defined in production rule `LaneModel` in line 7 et seqq. Since clothoids are already exemplary introduced in Section 4.4.1, in line 16 et seqq. its approximation is defined.

From this grammar, nodes for the ASG, reflecting the grammar's structure and named according to the production rule, are generated together with a lexer and parser for processing instances of this language using Java. These classes are instantiated automatically during the parsing process and connected to a graph for further usage. Right after the parsing process, a validation visitor traverses the ASG for checking several semantic constraints like indices of the surroundings' elements must be in a valid interval, whether the image's dimensions are positive, or mapping from numerical constants to an enumeration class.

4.5.2 Scenario-based Modeling Using C++

After implementing the grammar for modeling the stationary and dynamic surroundings with MontiCore, its use for C++ is discussed in the following. Since MontiCore generates a lexer and parser only for Java, an alternate approach must be chosen to directly use the grammar with C++ for the development of embedded software for ECUs.

For example, Yet Another Compiler-Compiler (*YACC*) can be used to define the grammar in Backus-Naur Form (*BNF*) for automatically generating a parser for C. The parser processes tokens provided by a lexer which itself must be supplied either by the developer [88] or provided by the lexical analyzer (*LEX*) [97]. Furthermore, support for ASG classes is missing at all. Thus, its use in an object-oriented system is rather improper.

For using a language processing framework that is compatible with C++, *Spirit* was chosen [23]. As part of the Boost project, a peer-reviewed, platform-independent, and widely used collection of libraries for C++, the parser framework is very suitable.

Like MontiCore, Spirit generates a lexer and parser from an EBNF specification of the grammar. Hereby, the grammar itself is provided using C++ template concepts [165] and implemented as a regular class, and thus, the lexer and parser are generated completely at *compile-time* using the regular compiler without the need for any other tool. While Spirit itself relies heavily on template concepts, only modern C++ compilers like GNU's G++ 3.1, Microsoft Visual C++ 7.1, or Intel C++ 7.1 can be used to compile the grammar's input files. The main advantage of a compile-time generated lexer and parser is the automatically assured consistency between the grammar, the language processing framework, and the source code using instances of the given grammar by avoiding additional language processing steps in the software build process.

```
...
ROAD = str_p("ROAD") >> NEWLINE >>
       str_p("ROADID") >> TAB >> NUMBER >> NEWLINE >>
       !(str_p("ROADNAME") >> TAB >> ALPHANUM >> NEWLINE) >>
       +(LANE >> NEWLINE) >> str_p("ENDROAD");

LANE = str_p("LANE") >> NEWLINE >>
       str_p("LANEID") >> TAB >> NUMBER >> NEWLINE >>
       LANEMODEL >> NEWLINE >> str_p("ENDLANE");

LANEMODEL = LANEATTRIBUTE >>
            *(CONNECTOR) >>
            *(TRAFFICCONTROL) >>
```

```
            (POINTMODEL | FUNCTIONMODEL);
...
POINTMODEL = str_p("POINTMODEL") >> NEWLINE >>
             +(IDVERTEX2 >> NEWLINE) >> str_p("ENDPOINTMODEL
             ");

FUNCTIONMODEL = str_p("FUNCTIONMODEL") >> NEWLINE >>
                (CLOTHOID | ARC) >> NEWLINE >> str_p("
                ENDFUNCTIONMODEL");
...
CLOTHOID = TYPEFUNCTIONMODEL >> NEWLINE >>
           str_p("DK") >> TAB >> NUMBER >> NEWLINE >>
           str_p("K") >> TAB >> NUMBER >> NEWLINE >>
           str_p("START") >> NEWLINE >> IDVERTEX2 >> NEWLINE
           >>
           str_p("END") >> NEWLINE >> IDVERTEX2 >> NEWLINE >>
           str_p("ROTZ") >> TAB >> NUMBER;
...
rule<ScannerT, parser_context<>, parser_tag<
SCNGrammarTokenIdentifier::ROAD_ID> > ROAD;
rule<ScannerT, parser_context<>, parser_tag<
SCNGrammarTokenIdentifier::LANE_ID> > LANE;
rule<ScannerT, parser_context<>, parser_tag<
SCNGrammarTokenIdentifier::LANEMODEL_ID> > LANEMODEL;
rule<ScannerT, parser_context<>, parser_tag<
SCNGrammarTokenIdentifier::POINTMODEL_ID> > POINTMODEL;
rule<ScannerT, parser_context<>, parser_tag<
SCNGrammarTokenIdentifier::FUNCTIONMODEL_ID> > FUNCTIONMODEL;
rule<ScannerT, parser_context<>, parser_tag<
SCNGrammarTokenIdentifier::TYPEFUNCTIONMODEL_ID> >
TYPEFUNCTIONMODEL;
rule<ScannerT, parser_context<>, parser_tag<
SCNGrammarTokenIdentifier::CLOTHOID_ID> > CLOTHOID;
...
rule<ScannerT, parser_context<>, parser_tag<
SCNGrammarTokenIdentifier::START_ID> > const& start() const {
    return START;
}
```

Listing 4.2: Excerpt from Spirit grammar for stationary surroundings.

Comparable to Listing 4.1, a similar excerpt from the grammar for modeling stationary elements realized using Spirit is shown in Listing 4.2. In line 2 et seqq., the definition of a road is provided which is split into several non-terminals. While MontiCore uses ANTLR as underlying lexer for processing the grammar's tokens, keywords like `ROAD` are parsed using so-called *lexeme parsers* which is indicated by `str_p`. Furthermore, several tokens representing an input stream are divided by `>>`.

Another difference is the definition of quantifiers like `*` for zero to many occurrences or `+` for one to many occurrences. In MontiCore, quantifiers are defined right behind the terminal or non-terminal. In Spirit, quantifiers are defined in front of the regarding terminal or non-terminal. Furthermore, the quantifier for none or one occurrence is defined as `!` in Spirit as contrary to `?` in MontiCore. Moreover, in Spirit every non-terminal must be marked by the template `rule` as shown in line 28 for example.

The following lines are analog to the grammar implemented using MontiCore. The grammar itself is implemented as a regular class in C++, every non-terminal and terminal is represented as a class' attribute in line 29 et seqq. Finally, in line 37 et seqq., a method called `start()` defines the start production rule from the grammar.

For parsing instances from a grammar, the language processing implementation presented here supports two different concepts. First, an observer-based concept was implemented calling registered listeners whenever a successful match for a token could be applied to an input stream of tokens. Furthermore, the internal Abstract Syntax Tree (*AST*) concept from Spirit was enhanced for its integration into $\mathcal{H}$esperia to create an ASG which is as easy to use as the one generated by MontiCore for processing a grammar's instance in an intuitional manner.

A class diagram showing the inheritance for the C++ implementation of the grammar for modeling the stationary surroundings is depicted by Figure 4.7. The super-class `Grammar` is implemented using the *facade design pattern* to encapsulate the concrete handling of grammar's instances which allows to support different versions of the Spirit framework which are not fully backward compatible.

Furthermore, this class is an *observer* implementing the aforementioned first concept for calling registered listeners about successfully matched tokens using the interface *ParserTokenListener*. This interface provides the method `void nextToken(const ParserToken&);` for notifying the listener about a successfully matched `ParserToken` which contains the value of the matched token as well as a caller supplied data field called `ParserTokenExtendedData`. For example, this field can be used to pass further information about the successfully matched token to the application like a numerical constant identifying the token itself.

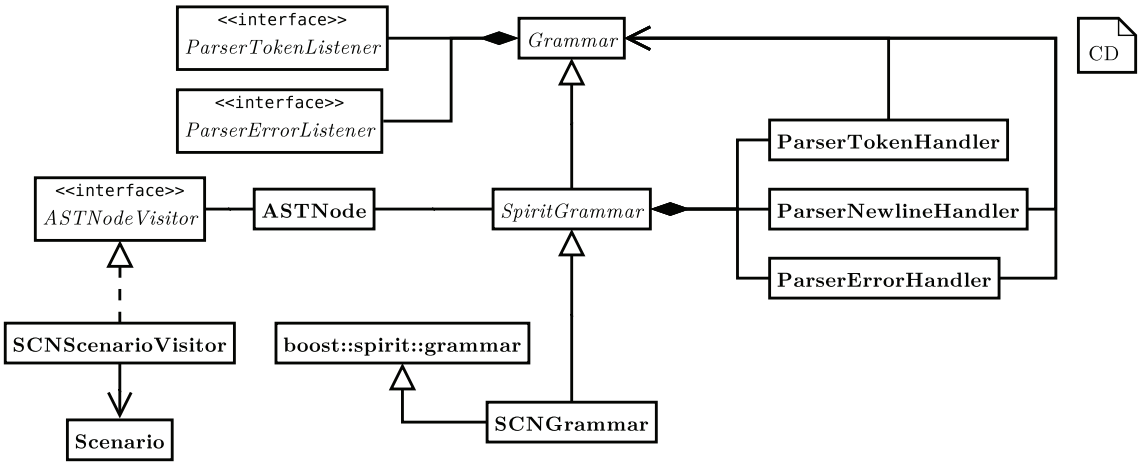


Figure 4.7: Class diagram describing the language processing for the stationary surroundings using Spirit. For the sake of clarity, all methods and attributes are omitted. An implementation-independent interface is realized in the abstract class `Grammar`. This class provides an observer for successfully parsed token from the grammar which calls a user-supplied listener; analogously realized is an observer which reports parsing errors. These observers are used to construct an AST from a given instance according to the grammar's structure. This AST can be easily traversed by user-supplied visitors to query information from a given instance or to transform the data structure.

Using `ParserErrorExtendedData`, a similar implementation is provided for error handling. The interface *ParserErrorListener* provides the method `void errorToken(ParserError&);` for notifying the caller about a failed match using the class `ParserError`. This class provides, besides the textual context in which the match failed, information about the line to ease finding the erroneous input. Like `ParserToken`, additional information supplied by the caller can be associated with `ParserError`. Internally, the information about line numbers is realized using a specialized listener `ParserNewlineHandler` simply counting successfully matched `NEWLINE` tokens.

For generating an ASG similar to the one generated by MontiCore for providing an interface to access attributes and associated elements from the grammar in an intuitional manner, Spirit provides a specialized parser called `parse_ast` to create an AST. The access to keywords from the grammar as well as values like the name of a defined road is realized using an iterator concept. In Spirit, an iterator traverses the internally created AST from the input grammar. Therefore, the grammar must contain information about the structure of the AST to be built namely root nodes or leaves. A root node is marked as `root_node_d`, while leaves are simply marked as `leaf_node_d`. Whenever the parser successfully matches tokens from the input stream which are enclosed by

`root_node_d`, the newly created node becomes the new root node of the current AST created from the previously successfully matched input.

However, for preserving the readability of the grammar, another concept was used utilizing some predicates from the iterator when using the `leaf_node_d` directive. The predicates are listed in the following:

- *Keywords.* Keywords from the grammar always have the identifier "0" and no further children.
- *Values.* Values from the grammar like a `Road`'s name have an identifier not equal to "0" and no further children.
- *Hierarchical elements.* Elements mapping the grammar's structure have an identifier not equal to "0" and children as well.

Using these predicates, an intermediate generic AST for generating the desired hierarchical key/value data structure is constructed automatically after reading a DSL's instance using the recursive descent parser as shown in Listing 4.3. The intermediate AST reduces the dependency to Spirit by using only the lexer and parser from the Spirit framework and allowing the concrete mapping to the final data structure to be independent from the underlying language processing framework.

```
...
void generateASG(const iter_t &it, ASGNode *parent) {
ASTNode *child = NULL; string key;
  for(unsigned int j = 0; j < it->children.size(); j++)
  {
    string data((it->children.begin() + j)->value.begin(), (↩
    it->children.begin() + j)->value.end());
    if ( (data != "") &&
     ((it->children.begin()+j)->value.id().to_long() == 0) &&
     ((it->children.begin()+j)->children.size() == 0) )
    { // Keyword found.
      child = new ASTNode(parent);
      child->setKey(key = data);
      parent->addChild(child);
    } else if ( (data != "") &&
      ((it->children.begin()+j)->value.id().to_long() > 0) &&
      ((it->children.begin()+j)->children.size() == 0) )
    { // Value found.
      if ( (child == NULL) || (child->getValue<string>() != "↩
      ") ) {
```

```
        child = new ASTNode(parent);
        parent->addChild(child);
      }
      child->setKey(key);
      child->setValue(data);
    } else if ( ((it->children.begin()+j)->value.id().to_long↩
    () > 0) &&
       ((it->children.begin() + j)->children.size() > 0) )
    { // Hierarchical element found.
      ASTNode *multipleChildren = new ASTNode(parent);
      generateAST(it->children.begin()+j, multipleChildren, ↩
      depth);
      multipleChildren->setKey(key);
      parent->addChild(multipleChildren);
    }
  }
}
...
```

Listing 4.3: Generating an intermediate AST using pre-processed data from Spirit.

Using the tree generated by this code, a visitor traverses finally this tree mapping the hierarchical key/value pairs to an ASG similar to the one generated by MontiCore. Hereby, all values are transformed to the necessary primitive data types like `double` or `unsigned int`. The same concept was analogously applied for the DSL which represents the surroundings' dynamic elements.

To summarize the development of a DSL and the processing of its instances, the lexer and parser framework MontiCore is very applicable to simplify the rapid and agile development of languages. However, due to a missing native support for C++, its integration would not be seamless and cause further processing steps during the software construction process. Therefore, an alternative concept which directly bases on C++ was necessary and finally chosen by the framework Spirit which was adapted to be more user-friendly and thus less error-prone.

5 The Framework $\mathcal{H}$esperia

In the following, the framework $\mathcal{H}$esperia[1] is presented. $\mathcal{H}$esperia is a framework for processing continuous input streams written entirely and solely in C++ [17]. The DSL as mentioned in Section 4.4 for modeling the system's context, a simulation component which is described separately in Chapter 6, and a non-reactive visualization component also described separately in Chapter 7 are core elements of this framework. In the following, general considerations and design drivers are discussed first. Next, its software architecture is outlined and the core libraries as well as selected applications are described in detail.

5.1 General Considerations and Design Drivers

In this section, the major design drivers for the framework $\mathcal{H}$esperia are listed. Mainly, they are based on [64, 123].

- *Usability.* The main focus during the design of the framework $\mathcal{H}$esperia was on usability. Usability includes both application development using $\mathcal{H}$esperia and development in $\mathcal{H}$esperia itself. While the former is related to the design of interfaces exported to the application developer, the latter applies to the design of all internal structures of $\mathcal{H}$esperia. Therefore, $\mathcal{H}$esperia was designed using an object-oriented programming language with intense use of mature design patterns where applicable [129, 130].
- *Reliability and robustness.* Another important aspect is reliability regarding the methods exported by interfaces to any caller as well as internal algorithms for data processing. Whenever an exported method from the Application Programming Interface (*API*) is invoked by a caller, its semantic must be evident and consistent.

[1] The name "$\mathcal{H}$esperia" is deduced from a town in California, where the team CarOLO was accommodated during the 2007 DARPA Urban Challenge [122]. Concepts behind the software framework which was used in the CarOLO project were extended and significantly improved which led to a complete new implementation written from scratch: The software framework $\mathcal{H}$esperia.

Furthermore, any exported method must either complete the demanded computation silently or fail avoiding any side-effects at the earliest point possible with a reasonable return code or an exception.

- *Performance.* Due to its application for ECUs on real hardware and in real system's surroundings which may require real-time data processing, performance in data processing and for inter-node communication is an important design consideration for $\mathcal{H}$esperia. Thus, efficient memory management for incoming data as well as concurrent handling of data structures in the processing chain before passing them to user applications are necessary.
- *Platform independence.* For allowing the use of $\mathcal{H}$esperia on different platforms, independence from a specific Operating System (*OS*) or hardware platform is also a main interest. This includes endianess as well as 32-bit and 64-bit systems. Furthermore, an abstraction from specific functions provided by an OS must be chosen, especially for concurrency including threads, mutexes, and conditions, as well as data input/output operations and the overall valid system time which is important for carrying out system simulations as outlined in greater detail in Chapter 6.
- *Third party libraries independence.* Like the independence from a concrete platform, applications realized with the framework $\mathcal{H}$esperia should not depend on specific libraries where possible. This design criterion is important because the libraries for computationally intense tasks like image or matrix processing using a hybrid approach based on a Central Processing Unit (*CPU*) combined with a Graphical Processing Unit (*GPU*) may change caused by changing design decisions due to enhancements and bug-fixes. For avoiding a preliminary decision for a specific library, only interfaces are specified to be fulfilled at least by any library which could be chosen for a specific task.
- *Evolvability and reusability.* To support future applications or other system contexts with new or modified requirements, the software framework should be applied easily. Furthermore, missing parts should be added easily as well without accidentally breaking existing concepts or implementations.

In the following, the software architecture of the framework $\mathcal{H}$esperia is described.

5.2 Software Architecture

In this section, the software architecture of $\mathcal{H}$esperia is outlined. Therefore, a high-level point of view on all components is introduced at first, while, later on, the two main li-

braries are described explicitly in greater detail in a package diagram.

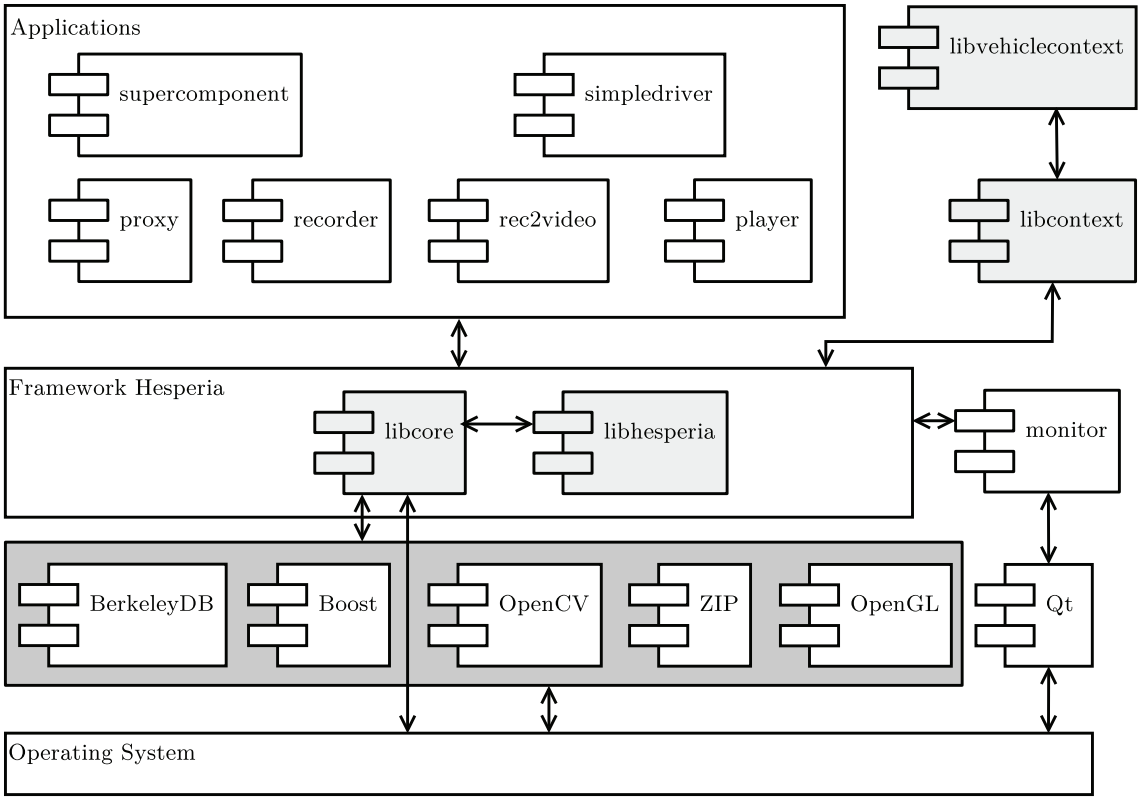


Figure 5.1: Package structure of the framework $\mathcal{H}$esperia. The framework consists of two major libraries: `libcore` and `libhesperia`. The former library encapsulates the interfaces for a specific operating system by providing elaborated programming concepts for I/O access or threading. Furthermore, this library wraps libraries from third parties and provides interfaces instead to higher layers. Thus, a third party library can be easily exchanged if necessary. The latter library, `libhesperia` provides concepts which allow a simplified development for distributed data processing applications. Therefore, this library provides classes which transparently realize data exchange; moreover, this library contains the DSL which was specified in Section 4.4.

In Figure 5.1, an overview of all components in a system using the framework $\mathcal{H}$esperia are depicted. On the lowermost layer, the OS is shown. The framework can be used both on a Portable Operating System Interface for Unix (*POSIX*)-compliant platform [1] or on Microsoft Windows platforms. For getting real-time capabilities, $\mathcal{H}$esperia relies on the operating system's supplied process handling. Therefore, the `rt-preempt` patch applied to the Linux kernel 2.6.27-3-rt [107] provided in the Linux distribution Ubuntu 8.10 was used.

The next layer lists all libraries used by $\mathcal{H}$esperia. Besides the aforementioned POSIX-compliance, $\mathcal{H}$esperia can be used on Microsoft Windows platforms using the Boost libraries for C++; these libraries can be used on POSIX-compliant platforms as well. For

example, a transaction- and thread-safe key/value data-store is provided using the BerkeleyDB. For processing images and matrices, currently the OpenCV library is used [25]. For accessing compressed scenario data files as described in Section 4.5, a portable zip library is included. Finally, for processing data for visualization as well as simulation, libraries from OpenGL are used.

On top of the libraries lays the framework $\mathcal{H}$esperia. The framework consists of two core components, namely `libcore` and `libhesperia`. While the former library encapsulates access to the OS and all aforementioned libraries, the latter one provides *concept*s for all higher layers. Both libraries are described in greater detail in the next sections. Using these libraries, several applications are provided to support the development of software for sensor- and actuator-based autonomous systems and especially AGVs which were also realized with the software framework $\mathcal{H}$esperia. These applications are outlined in the following sections.

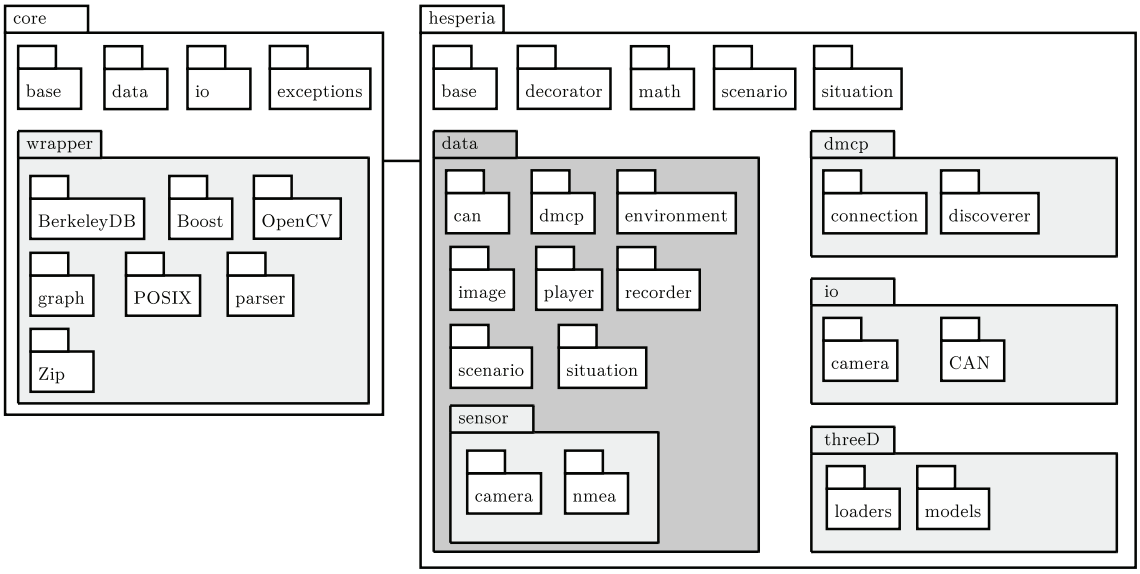


Figure 5.2: Packages of the framework $\mathcal{H}$esperia: The left hand side is realized in `libcore` which encapsulates the access to the operating system and to third party libraries as already mentioned. The right hand side is realized in `libhesperia`. Besides high-level concepts for transparent communication for example, basic data structures which support the development of sensors-based applications which operate in the $\mathbb{R}^3$ are provided. Furthermore, device-independent visualization concepts which are outlined in Section 5.4.5 are integrated.

As shown in Figure 5.2, both libraries consist of several packages. `libcore` provides rudimentary services in its packages `base`, `data`, `io`, and `exceptions`, as well as all interfaces to necessary libraries in package `wrapper`. The library `libhesperia` however uses `libcore` for both to integrate the DSL for the stationary and dynamic

context description and to provide concepts to higher layers. Furthermore, all serializable and therefore exchangeable data structures between all components are defined in this library.

5.3 Encapsulating the Operating System and Required Third Party Libraries: `libcore`

The library `libcore` consists of several packages which are described in the following. First, the package `wrapper` is described for ensuring library independence. Next, rudimentary concepts using the interfaces provided by the package `wrapper` realized in the packages `base`, `data`, and `io` building the conceptual base for `libhesperia` are described.

5.3.1 Package `wrapper`

The main goal of package `wrapper` is to encapsulate any library to be used in higher levels. On the example of `CompressionFactory` for providing access to zip compressed data, this package is described.

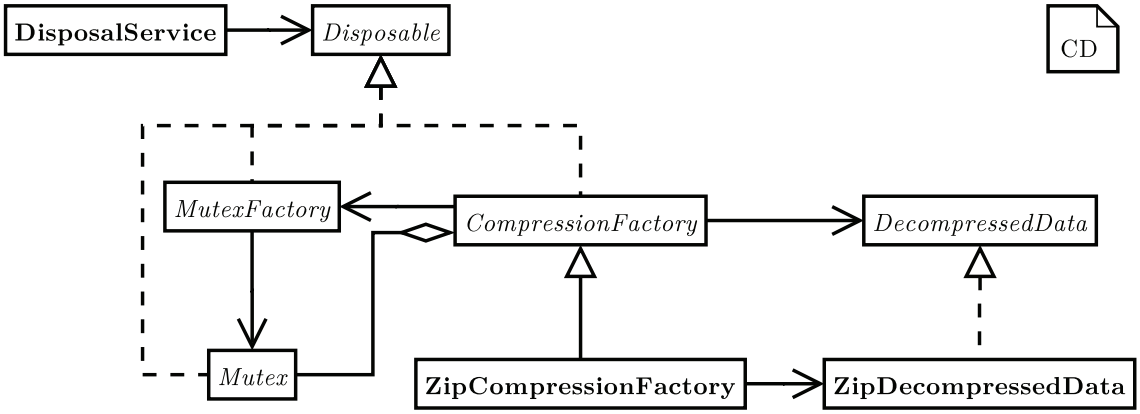


Figure 5.3: CompressionFactory for providing access to compressed data.

As shown in Figure 5.3, `CompressionFactory` is realized as an *abstract, singleton factory* exporting one method with the following signature `DecompressedData* getContents(std::istream&);`. This method reads as many bytes as available from the given input stream using the STL allowing input from files, memory, or any input source compliant to STL input streams like wrapped network connections. It creates

the meta data structure `DecompressedData` describing the contents of a successfully decompressed input stream. This data structure provides a list of available entries as well as a method for getting an STL input stream for every entry of the decompressed zip archive. For realizing a thread-safe singleton implementation, the access to the method `CompressionFactory& getInstance();` must be mutually excluded using a `Mutex`. However, a mutex itself is also library-dependent wrapped. Therefore, the abstract factory for creating the appropriate mutexes must be used.

From this *abstract factory*, the concrete factory `ZipCompressionFactory` providing `ZipDecompressedData` is derived for wrapping the zip library. `ZipDecompressedData` actually decompresses the zip archive by reading the given input stream using the methods provided by the wrapped zip library. If the given input stream could not be used for decompressing the data, the list of available entries is simply empty. Internally, all successfully decompressed entries are stored in memory using `stringstreams` to provide the standard interface based on `std::iostream` for further data processing.

Both factories, `MutexFactoy` as well as `CompressionFactory` or rather their library-dependent concrete factories also implement the abstract interface `Disposable`. This interface does not export any further method but simply declares all deriving classes to be convertible to this type. Using this interface, a periodically and at program's exit running `DisposalService` removes any instances from the type `Disposable` when they are no longer needed to release previously acquired memory. Using `CompressionFactory` and `DecompressedData`, applications on higher levels do not need to care about a specific library for decompressing data. Instead, they simply use an interface asserting the availability for the required functionality.

In the following, all factories provided by the package `wrapper` are described. The actually selected libraries for wrapping are specified in a header file using a system-wide consistent enumeration scheme.

- *CompressionFactory.* This factory was already described.
- *ConcurrencyFactory.* This factory creates a thread by invoking the method `void run();` from the interface `Runnable`. Furthermore, a statically available sleep method for suspending the execution for a given amount of time is exported. This factory wraps the library Boost and regular POSIX calls as well.
- *ConditionFactory.* This factory creates a `Condition` for suspending the concurrent execution of several threads until the condition for a waiting thread is met. This factory wraps the library Boost and regular POSIX calls as well.

- *ImageFactory.* Using this factory, images are created either by reading an input stream, by creating an empty image specifying the format and the image's dimensions, or by creating the necessary meta information for already existing images in memory. This factory wraps the library OpenCV.
- *KeyValueDatabaseFactory.* This factory creates either a wrapped transaction- and thread-safe BerkeleyDB or simply a wrapped `std::map` for storing and retrieving key/value-pairs.
- *MatrixFactory.* This factory creates an empty NxM matrix by providing the data structure `Matrix`. This data structure defines besides the element-wise access to the matrix' contents the addition, multiplication, and transpose operations. Furthermore, a template method is provided to get access to the memory representing the raw matrix for using operators which are missing in the exported interface. Obviously, this exported method violates the demand for library independence, but any application relying on this method can safely query the system if the necessary library is wrapped and throw an exception otherwise. This factory wraps the library OpenCV as well.
- *SharedMemoryFactory.* Using this factory, a memory segment between independent processes can be created and shared using `SharedMemory` for achieving fast inter-process communication. On construction, additional memory at the beginning of the memory segment is used to create a semaphore to ensure mutual exclusion for concurrent processes. This factory wraps also the library Boost and regular POSIX calls.
- *TCPFactory.* This factory can be used to create either a connection to an existing server by returning a `TCPConnection` or to setup a socket accepting connections using a `TCPAcceptor` for a specific port. Both objects can be used to transfer data bidirectionally. While the former instance already encapsulates an established connection which is ready to use, the latter object implements an observer for incoming connections. Using this object, the caller must register a `TCPAcceptorListener` for getting notified about new connections encapsulated in an instance of `TCPConnection`. For sending data, simply the exported method `void send(const std::string&);` can be used. For receiving data, a `StringListener` must be registered at a concrete `TCPConnection` as explained in the following. This factory wraps the library Boost and regular POSIX calls as well.
- *TimeFactory.* This factory creates a `Time` instance containing the current time. This factory is necessary for simulation purposes and wraps also the library Boost

and regular POSIX calls.

- *UDPFactory.* Like `TCPFactory`, this factory creates either a sender for data transfers or a receiver using User Datagram Protocol (*UDP*). Depending on the address supplied for the receiver, either a regular UDP socket is created or the created UDP socket is joined to a UDP multi-cast group. This feature is used in `libhesperia` for realizing so-called conferences. The sender and receiver objects also use `std::string` for sending and a `StringListener` for receiving data. This factory wraps the library Boost and regular POSIX calls as well.

All the factories listed above use the same concept for creating concrete data structures as already described on the example for `CompressionFactory`. Furthermore, all factories implement the interface `Disposable` as well.

The smallest datum for sending and receiving data is `std::string`. Thus, `libcore` provides a `StringObserver` exporting the method `void setStringListener(StringListener*);` for registering or unregistering a concrete instance implementing the interface `StringListener`. Using this interface exporting the method `void nextString(const std::string&);` a component can receive new data for further processing. Combining both concepts, a `StringPipeline` decouples the receiver and the consumer of newly received data. This pipeline is used transparently for any application in the UDP receiver to separate the thread responsible for handling the library-dependent receiving method from the thread responsible for further processing the received data in higher layers. Therefore, a simple thread-safe FIFO queue using a `Condition` to notify the waiting consumer was implemented.

Another wrapper providing no factory is the package `parser` producing a lexer and parser for the DSL which is used to model the system's context as outlined in Section 4.4. Due to the special handling of a compile-time grammar with the production of an intermediate AST as described in Section 4.5.2, the grammar's concrete instantiation is directly implemented in `libhesperia`. As already described before, the generic handling of tokens produced by the parser is implemented in `libcore`. However, the visitor for generating the data structure itself is implemented in `libhesperia`.

5.3.2 Basic Concepts

In the following, selected basic concepts provided by `libcore` are described. Some of them are elaborated in `libhesperia`.

5.3.2.1 Application Template

Besides wrapping libraries, `libcore` provides several basic concepts being extended in `libhesperia`. As described in greater detail in Section 5.4, a running application in the framework $\mathcal{H}$esperia belongs to exactly one group called *conference*. Several applications from the same type are distinguished using identifier. Furthermore, every application can be configured to run at a fixed frequency. All these parameters can be passed to an application using command-line arguments. These arguments are processed by the application skeleton class `AbstractCIDModule` which is part of package `base`. Furthermore, the application template provides a method for querying its current running state. Using this method, the application can be terminated safely by registering a signal handler to catch signals by the OS to the application like `SIGTERM` for POSIX-compliant systems. The application templates are the main entry for system simulations as described in Chapter 6.

5.3.2.2 Reliable Concurrency

Modern operating systems offer the possibility to execute parts of an application in parallel. For using this concurrency in higher layers and in user-contributed applications, the concept `Service` is provided to encapsulate the entire management of threads. Any class that needs to be executed concurrently simply derives from `Service` overriding the method `void run();`. This concept is enhanced for real-time computing by the class `RealtimeService`. Any service of this type simply derives from this class overriding `void nextTimeSlice();`. Internally, `RealtimeService` is a wrapper class which encapsulates the technical implementation for a specific operating system. Moreover, any application which uses real-time services must be executed using a privileged user account to allow the correct setup. Otherwise, an exception is thrown.

5.3.2.3 Thread-Safe Data Storage

Beyond the basic application template and concurrency facilities, basic storage concepts for data structures are provided, namely FIFO queues, Last In First Out (*LIFO*) stacks, and simple key/value data-stores. All these data-stores are both thread-safe and capable of using conditions to notify changes.

The data structure meant to be used with these data-stores is `Container`. This class is an envelope data structure around a pair consisting of a constant and consistent numerical identifier allowing type definition and an arbitrary object implementing the interface

`SerializableData`. A `Container` itself is serializable to `std::ostream` and deserializable from `std::istream`. Thus, it can be transferred using concepts provided by the STL. For tracking a datum, the sent and received timestamps are recorded for a transported `Container`.

5.3.2.4 Query-able Data Serialization

The actual serialization of instances implementing the interface `SerializableData` is inspired by [125]. However, the main disadvantage presented in that work is the deserialization's dependency on the serialization order which is caused by separate methods for serializing and deserializing the data. In the Boost library, this problem is avoided by using the non-standard serialization and deserialization operator `&`. However, whenever a data structure changes due to further development over time, older versions of the framework might get incompatible.

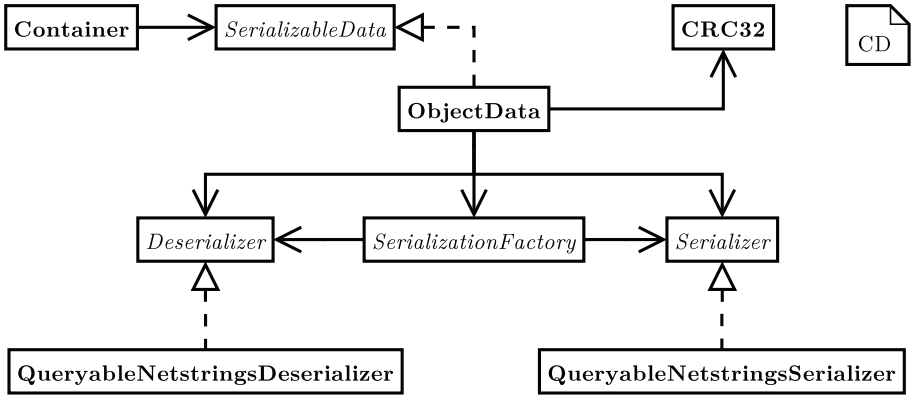


Figure 5.4: Template-based query-able serialization: The data to be serialized is realized by `ObjectData`. This class derives from the interface `SerializableData` which itself provides serialization and deserialization methods which are called by the envelope data structure `Container`. These methods are realized using the supporting classes `Serializer` and `Deserializer` which encapsulate the handling of hardware-dependent endianess for example.

For avoiding both problems, a so-called *template-based query-able serialization* was developed as shown in Figure 5.4. The main idea behind this concept is the storage of a datum to be serialized together with a per attribute identifier. When an object `ObjectData` should be serialized, it uses the `SerializationFactory` to get an appropriate `Serializer`. This instance actually serializes the data into a portable format regarding the platform-dependent endianess using type-dependent `void write(const uint32&, T);` methods. The first parameter is an identifier for every object's attribute.

For avoiding to supply this parameter manually by the caller, it should be computed already at compile time since the object's attributes and their order do not change at runtime.

```
...
#define STRINGLITERAL1(a) CharList<a, NullType>
#define STRINGLITERAL2(a, b) CharList<a, CharList<b, NullType> >
#define STRINGLITERAL3(a, b, c) CharList<a, CharList<b, CharList<c, NullType> > >
...
namespace core {
  namespace base {

    const uint32_t CRC32POLYNOMIAL = 0x04C11DB7;

    class NullType {
      public:
        enum { value = -1 };
        enum { hasNext = false };
        typedef NullType tail;
    };

    template <char x, typename xs>
    class CharList {
      public:
        enum { value = x };
        typedef xs tail;
    };

    template <char x>
    class CharList<x, NullType> {
      public:
        enum { value = x };
        typedef NullType tail;
    };

    template<char c, uint32_t result >
    class CRC32_COMPUTING {
      public:
```

```
35          enum { RESULT = result ^ (c ^ CRC32POLYNOMIAL) };
        };

        template <char c, int32_t res, typename T>
        class CRC32_RECURSIVE {
40        public:
            enum { RES = CRC32_COMPUTING<c, res>::RESULT };
            enum { RESULT = RES + CRC32_RECURSIVE<T::value, RES, ↩
            typename T::tail>::RESULT };
        };

45      template <char c, int32_t res>
        class CRC32_RECURSIVE<c, res, NullType> {
          public:
            enum { RESULT = CRC32_COMPUTING<c, res>::RESULT };
        };
50
        template <typename T>
        class CRC32 {
          public:
            enum { RESULT = CRC32_RECURSIVE<T::value, 0, typename↩
             T::tail>::RESULT };
55      };
    }
    ...
```

Listing 5.1: Compile-time computation of identifiers for serialization.

This computation is shown in Listing 5.1. The computation itself is invoked using the serialization `s.write(CRC32< STRINGLITERAL3('v', 'a', 'l') >::RESULT, m_value);`. The first argument to the method uses the compile-time computation by substituting `STRINGLITERAL3('v', 'a', 'l')` by the nested character list `CharList<'v', CharList<'a', CharList<'l', NullType> > >` in the pre-processor stage. The structure of this nested character list is defined by the classes in lines 11 et seqq., 19 et seqq., and 26 et seqq. The macros defined at the beginning in line 2 et seqq. are only for convenient use of these structuring classes. The resulting nested templates are passed as a template parameter to the class `CRC32` in line 52 et seqq. The result of the computation performed at compile-time is stored in the class' enum as the constant value `CRC32::RESULT`.

The computation itself is delegated to the class in line 39 et seqq. by separating the first element of the nested character list as first template parameter, the initial result of 0 and the nested character list without the head element. In that class, the first enum computes the Cyclic Redundancy Check (*CRC*) sum using the template class in line 33 et seqq. For computing this sum, the CRC-32 polynomial is used due to its sensitivity regarding modifications on the input data and thus defined in line 9. The result is added to the result from a recursive call to the same template class, using the first element of the tail as first parameter, the currently computed result, and the tail without its first element. The computation ends if the `NullType` is reached and therefore, the template class in line 46 et seqq. is applied to return the CRC-32 sum for the last character. Using these template classes, the computation of identifiers at compile-time is possible to allow the serialization of pairs consisting of human readable identifiers and values.

For deserialization, the object `ObjectData` gets the `Deserializer` by querying the `SerializationFactory` as well. At construction of the `Deserializer`, the input stream is parsed to build a simple hash-map containing the previously identifier/value pairs. Every time, `ObjectData` wants to deserialize one of its attributes, it simply queries the `Deserializer` using the identifier of the attribute computed already at compile-time.

```
...
ostream& AClass::operator<<(ostream &out) const {
    SerializationFactory sf;
    Serializer &s = sf.getSerializer(out);

    s.write(CRC32 < HESPERIA_CORE_STRINGLITERAL5('d', 'a', 't↩
    ', 'a','1') >::RESULT,
                    getMyFirstData());

    s.write(CRC32 < HESPERIA_CORE_STRINGLITERAL5('d', 'a', 't↩
    ', 'a','2') >::RESULT,
                    getMySecondData());
    return out;
}

istream& AClass::operator>>(istream &in) {
    SerializationFactory sf;
    Deserializer &d = sf.getDeserializer(in);

    uint32_t mySecondData = 0;
```

```
    d.read(CRC32 < HESPERIA_CORE_STRINGLITERAL5('d', 'a', 't'↩
    , 'a', '2') >::RESULT,
20                  mySecondData);
    setMySecondData(mySecondData);

    uint32_t myFirstData = 0;
    d.read(CRC32 < HESPERIA_CORE_STRINGLITERAL5('d', 'a', 't'↩
    , 'a', '1') >::RESULT,
25                  myFirstData);
    setMyFirstData(myFirstData);
    return in;
}
...
```

Listing 5.2: Compile-time computation of identifiers for serialization.

A sample usage of the `SerializationFactory` and the compile-time computation of indices is shown in Listing 5.2. The first method writes two attributes into a given output data stream. Both attributes have a unique identifier which is computed at compile time from the given human readable name. The second method queries a given input stream to retrieve both attributes in an arbitrary order using human readable names again which are mapped to unique identifiers at compile-time. Thus, a fail-safe usage of data serialization and deserialization can be provided by the software framework *Hesperia*.

5.3.2.5 Generic Directed Graph

For data structures representing nodes connected using directed edges, `libcore` provides a wrapper for a generic directed graph around the Boost Graph Library [144]. Alike the wrapper for the parser classes, this class can be used without a factory as shown in Figure 5.5.

The main class for creating and operating on a graph is `DirectedGraph`. This class constructs a directed graph from any object implementing the interface `Vertex` as node and from any object implementing the interface `Edge` as connection between the graph's nodes. The most important methods provided by `DirectedGraph` are `void updateEdge(const Vertex *v1, const Vertex *v2, const Edge *e);` and `vector<const Vertex*> getShortestPath(const Vertex &v1, const Vertex &v2);`. The former method constructs or updates the graph by either inserting an edge e between the nodes `v1` and `v2` or updating an existing edge.

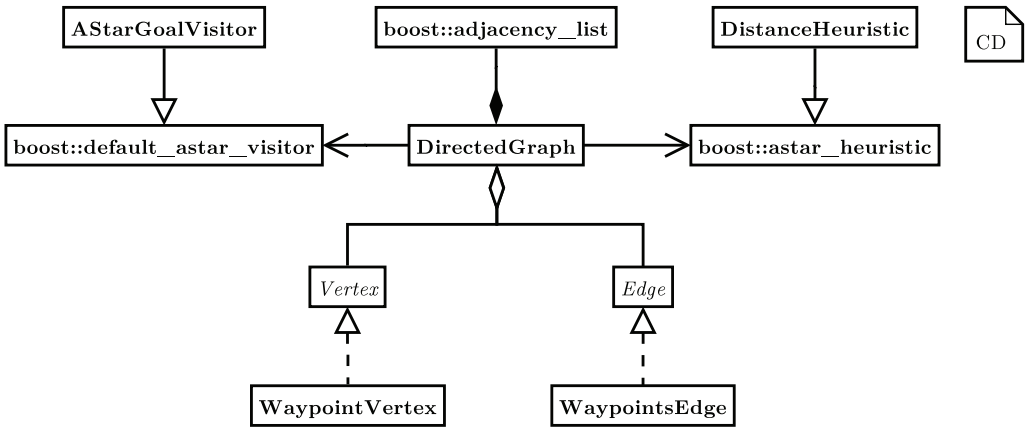


Figure 5.5: Generic directed graph based on the Boost Graph Library [144]. This provided concept encapsulates the underlying library and provides an integrated interface to the user-supplied applications on higher layers. Thus, the construction and handling of graphs and their algorithms are simplified.

The latter method tries to find the shortest path between the given two vertices. Therefore, `DirectedGraph` uses an A∗-algorithm provided by the wrapped Boost Graph Library itself [14] evaluating the edges' weights. Thus, the complex interfaces provided by the wrapped library could be reduced to an essential subset of at least necessary objects and methods. Therefore, they could completely be hidden from high-level applications to avoid errors due to misuse without restricting the performance of the wrapped library itself.

5.3.2.6 Convenient Data Exchange

Another basic concept provided by `libcore` is realized by the package `io`. This package simply implements the `StringListener` concept from package `wrapper` in the class `ContainerConference`. This class uses a UDP receiver for joining a UDP multi-cast session and registering itself as a `StringListener` at the `UDPReceiver`. Using a UDP multi-cast session, this class automatically receives any packet sent to this multi-cast group without creating a technical dependency between communication partners. Whenever a new `std::string` is received, this class tries to parse a `Container` from this buffer. Since this class implements the interface `ContainerObserver`, higher layers can register a `ContainerListener` to get notified about incoming `Containers`. Thus, any application can transparently receive complex data structures without bothering to deserialize the data or to setup a communication. Right after instantiation, the application starts receiving messages. Furthermore, to filter incoming `Containers`, a thread-safe data-structure as already described in Section 5.3.2.3 can be used easily. Thus,

different filtering concepts like FIFO-, LIFO-, or key/value-data-store can be realized either for only one specific type or for all incoming data. Furthermore, the data-stores can be reused and registered several times to combine several incoming streams of `Containers` if necessary.

All these concepts are the base for `libhesperia` which realizes further concepts allowing the simplified creation of distributed applications using a mature and stable API.

5.4 Providing Extensible and Re-Usable High-Level Concepts: `libhesperia`

On top of `libcore`, the component `libhesperia` is provided as the core component for the framework $\mathcal{H}$esperia. Its main concepts are described in the following.

5.4.1 Concept: *ClientConference*

The main concept for communication implemented in `libhesperia` is realized using UDP multi-cast and called *ClientConference*. A client conference is created using a unique identifier. Every application can simply join an existing ClientConference by setting the obligatory command-line parameter `-cid` appropriately.

All data exchanged in a ClientConference is wrapped in `Containers` as mentioned in Section 5.3.2.6. For receiving a `Container` of a special type, the application must simply decide the manner for getting the data. As described earlier, `libcore` provides rudimentary and thread-safe data-stores. An application simply registers a data-store separately for different `Containers` or uses the same data-store for all data. As soon as new data of the desired type is sent within the UDP multi-cast group, it is automatically received by `libcore` using a `StringPipeline` and placed into the registered data-stores for further processing. Using the `StringPipeline`, data receiving and processing is decoupled and the processing thread cannot block the receiving thread.

5.4.2 Concept: *Dynamic Module Configuration*

For configuring the application, a central and thus consistent configuration concept called *Dynamic Module Configuration* was implemented. This concept uses a Dynamic Module Configuration Protocol (*DMCP*) inspired by the well-known Dynamic Host Configuration Protocol (*DHCP*) for configuration clients in networks. As soon as an application is

started, it sends DMCP_DISCOVER messages to a specific port of the desired ClientConference using UDP multi-cast.

For deploying configurations for a specific ClientConference, a special component called supercomponent must be running. This component listens for DMCP_DISCOVER requests and replies using the DMCP_RESPONSE message containing information about the supercomponent itself. These information contain the IP address as well as a listening Transmission Control Protocol (*TCP*) port. Using these parameters the new application establishes a dedicated connection to the running supercomponent.

The newly created TCP connection is used to provide the application specific configuration using a simple key/value text file, wherein all keys can be hierarchically ordered or annotated using an application specific identifier. The application-dependent configuration is generated using one single configuration as shown in Listing 5.3.

```
# GLOBAL CONFIGURATION
#
global.scenario = file://../../Scenarios/CampusNord.scnx

# CONFIGURATION FOR PLAYER
#
player.input = file:///dev/stdin
player.autoRewind = 0
player.remoteControl = 0
player.sizeOfCache = 1000
player.timeScale = 1.0

# CONFIGURATION FOR PROXY
#
proxy:1.irt.insdata.server = 192.168.0.45
proxy:1.irt.insdata.port = 2345
proxy:2.jaus.controller.server = 192.168.0.100
proxy:2.jaus.controller.port = 3794
...
```

Listing 5.3: Centralized configuration concept.

This configuration is read completely by the supercomponent at start up. Whenever an application sends a DMCP_DISCOVER message, the application's name is transferred. This name is the first part of all keys in the configuration file called section delimiter. It is followed by an optional numerical identifier for distinguishing several running instances from the same type. Next, the key which can be hierarchically structured itself is specified

followed by the actual value. Using the newly created connection together with application specific subsets of one single configuration file provided by only one source, different configurations can be both maintained centrally and deployed for specific applications on demand.

5.4.3 Concept: Enhanced Data Structures

As already shown in Figure 5.2, `libhesperia` offers an enhanced object-oriented set of data structures which are serializable to be exchanged between independent processes which might be running on different nodes in a network. These data structures will be explained in the following.

- `hesperia::data::can`. This package contains a Controller Area Network (*CAN*) message. It can be used to encapsulate raw data read from or written to a CAN bus.
- `hesperia::data::dmcp`. Inside this package, all messages for joining an application in a ClientConference using a dedicated `supercomponent` are provided. Furthermore, statistical data about all running applications can be collected by the `supercomponent` itself as described in Section 5.5.
- `hesperia::data::environment`. This package consists of all necessary basic data structures to model elements from the surroundings and to apply manipulations to them. As already described in Section 4.2, every rigid body in the modeled environment is represented by a three-dimensional position and orientation. This representation is realized in `Position`. The most important derivative is `PointShapedObject` enriching the latter data structure by information about velocity and acceleration. For modeling the own AGV in the surroundings for example, this data structure is simply derived to `EgoState` for convenient purposes only. For mapping objects detected by sensors, either the `PointShapedObject` or an `Obstacle` enriching the latter one by the detected object's shape can be used. Besides these mappings, data structures containing all necessary operations for Cartesian coordinates, WGS84 coordinates, matrices, and quaternion representations are provided.
- `hesperia::data::image`. This package contains all meta information about an image like dimension or color depth. This data is intended to be used for exchanging images between several processes using a shared memory segment.
- `hesperia::data::player`, `hesperia::data::recorder`. These

packages contain command messages to control either the playback or the non-reactive recording of data from a running system.

- `hesperia::data::scenario`. This package contains a tree-like data structure describing the surroundings' model as described in Section 4.4.1. Combined with the `CompressionFactory` realized in `libcore`, an intuitional and convenient access to all attributes of the surroundings' model is provided.
- `hesperia::data::situation`. Complementary to the previous package, this one contains all information about the dynamic system's context as described in Section 4.4.1.
- `hesperia::data::sensor`. In this package, data structures for wrapping sensor's raw data like laser scanner data [143] or selective messages from the National Marine Electronics Association (*NMEA*) 0183 format describing GPS data [110] like the GPS Recommended Minimum Specific GPS/Transmit Data (*GPRMC*) are provided.

Besides these packages, a data description language was designed to simplify the creation of new data structures. The language is shown in Listing 5.4 and was defined using MontiCore as well.

```
grammar DataDescriptionLanguage {

  DataDescriptionFile = (DataStructures:DataStructure)+;

  DataStructure =
    FullQualifiedPackageName:FullQualifiedPackageName
    Name:IDENT
    (":" SuperDataStructure:SuperDataStructure)?
    "{" Attributes:Attributes "}";

  FullQualifiedPackageName = (PackageName "::")*;

  SuperDataStructure =
    FullQualifiedPackageName:FullQualifiedPackageName
    Name:IDENT;

  PackageName = Name:IDENT;

  Attributes = TypeDeclaration*;

```

```
  TypeDeclaration =
    TypeName:IDENT
    isList:"*"?
    Name:IDENT ";" ;
}
```

Listing 5.4: Data description language.

The main purpose behind this language is to simplify the error-prone and time-consuming process of correctly creating new data types. Due to the provided concepts for serialization, even a small amount of attributes which should be exchanged between several applications requires several portions of source code which is similar for any data structure. Thus, a small application based on MontiCore was realized which allows an easy definition of new data structure as shown in Listing 5.5.

```
environment::Point2 {
  double x;
  double y;
}
environment::Point3 : environment::Point2 {
  double z;
}
environment::PointSet {
  Point3* listOfPoints;
}
```

Listing 5.5: Example for the data description language.

The application which processes these instances of the grammar creates appropriate header and source files for C++. Furthermore, the required getter- and setter-methods are generated as well as the serialization and deserialization methods. For lists, methods to add new items and to retrieve the entire list are generated. Moreover, methods to allow a copy of the data structure are derived automatically.

5.4.4 Concept: Integration of Modeling DSL

As already mentioned in Section 4.4.1, the DSL for modeling the stationary and dynamic elements of the surroundings is directly integrated in `libhesperia`. Thus, it is very convenient to access modeled elements both in the framework itself and in applications if desired. In the following, the processing of the language inside `libhesperia` is outlined.

For using the DSL, a lexer and parser for processing instances of the grammar are necessary. As already described in Section 4.5.2, Spirit was enhanced to allow convenient use by creating the data structure provided by `hesperia::data::scenario` and `hesperia::data::situation`. The main ideas behind the language processing are outlined on the example of the stationary modeling.

Using the aforementioned `CompressionFactory`, a data structure called `SCNXArchive` created by `SCNXFactory` encapsulates the handling of SCNX archives. The data structure exports methods for accessing simply the parsed and mapped grammar as already described in Section 4.5.2 and allows access to the defined aerial and height images in the archive using the `ImageFactory` as outlined before. Furthermore, all associated complex models using 3D modeling tools are exported to the caller including their meta data and the content of the OBJX archive as well. Finally, all situations associated with a stationary model can be retrieved from this data structure.

5.4.5 Concept: Device-Independent Data Visualization

For providing an intuitional understanding of the complex surroundings, a visualization for the data structures describing attributes and relations between elements of the modeled stationary and dynamic surroundings is necessary. But instead of enforcing only one type of visualization, a device-independent data visualization is provided by `libhesperia`.

The main principle is shown in Figure 5.6 on the example for the stationary surroundings. Instead of mapping the existing data structures to a data structure dedicated for representation only by producing redundant data, the existing tree generated by the DSL processing framework can simply be reused. Therefore, the generated tree is traversed using the interface `ScenarioVisitor`. This interface is implemented by `ScenarioRenderer`, the device-independent and data-dependent renderer. This class implements the `void visit(ScenarioNode&);` method and delegates every call to type-dependent methods like `void visit(Polygon&);` using type conversion at run-time.

Furthermore, `ScenarioRenderer` has an associated `Renderer` providing a set of pure virtual methods to be meant for primitive drawing operations like `void setPointWidth(const float&);` or `void drawLine(const Point3&, const Point3&);`. These methods are called by the type-dependent visiting methods during tree traversal. From this abstract class, `Renderer2D` and `Renderer3D` are derived. The former one is also an abstract class mapping all three-dimensional drawing operations into two-dimensional ones, while the concrete drawing methods are still left unimplemented. The latter one is an implementation of drawing primitives using OpenGL

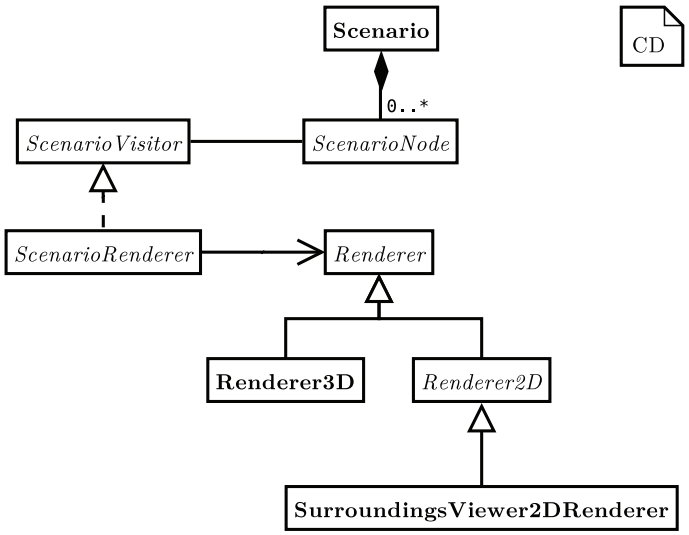


Figure 5.6: Device-independent data visualization: Any data which should be visualized uses the interface `Renderer` which provides rudimentary drawing primitives for drawing points or lines in $\mathbb{R}^3$ for example. This interface is implemented by a concrete implementation for the OpenGL context which is also provided by `libhesperia`. For generating a 2D view on a given data, some methods from the interface `Renderer` are combined by flattening the z coordinates which is realized in the abstract class `Renderer2D`. Thus, a concrete realization which uses this class simply implements the reduced set of drawing primitives which is outlined in Section 7.2.

as a platform-independent industrial standard; this renderer is used both for visualization as outlined in Chapter 7 and for simulation purposes as described in Section 6.4.

Since `Renderer2D` is still an abstract class which misses its concrete device-dependent implementation, the use of both is described in detail in Chapter 7 for a non-reactive monitoring application. This application is meant to visualize the stationary surroundings with their dynamic elements.

Using the concept described here, a concrete visualization using a device-independent data representation can be realized. Furthermore, the existing tree-like data structures for the surroundings' scenario and situations can be simply reused. Thus, no additional data structure for visual representation depending on a special scene graph library like [114] is necessary.

5.5 Applications' Life Cycle Management: `supercomponent`

The special support application `supercomponent` is responsible for managing a concrete ClientConference. A running system which uses a ClientConference for communication consists of several user-contributed applications or tools which are part of the software framework $\mathcal{H}$esperia. All these applications must connect at their start-up to the supervising `supercomponent` which manages a centrally provided system configuration. An application-dependent subset of this configuration data is deployed automatically to a connecting client application using the newly established TCP connection initiated by the client.

Furthermore, this component receives periodically sent `RuntimeStatistics` from every client application participating in a ClientConference. This data structure contains information about the time consumed for computation relative to the defined client's individual frequency. The `supercomponent` assembles all `RuntimeStatistics` into a periodically sent `ModuleStatistics`. These information can be used to evaluate the system's performance for example.

Furthermore, `supercomponent` is notified whenever any client application leaves the ClientConference either regularly by a return code sent by the leaving client at exit or technically, when a client application exits unexpectedly through the invalid TCP connection to the lost client application. These information can be used to track any problems in a running system consisting of several independent applications for example.

5.6 Supporting Components

Besides the application `supercomponent`, several small other tools are part of the framework $\mathcal{H}$esperia. These are outlined briefly in the following.

5.6.1 Component: `proxy`

This component must be used to translate data structures between systems provided by different independent suppliers. Therefore, it joins a ClientConference to broadcast data received by a system provided by a third party or it sends data to a system received from the ClientConference. Due to the concept ClientConference, every application running on top of the framework $\mathcal{H}$esperia automatically and transparently communicates with a

third party system. This application is used in the case study to communicate with the AGV as described in Chapter 8.

5.6.2 Component: `recorder`

As expected, this component non-reactively records every `Container` broadcasted in a running ClientConference to a file. Therefore, it simply registers itself as FIFO-receiver for every broadcasted `Container` and uses an in-memory cache running concurrently for decoupling disk I/O operations. The resulting file contains all received `Containers` serialized in chronological order. For writing the file, `std::fstream` can easily be used with the framework *Hesperia*. Moreover, this tool can be controlled remotely to suspend or resume a running recording session using `RecorderCommand` sent to the ContainerConference.

5.6.3 Component: `player`

As counterpart to `recorder`, `player` replays previously recorded ClientConferences by using a given recorded file or simply by reading from `stdin`. Furthermore, `player` can be configured to scale the time between two broadcasted `Containers` to perform a faster or slower playback either infinitely or only once. Just like `recorder`, `player` can be controlled remotely to suspend, to resume, to rewind, or to play stepwisely recorded data using `PlayerCommand`. Moreover, the input data is cached before replayed to provide a continuous playback stream of `Containers` to avoid interfering the delay between two records when accessing data on the disk.

5.6.4 Component: `rec2video`

Like `player`, `rec2video` simply reads a previously recorded ClientConference. But contrary to the former, this component uses the three-dimensional data representation to compute single images to be rendered afterwards into a video file for demonstration purposes. Therefore, it adjusts the playback of previously recorded `Containers` to 25 frames/second using a simulated clock to control the rendering for the next frame of the current system's state.

6 Simulation of the System's Context

In this chapter, $\mathcal{H}$esperia's use for the simulation of the system's context is described. First, general considerations and design drivers are outlined to be regarded for the simulation of the system's context for sensor- and actuator-based autonomous systems. Next, an overview of the component for the simulation of the system's context realized in `libcontext` in the framework $\mathcal{H}$esperia is described. In the following, several major aspects of the simulation of system's context's are presented including the computation of the position, rotation, velocity, and acceleration of the AGV as well as of dynamic elements from the surroundings. Furthermore, the generation of sensor specific low-level data like cameras and laser scanners as well as high-level data abstracting the surroundings is presented.

6.1 General Considerations and Design Drivers

As shown in Figure 6.1, the main design principle for sensor- and actuator-based autonomous systems as well as an AGV is a data flow-oriented design realizing the well-known pipes and filters design pattern reacting on stimuli from the system's context. Thus, incoming data is processed in an encapsulated manner for extracting relevant features and producing a set of enriched or modified information for the next stages.

Considering this data flow, the component which realizes the production of synthetic data based on environmental information by using the DSL for stationary and dynamic elements is called *simulation of the system's context* which is indicated as the *Virtualization Layer* in Figure 6.1. This layer is responsible for the simulation of a running SUD which is processing continuously incoming data by a discretization for the valid overall system time. Hence, this layer must control and increment the system clock and manage all involved applications of the SUD to use the controlled time at their specific schedule.

Therefore, the adaption of the virtualization layer for autonomous vehicles consists of the generic time controlling and SUD management which is realized in `libcontext` and the context-dependent models like the computation of the position, rotation, acceleration,

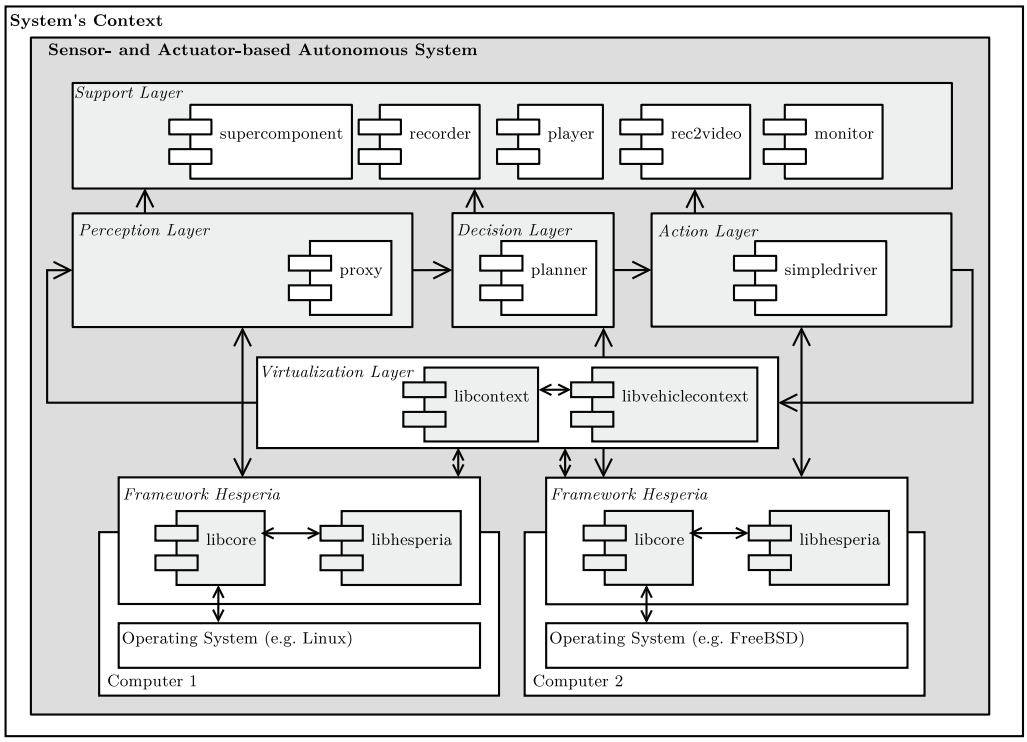


Figure 6.1: Integration of the system simulation within the framework $\mathcal{H}$esperia: Using concepts and algorithms from $\mathcal{H}$esperia, the application which realizes the sensor- and actuator-based system is on top of $\mathcal{H}$esperia. For closing the loop between the action layer and the perception layer to enable interactive and unattended simulations, the virtualization layer with `libcontext` and `libvehiclecontext` is used. While the former is necessary to realize the run-time discretization for the SUD by providing time control and scheduling, the latter provides models like the bicycle model for the specific use for autonomous vehicles for example. Thus, the framework $\mathcal{H}$esperia provides an application-independent virtualization layer to realize system simulations.

and velocity for the AGV for the next time slice based on a given model. This is realized in `libvehiclecontext` and is therefore a specific customization for the virtualization layer. Moreover, not only the AGV must be simulated but also components from its system's context which serve as the surroundings' model to generate specific input data.

Furthermore, the simulation of system's context must use an interface from the system to provide all environmental information *depending* on the input stage for the pipes and filters processing chain where the synthetic data enters the system. While on higher layers like the decision layer abstract objects with discrete information are identifiable at every time slice, on lower layers like the perception layer input data is gathered from sensors detecting the surroundings. To avoid modifications of the SUD for providing the required input data, the simulation of the system's context must compute synthetic input data from the current system's context's situation and feed it to the pipes and filters chain in the

same format like a real sensor would do.

Regarding these information and the aforementioned major design principle of sensor- and actuator-based autonomous systems, the following list contains some general considerations and design drivers for the simulation of the system's context.

- *Decoupling from the real time.* To provide both interactive as well as unattended simulations of the system's context, which can be suspended or resumed using all available computing power, the simulation of the system's context must decouple the system time which is valid for the system and the system's context from the real time. Therefore, not only the simulation of the system's context but also the components being used together with the simulation of the system's context must use the *virtual time base*. On a real system, the virtual time base is equal to the real time. Thus, the simulated system would run as fast as possible depending on the calculated largest time step which fulfills each required frequency from all SUD's applications.
- *No visualization.* Often, simulations are equalized with intuitional visualizations. In the following, the terms simulation and visualization describe different things and are *not* used interchangeably. The simulation is responsible for controlling the system time, the scheduling, and the computation of the reactions of the system's context. Contrary to the simulation, the visualization completes a simulation by visualizing the enormous data in an intuitional manner. Thus, the visualization is a front-end from a simulation application but independent from the simulation and can be reused in further contexts as described in Chapter 7.
- *Providing stage dependent data.* As already outlined before, the type and amount of synthetic input data depends on the layer of the pipes and filters data processing chain where it enters the system. Therefore, highly detailed raw data must be provided at the lowest level which is the perception layer as shown in Figure 6.1, while only selected information from the system's context in an abstract representation may already be sufficient on higher layers like the decision layer to generate desired actions in the system.
- *Extensibility.* As shown in Figure 6.1 as well, the actually required simulation context like vehicle models for AGVs is independent from the generic system simulation. While the latter is responsible for the time control and overall scheduling, the former provides models which describe relations and aspects for the system or its contexts. Thus, the system simulation shall be independent from the actually required models for parts of the SUD or its context and must instead realize only the required concepts for carrying out system simulations.

Before selected parts from the simulation of the system's context are presented which realize the aforementioned design considerations using the framework $\mathcal{H}$esperia , further considerations related to the management of an SUD and its system's context are outlined. Furthermore, the control of the system-widely used time is discussed.

6.2 Controlling an SUD and the Time

According to [178], systems can be in general classified into *Discrete Event System Specification*, *Discrete Time Specification*, and *Differential Equation System Specification* which also defines the required simulation technique to be used. Characteristics of these systems are outlined in the following:

- *Discrete event system specification (DEVS)* Starting at an initial system state, these systems are specified by a set of ordered timed events which activate a transition to the next valid system state. The required simulator is an event-based processor which supervises the event processing and the state transitions [115].
- *Discrete time system specification (DTSS)* These systems have a discrete time base and thus, subsequent system states can be calculated from the results of the previous time step. The necessary simulator for this class is a recursive simulator to calculate the difference equations. Its mathematical model is $s(t+1) = a * s(t) + b * x(t)$.
- *Differential equation system specification (DESS)* Contrary to the aforementioned systems, these ones have a not only continuous time but also continuous states. Therefore in general, the required simulator is a numerical integrator for calculating the differential equations whose mathematical model is defined as $q' = a * q + b * x$.

To complete the aforementioned list there are some other simulation types to mention. For example, *Monte Carlo Simulations* [105] which are static simulations or *System Dynamics* [55] which is mainly used to describe and to analyze complex systems like economic relations.

Considering the methodology for automating the acceptance tests for sensor- and actuator-based systems as outlined in Chapter 3, the entire SUD must be controlled to supervise its control flow and communication during an acceptance test. Therefore, the independently operating applications must be executed in a deterministic order according to their specific execution frequency.

When the frequency for each application is constant for the entire run-time and known a priori, a deterministic execution order can be calculated. In the case of equal frequencies,

a manually specified execution order is defined to preserve the deterministic execution order. Furthermore, the maximum allowed time step for fulfilling the required execution order of all SUD's applications can be derived as well. Thus, a DTSS is described.

Using these preconditions, the simulation consists mainly of two major parts: An SUD's scheduling, communication, and time control simulation environment which is defined as $\mathcal{S}_{\text{sched}}$ and the system's context simulation applications called $\mathcal{S}_{\text{env}}$ which is domain specific for the SUD. For an AGV, an exemplary application from $\mathcal{S}_{\text{env}}$ might be a synthetic sensor's raw data provider as outlined in Section 6.4.6. In Listing 6.1, the scheduling algorithm is defined.

```
function getMaximumDeltaT(list SUDsApplications, list ↩
SystemContextApplications)
  deltaT := 1000/SUDsApplication.head().getFrequency()

  for each app in SUDsApplications:
    deltaT := greatestCommonDivisor(deltaT, 1000/app.↩
    getFrequency())
  for each scapp in SystemContextApplications:
    deltaT := greatestCommonDivisor(deltaT, 1000/scapp.↩
    getFrequency())

  return deltaT

function needsExecution(T, app)
  return ((T \% (1000/app.getFrequency())) == 0)

procedure Scheduler(list SUDsApplications, list ↩
SystemContextApplications)
  T := 0
  deltaT := getMaximumDeltaT(SUDsApplications, ↩
  SystemContextApplications)

  if t > 0 then
    while true
      for each app in SUDsApplications:
        if needsExecution(T, app) then app.step((1.0/app.↩
        getFrequency()))

      for each scapp in SystemContextApplications:
```

```
        if needsExecution(T, scapp) then scapp.step((1.0/app.↩
        getFrequency()))
25
      T := T + deltaT
```

Listing 6.1: Pseudo-code for the general scheduling simulation.

As shown in Listing 6.1, the overall valid system time T is incremented step-wisely in line 26 using the maximum possible constant time step ΔT which is computed in milliseconds in line 16. This constant time step is derived from the joined set of all applications from the SUD and the applications from the system's context ($\mathcal{S}_{\text{env}}$) using the function specified in line 1 et seqq. Thus, a fixed step-continuous clock is provided to the SUD and all applications from the system's context. This clock is independent from the real system time and therefore can be incremented as fast as possible. The function assumes that no frequencies are provided which are zero; moreover, the resulting constant minimal time step is at least 1 if no common greatest divisor greater than 1 can be calculated for the SUD and all applications from the system's context.

Following, the currently valid system time T is used to determine all applications from the SUD which must be executed with the constant time step which is shown in line 20 et seqq. Therefore, its required frequency is used to calculate whether it must be executed at the currently valid system time using the function which is specified in line 11 et seqq.; if an execution is required the application performs a step forward using its individual time step. Thus, a predefined deterministic execution order for the SUD's applications is preserved. The same algorithm is applied analogously for the applications of the system's context.

Thus, the outlined scheduler $\mathcal{S}_{\text{sched}}$ realizes a simulator for the aforementioned DTSS which is used for controlling the SUD and the applications of its system's context. Due to the architectural encapsulation of the scheduler $\mathcal{S}_{\text{sched}}$ which only manages the system time and the overall scheduling, and the system's context $\mathcal{S}_{\text{env}}$, different simulation algorithms can be realized to provide the required data from the system's context. Thus, a simulation which uses a numerical integrator can be regularly triggered from $\mathcal{S}_{\text{sched}}$ for example.

In the following, the implementation within $\mathcal{H}$esperia for the outlined concept of $\mathcal{S}_{\text{sched}}$ is presented in greater detail.

6.3 Controlling Applications: `libcontext`

In this section, `libcontext` for realizing simulations of the system's context is outlined. This library provides a system's context-independent simulation for any application realized with the framework $\mathcal{H}$esperia without the necessity of modifying the application itself according to the general scheduling and controlling algorithm as outlined in Section 6.2. The general algorithm for the simulation provided by `libcontext` and realized with the framework $\mathcal{H}$esperia is outlined in the following:

1. *Initialize system's context.* First, the entire communication in the system as well as the time must be replaced by a controllable instance respectively.
2. *Setup supervision for the control flow of the system under test.* For supervising the control flow of the application as well as to validate its computation, the system under test must be controllable as well.
3. *Initialize virtualized clock.* Setup the desired system's time. Furthermore, compute the maximum possible time increment fulfilling all requested frequencies by all systems under test as well as all system's parts.
4. *Do step for the system's parts.* Compute the next values in the system's parts like a virtualized sensor's raw data provider or any high-level data provider according to their required frequencies.
5. *Do step for the systems under test.* Step forward for all systems under test regarding to the required frequencies. Moreover, enable communication for the currently activated system under test until it completes one computing cycle.
6. *Evaluate.* If required, validate the computed and sent data from the systems under test.
7. *Increment virtualized system's time.* Increment the clock by the computed maximum possible time step and start over.

In the following, the architecture which realizes this algorithm is described in detail.

6.3.1 Interface for Controlling Time and Communication

As shown in Figure 6.2, the general inheritance for any application realized with the framework $\mathcal{H}$esperia is depicted on the left hand side. The super-class for every application is `AbstractModule`, which is further specialized into `InterruptibleModule`. This abstract class realizes a concept for controlling the application's control flow which is described later. Besides a parser for arguments passed to an application using the

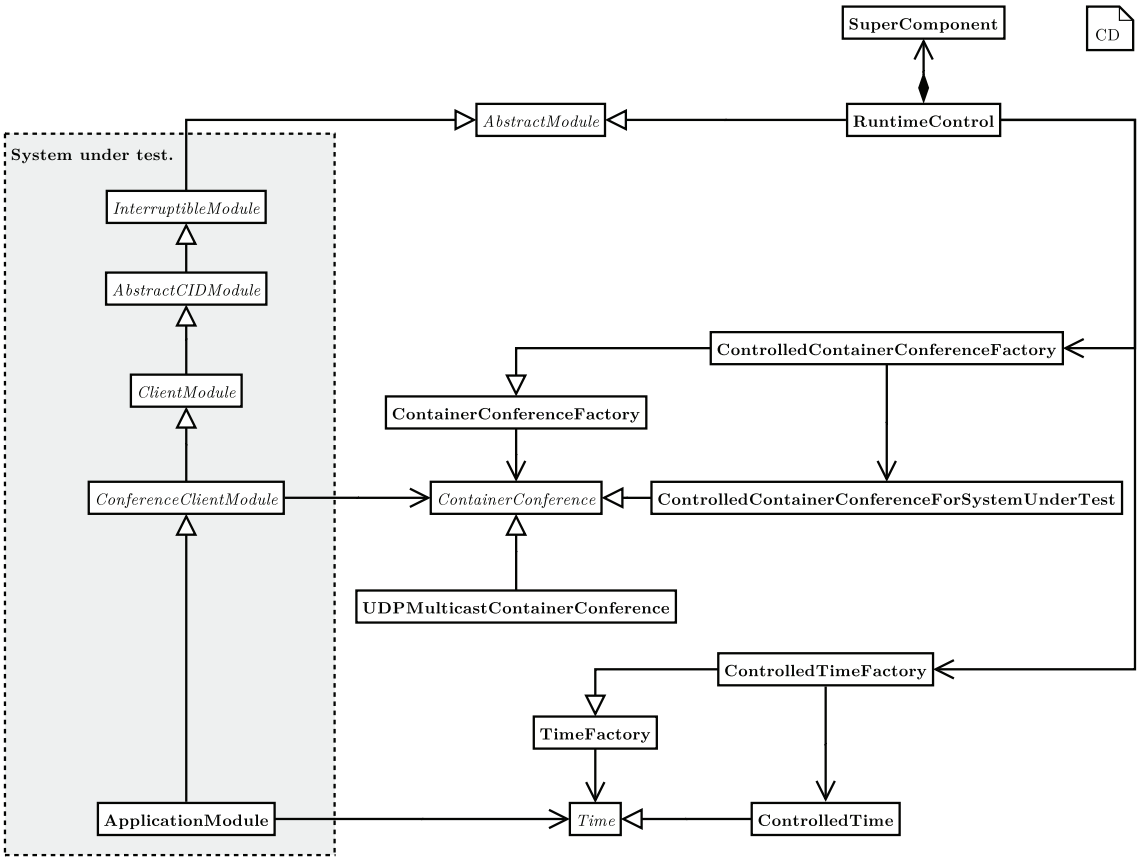


Figure 6.2: Control of time and communication: On the left hand side, the SUD is shown which is realized in `ApplicationModule` by the user. For controlling the overall system time and communication, the class `RuntimeControl` overrides the regular implementation of `ContainerConferenceFactory` by providing a pure software solution which manages the sending and receiving of `Containers` between several applications. The same concept is applied to the `TimeFactory` which is intercepted by a specialized variant which allows the controlled incrementation of the system-wide time.

command line, `AbstractCIDModule` provides information about the desired multicast group for realizing a ContainerConference. The `ClientModule`, which derives from the latter class, implements the concept for realizing a DMCP client to retrieve configuration data from a `supercomponent`. Finally, this class is further specialized into `ConferenceClientModule` to join a `ContainerConference` using the `ContainerConferenceFactory`. The actual application itself derives from the latter class and implements the necessary methods `void setUp();`, `void body();`, and `void tearDown();` resulting in the regular application's state machine.

Also deriving from `AbstractModule`, `RuntimeControl` is the core class for realizing simulations of the system's context. This class shall substituting the context for all implemented components, thus, it consists furthermore of a selected im-

plementation of a `supercomponent` for providing configuration data to any application which is executed under supervision of `RuntimeControl`. Moreover, this class has access to `ControlledContainerConferenceFactory` as well as `ControlledTimeFactory` which are both used to replace the regular ones.

For replacing the `ContainerConference` which represents a UDP multi-cast group, `ContainerConferenceFactory` is derived to `ControlledContainerConferenceFactory` which returns a specialized `ContainerConference` for controlling the communication to the requesting component as well as the communication initiated by that component. The use of the `ControlledContainerConferenceForSystemUnderTest` is described later; an instance of this class is returned to a component upon request.

Comparable to `ControlledContainerConferenceFactory`, the `ControlledTimeFactory` is derived from `TimeFactory` and is used to control the current valid time system-widely by returning the controlled time upon request. Since a component uses the enhanced class `TimeStamp` for realizing time computations, it transparently uses the substituted factory because `TimeStamp` itself uses `TimeFactory`.

6.3.2 Supervising an Application's Control Flow and Communication

With both factories, the software architecture for substituting the interfaces for realizing communication or to request the current time is defined. Their usage is shown in Figure 6.3.

Starting at `RuntimeControl`, a `RuntimeEnvironment` is passed to this class describing the `ConferenceClientModules` to be executed for the system under test and all `SystemContextComponents` describing either unavailable parts of the system, for example an application realizing a sensor data fusion or even an application computing sensor's raw data. At least one element of both types must be provided to start a simulation of the system's context. Both the abstract class `AbstractCIDModule` and the interface `SystemContextComponent` derive from the interface `Periodic` providing the method `float getFrequency() const;`. Thus, for every application as well as for any system's part, different but constant run time frequencies can be defined. By default, 1Hz is used for applications if nothing is defined. Depending on these frequencies, `RuntimeEnvironment` computes the greatest possible time step to use for correctly fulfilling every requested frequency. This time step is used to continuously increment the virtualized system time.

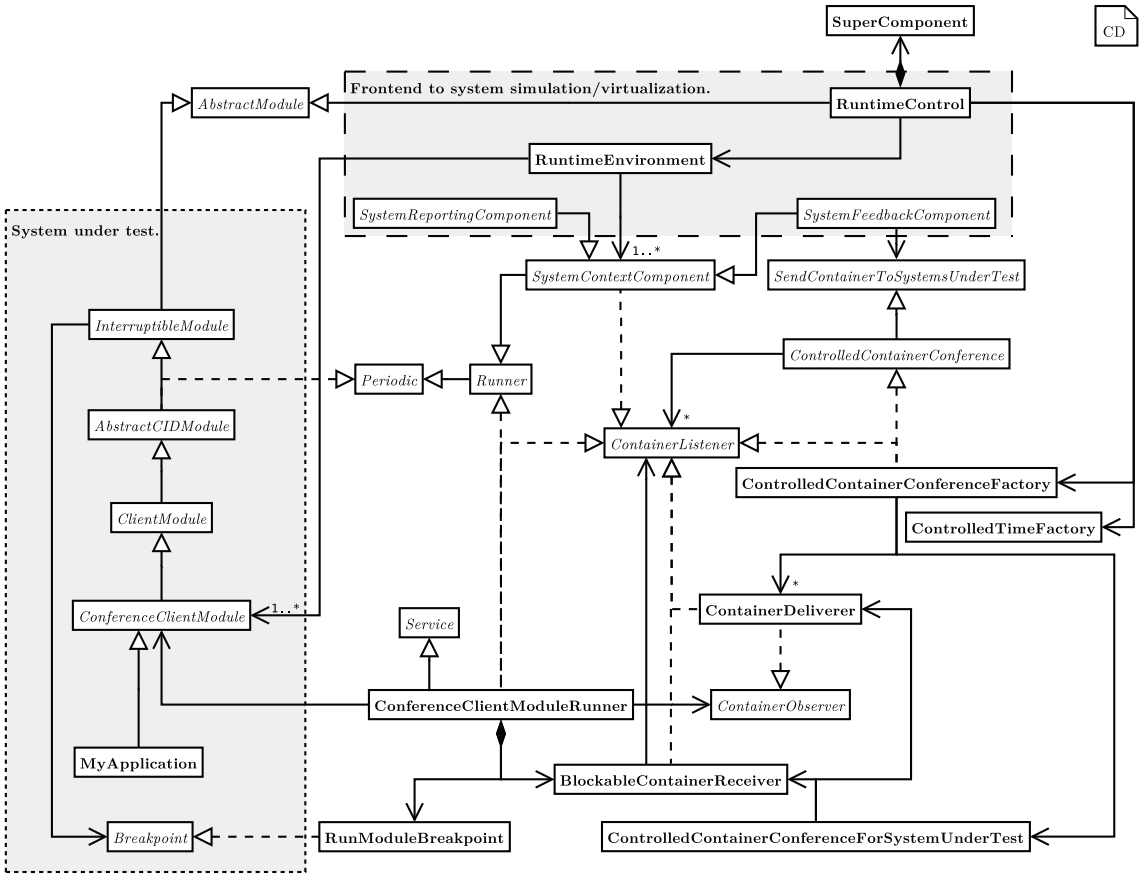


Figure 6.3: Run-time control for the SUD: On the left hand side, the SUD is shown which implements automatically the interface `InterruptibleModule` and `Periodic`. While the former is required to register a special object called `Breakpoint` to intercept regularly the running application, the latter is necessary to calculate and realize a system-wide scheduling which is implemented by `RuntimeEnvironment`. This class controls and schedules the required components' frequencies; furthermore, it supervises the sending and receiving of `Containers` by using the class `ControlledContainerConferenceFactory`.

The method `void step(const core::wrapper::Time&);` provided by the interface `Runner` is used to actually perform a step either in the simulation of the system's context or in the system under test realized by a class derived from `ConferenceClientModule`. The call to this method implemented by the system under test as well as all system's parts is initiated by `RuntimeControl` providing the absolute current valid virtualized system's time. Thus, every class implementing the interface `Runner` is called with its desired frequency for a constant time step by simply computing the difference using the time from the previous call and the current time at method's call. For scheduling all systems under test and all `SystemContextComponents`, `RuntimeControl` realizes a *time-triggered, completing computation-scheduler*, con-

sidering the desired frequencies. Thus, whenever either an application from the system under test or a `SystemContextComponent` must be executed for a given time t, `RuntimeControl` executes the instance exclusively until it completes its computation. The scheduling is outlined in detail in Section 6.3.2.3.

6.3.2.1 Supervising an Application's Control Flow

To supervise an application's control flow which derives from `ConferenceClientModule`, the concept *Breakpoint* is used. Since every `ConferenceClientModule` derives also from `InterruptibleModule`, the method `void setBreakpoint(Breakpoint*);` is available. The method takes an instance of an object implementing the interface `Breakpoint` and thus providing the method `void reached();`. This method is called whenever the application calls `ModuleState::MODULE_STATE getModuleState();`. Since every data processing application is intended to run infinitely depending on the return value of the latter method which is called periodically in the main loop's condition, the application is interrupted either before or right after one loop's iteration. Whether the application is interrupted before or after completing one loop's cycle depends on the type of main loop: If a head controlled loop is used, the interruption is occurring right before a loop's iteration, while a foot controlled loop causes an interruption right after completing a loop's iteration.

The interface `Breakpoint` is realized by the class `RunModuleBreakpoint`, which blocks the callee's thread to this method until a condition in this object is turning true caused by another thread. Thus, an application's main loop can be held until the next time step for this system under test is available. The instance of `RunModuleBreakpoint` belongs to exactly one `ConferenceClientModule` and thus is part of `ConferenceClientModuleRunner`.

The object `ConferenceClientModuleRunner` implements the interface `Runner` and realizes a stepwise execution using the aforementioned `RunModuleBreakpoint`. Thus, `RuntimeControl` can easily call the method `void step(core::wrapper::Time&);` for an instance of `ConferenceClientModuleRunner` wrapping the real object of the system under test while waiting for the next call to the method `void reached();`. For separating the `RuntimeControl`'s thread from the a system under test's thread, the wrapped application is run concurrently using `Service`. Thus, even in a case of an unforeseen thrown exception in the system under thread, `RuntimeControl` can safely catch the exception and generate proper reports.

6.3.2.2 Supervising an Application's Communication

For controlling the communication to a system under test or initiated by such a component, several aspects must be considered. Since `ContainerConferenceFactory` creates a suitable `ContainerConference` for encapsulating a UDP multi-cast group for exchanging `Containers` and thus, no direct connections to any other specific application must be opened by the application which avoids direct dependencies between all running applications. However, a `Container` sent to a `ContainerConference` is available for any joined application in that UDP multi-cast group. Moreover, to supervise not only a system under test's control flow as described before but also its specific communication, the application is only allowed to send data when it is scheduled to compute its main loop for the next time step. For realizing this behavior, all communication is routed by the `ControlledContainerConferenceFactory` to send data to any system under test or to distribute sent `Containers` to other systems under tests as well as `SystemContextComponents`.

Since the `ContainerConferenceFactory` expects an application's specific data for joining a UDP multi-cast group, which is provided by `AbstractCIDModule`, the application's specific `ContainerConference` is directly created in the constructor of `ConferenceClientModule` within `libhesperia`. Therefore, only one instance of a class implementing the interface `ContainerConference` is created for an application. The wrapper `ConferenceClientModuleRunner` takes advantage of this application's property by requesting the application's specific instance of `ContainerConference` inside its constructor. This instance is cast into `ControlledContainerConferenceForSystemUnderTest` containing further information to supervise the wrapped application correctly. If the cast fails, the user implementing a simulation for the system's context did not call the method `void setup(const enum RuntimeControl::RUNTIME_CONTROL&);` prior to the actual `RuntimeControl`'s `void run(RuntimeEnvironment&, const uint32_t &timeOut);` method to enforce `RuntimeControl` to replace any existing `ContainerConferenceFactory` with `ControlledContainerConferenceFactory`. Moreover, this misuse is checked by `RuntimeControl` as well.

The class `ControlledContainerConferenceForSystemUnderTest` consists of an instance of `BlockableContainerReceiver` as well as an instance of `ContainerDeliverer` realizing the communication from and to the system under test, respectively. The former is necessary to send a `Container` from the system under test to other systems under test as well as to all `SystemContextComponents`, while

the latter is responsible for distributing a Container to the system under test.

As already mentioned in Section 5.3.2.6, a system under test receives the data using a FIFO-, a LIFO-, or a key/value-data-store. This is realized by registering the current instance of ConferenceClientModule as a ContainerListener at the instance of ContainerConference. Thus, running an application as a system under test supervised by RuntimeControl, no changes to the application are necessary at all to use a ControlledContainerConference for receiving Containers.

Since the ContainerConferenceFactory does not have any control over a ContainerConference's life cycle because the application can simply destroy the instance, the communication controlled by ControlledContainerConferenceFactory must be decoupled from any returned instance to a system under test. Hence, every ControlledContainer-ConferenceForSystemUnderTest consists of an own instance of ContainerDeliverer which itself implements the interface ContainerObserver. This instance is used by the ControlledContainerConferenceFactory to distribute a Container to a system under test. Therefore, the system under test's specific instance of ControlledContainerConferenceForSystemUnderTest which inherits from ContainerConference, registers itself as ContainerListener at the ContainerDeliverer. Hence, as soon as any Container is sent to the system under test using the ControlledContainerConferenceFactory, it gets delivered to any datastore registered for filtering by the application since the ContainerConference delivers the Container to the ContainerListener which was registered at itself. Since that ContainerListener is directly registered in the constructor of ConferenceClientModule which registers itself as ContainerListener, any Container is finally delivered to the application itself according to its filter setup using a FIFO-, LIFO-, or key/value-data-store.

Whenever the application destroys its instance of ContainerConference and thus the ControlledContainerConferenceForSystemUnderTest, either regularly or by throwing an exception, the instance deregisters itself as ContainerListener at ContainerDeliverer in its destructor. The ContainerDeliverer now simply discards any Container sent to this system under test. Hence, problems caused by an unexpected thrown exception do not affect the ControlledContainerConferenceFactory and moreover no other supervised application or SystemContextComponent. Thus, the concept of ContainerDeliverer ensures that a Container sent at time t is delivered in the same time step synchronously without any delay. Furthermore, due to the thread-safe

implementation of the FIFO-, LIFO-, and key/value-data-store, the delivering thread cannot be blocked.

For implementing sending facilities for a system under test, it is necessary to restrict their communication to the time step when it gets activated by `RuntimeControl`. As mentioned before, every system under test has exactly one `ContainerConference` which is used to send data. Since `ControlledContainerConferenceForSystemUnderTest` derives directly from `ContainerConference`, this feature can be controlled by that instance. For blocking an application from sending containers when it is not activated, an application-dependent instance of `BlockableContainerReceiver` is used. This class implements the interface `ContainerListener` to provide the method `void nextContainer(Container&);` to the system under test, which calls it implicitly whenever the method `void send(Container&) const;` from `ContainerConference` is called.

Furthermore, `BlockableContainerReceiver` has an instance of an object implementing the interface `ContainerListener` as well. This instance is the `ControlledContainerConferenceFactory`, which finally distributes the `Containers` sent from the system under test to all systems under test registered at `RuntimeEnvironment` and to all `SystemContextComponents`.

For restricting an application to send its data only when it gets activated, the aforementioned concept of `Breakpoint` is simply reused. Whenever an application calls the method `void reached();` of the implementing class `RunModuleBreakpoint`, the instance `BlockableContainerReceiver` is locked to avoid sending data. Thus, every thread spawned by the system under test which tries to send data using the single instance of `ContainerConference` is blocked from returning until sending is allowed. For releasing all blocked threads and thus allowing sending again, right before returning the callee's thread to `void reached();` of the interface `Breakpoint`, the lock is removed from `BlockableContainerReceiver` and the system under test can send `Containers` until the break-point is reached again.

The decoupling of accepting `Containers` sent by an application and distributing the `Containers` realized by `BlockableContainerReceiver`, blocking the `ControlledContainerConferenceFactory` in processing `Containers` sent by systems under test due to malfunctions in their implementation can be simply avoided. Even in the case of an unexpected thrown exception by the system under test, the `ControlledContainerConferenceFactory` is still unaffected since the life-cycle of every `BlockableContainerReceiver`'s instance is controlled

entirely by `ControlledContainerConferenceFactory` and the system under test simply calls the method `void nextContainer(Container&);` of the interface `ContainerListener` implemented by `BlockableContainerReceiver`. If the application gets destroyed, the instance of the class `BlockableContainerReceiver` is not affected.

6.3.2.3 Scheduling Applications and `SystemContextComponents`

A sequence chart as an example for controlling a system under test is shown in Figure 6.4. On the topmost row of that figure, the minimum amount of necessary applications to run a simulation of the system's context are shown. For the sake of clarity, communication related instances like `ControlledContainerConferenceFactory` or `ContainerDeliverer` are left apart.

The first instance, denoted by its super-class `SystemContextComponent` realizes a specific part of the system context's simulation. Next to that instance, the overall valid system's clock is shown. The third component is the actual `RuntimeControl` which controls all other components, performs a step in the system's context using `SystemContextComponents` or the actual system under test using `ConferenceClientModuleRunners` respectively, and increments the time. This instance communicates only indirectly with the `ConferenceClientModule` using an instance of `ConferenceClientModuleRunner`, which is shown as fourth object. Followed by the instance of `RunModuleBreakpoint`, this object is actually controlling the `ConferenceClientModule`'s control flow as already described above. An instance of `BlockableContainerReceiver` is responsible from blocking or releasing the communication of the system under test. Finally, the instance of `ConferenceClientModule` realizing the system under test is shown.

In that figure, a snapshot of a currently running simulation of the system under test and the system's context is shown. Therefore, all initialization calls are left apart. Starting at the system under test on the rightmost side of that figure, this instance is calling `void reached();` inside its call to `ModuleState::MODULE_STATE getModuleState();` indicating the completion of one computing cycle. The returning of this call is blocked by `RunModuleBreakpoint` to interrupt the system under test and to execute other tasks. Instead, the control flow returns to `RuntimeControl` which increments the `Clock` using the computed maximum possible time step.

Afterwards, the simulation of the system's context and the system under test starts its next cycle for the new time. Thus, `RuntimeControl` checks if the

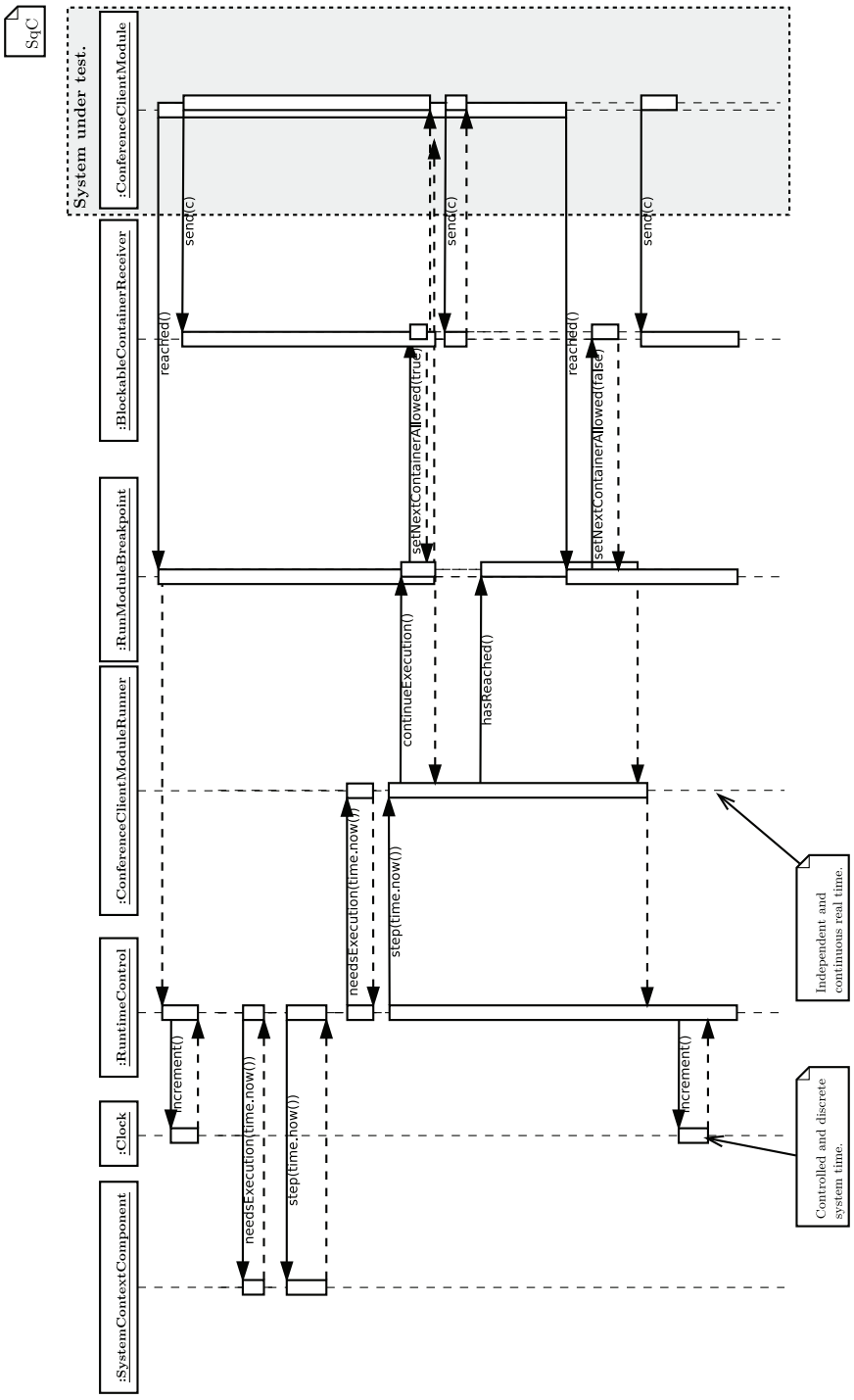


Figure 6.4: Sequence chart showing the messages sent between controlling and controlled objects: The controlled system time is encapsulated in the instance of the class `Clock` while the lifelines of the UML sequence chart represent the actually consumed real time which is required for managing the scheduling of the SUD, its communication, and the system time.

`SystemContextComponent` needs to be executed and actually executes it when necessary. Since all calls to any `SystemContextComponent` are synchronous, the control flow returns right after completing the computation of the `SystemContextComponent`.

Following the `SystemContextComponents`, `RuntimeControl` checks if the `ConferenceClientModule` wrapped into `ConferenceClientModuleRunner` needs to be executed. When the system under test must be executed for the current time as shown in the example, `RuntimeControl` calls `void step(const core::wrapper::Time&);` providing the current valid system's time to `ConferenceClientModuleRunner`.

Before returning the call `void continueExecution();` passed to `RunModuleBreakpoint`, the latter instance releases the blocked communication letting any blocked `void send(Container&);` calls finish the requested sending operation using the actual system's time and return to their callees as shown in the figure. Furthermore, the blocked call to `void reached();` is released again to continue the computation in `ConferenceClientModule`. Moreover, the system under test can send further `Containers` without being blocked.

`ConferenceClientModuleRunner` is waiting for the next call to `void reached();` by checking periodically the state of `RunModuleBreakpoint`. If the system under test should not return within the required time out, `ConferenceClientModuleRunner` throws an exception which is caught by `RuntimeControl`. Thus notifying `RuntimeControl` of a malfunctioning system under test, this class can safely release all blocked threads and communications to end its execution and to report this error to its outer scope.

Otherwise, `RunModuleBreakpoint` gets notified about the next completed computing cycle by a call to `void reached();` initiated by `ConferenceClientModule`. Followed by a call to `void setNextContainerAllowed(false);` to `BlockableReceiver`, the further sending of `Containers` is delayed to the next computing cycle and the periodic checking of `ConferenceClientModuleRunner` returns the control flow to `RuntimeControl`. This instance in turn increments the `Clock` and starts over the next cycle in the simulation for the system's context and system under test.

6.3.3 Remarks

The virtualized system time is constant for one complete cycle both in a system under test as well as in any `SystemContextComponents`, because the overall system time is incremented right after a step in the system's context followed by a step in all system's under test. Thus, the communication happens actually in *no time*, i.e. any `Container` sent from a system under test to a `ContainerConference` or from a `SystemContextComponent` to all systems under test consume no time between sending and receiving when abstracting away from currently blocked concurrently sending threads in the system under test. On one hand, the order of sent data is deterministic for a running system simulation of single-threaded applications; this is an advantage because the evaluation of such a system is reliable and repeatable. On the other hand, an evaluation of the SUD's reactions on an increasing system load [146] is not possible. However, since all communication is routed using `ControlledContainerConferenceFactory`, that class could artificially delay any distributed `Container` according to an identified latency model like evaluated in Section 8.1.2.2 as well as implement load- and payload-dependent behavior for a UDP multi-cast conference by dropping or reordering some `Containers` and thus realizing a noise model for the communication.

Additionally, `RuntimeControl` can evaluate whether an application can fulfill the required frequency. Thus, it simply measures the time required for completing one cycle between releasing the application's thread from the `Breakpoint` until it reaches it again. Extrapolating the consumed time regarding the desired frequency, an estimation about the required computing power can be made which is obviously machine-dependent.

Following, the actually needed simulation time which is consumed during one step in the `ConferenceClientModule` is equal to the real consumed system time. This means that a controlled `ConferenceClientModule` actually delays its current available time slice by using a call like `void sleep(const uint32_t&);`. However, `RuntimeControl` will cancel the current execution due to missing the predefined maximum execution time. Thus, any `ConferenceClientModule` which needs to actively delay its execution should rather map the desired waiting time to several sequential calls to its main loop considering its defined frequency.

6.4 Providing an SUD-dependent System Context: `libvehiclecontext`

In this section, algorithms are outlined for providing a system's context which depends on an SUD and its actions and reactions. These algorithms can be used to generate input data which is fed into the data processing chain of a sensor- and actuator-based autonomous system. The main purpose for the algorithms is to support the development and the unattended system simulations for AGVs; however, some of the algorithms which are discussed in the following are not limited to AGVs only but may be applied within other contexts as well.

The algorithms are implemented in the library `libvehiclecontext`. On one hand, this library can be used to realize unattended system simulation by reusing components from `libcontext` as already outlined in Figure 6.1; on the other hand, the same library and thus the same components can be used unmodified to provide interactive simulations which can be used by a developer during the software development.

6.4.1 Unattended System Simulations using `libvehiclecontext`

As mentioned before, `libvehiclecontext` bundles all SUD-related algorithms. For AGVs, these algorithms include a model for a position provider to imitate the system's behavior of an IMU which is described in greater detail in in Section 6.4.3. Besides the position, velocity, or acceleration for the AGV, further detailed information of its surroundings for example other vehicles or obstacles are required. As already outlined in Section 2.3 for example, several different sensors which perceive the AGV's surroundings are necessary. Therefore, `libvehiclecontext` provides sensor models for a color monocular camera which is described in Section 6.4.4. Furthermore, a model of an actively scanning sensor on the example of a single layer laser scanner is included. This model is described in Section 6.4.6. Both models provide so-called low-level input data because all data gathered from the sensors is raw data which must be processed to get information from the AGV's system's context. To provide a more high-level data producing model instead, a simple sensor data fusion which produces contours to abstract from perceived objects is included. This model is described in Section 6.4.7.

All aforementioned algorithms are scheduled as `SystemFeedbackComponents` from `libcontext`. Thus, they can easily be used in an arbitrary combination in unattended system simulations. However, these system simulations abstract from the real time as outlined in Section 3.2.4; moreover, due to the intention to be executed with-

out being supervised by developers to realize unattended system simulations, they cannot be directly used in an interactive manner. Therefore, an extension to the concept of `libvehiclecontext` is necessary which is described in Section 6.4.2.

6.4.2 Interactive System Simulations using `libvehiclecontext`

As mentioned before, `libvehiclecontext` is intended to be used together with `libcontext` to carry out unattended system simulations. However, during the actual software development, interactive simulations to evaluate an algorithm's behavior directly by the developer is often necessary. Therefore, an extension to this library must be provided.

Besides the aforementioned algorithms to model various aspects of an AGV, a wrapping application called `vehiclecontext` is included which allows direct and interactive execution during the development. To realize this interactive execution, the purely virtualized system time, which is necessary for unattended system simulations which are executed under the supervision and control of `libcontext` as outlined in Section 6.3, must be mapped to the real system time. This is necessary to allow the combined usage of `libvehiclecontext` which depends on `libcontext` and other applications which are not under the control of `libcontext`. To achieve this mapping for the simulation time onto the real time, the consumed time t_{consumed} for carrying out one step in the configured system's context by calling its `SystemContextComponents` must be subtracted from the calculated nominal constant duration of one time slice which bases on all defined frequencies as outlined in Section 6.2. Thus, `libvehiclecontext` can easily be used alongside with other independently running application.

Furthermore, `vehiclecontext` can be executed several times for one specific ClientConference. Thus, different instances of `libvehiclecontext` wrapped by `vehiclecontext` can be executed in parallel each with different configurations. Hence, interactive system simulations can be distributed using several independent computing nodes to spread the overall computation load.

6.4.3 Position Provider: Bicycle Model

In the following, the model used for modeling and simulating a vehicle is provided. First, its geometrical relations are explained, followed by its limitations.

In Figure 6.5, the so-called *bicycle model* and its geometrical relations are shown. Its name bases on the assumption that the wheels on the front axle and the wheels on the rear

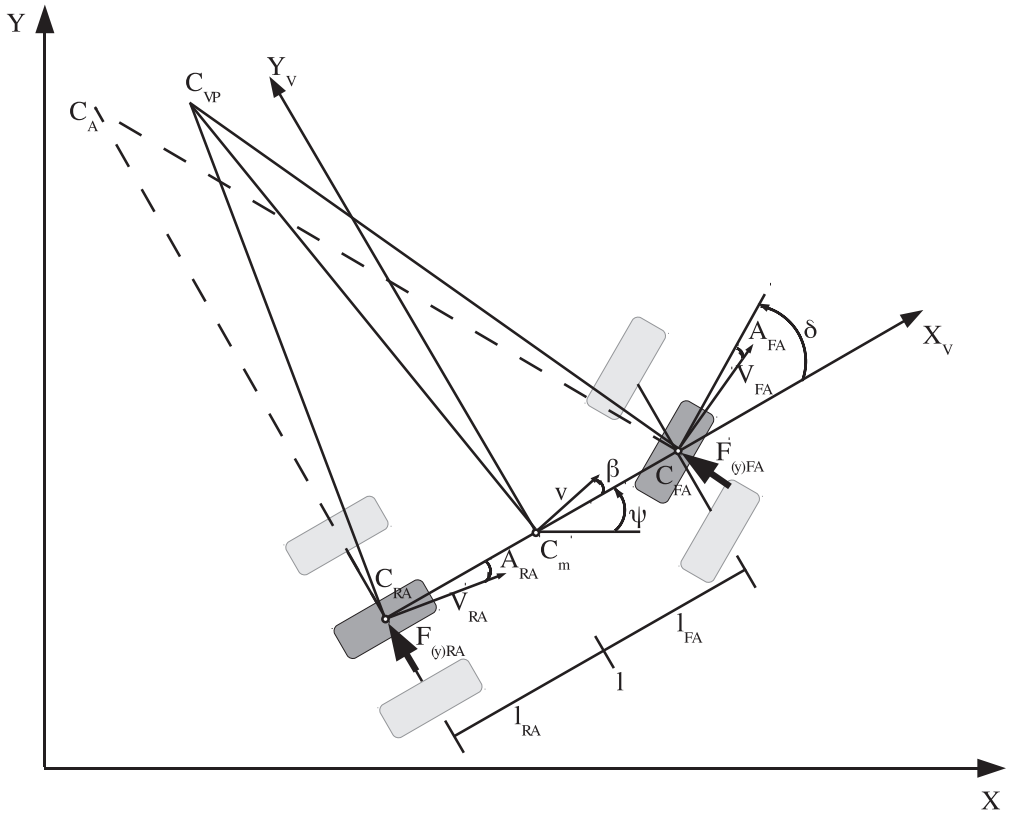


Figure 6.5: Geometry of the bicycle model (based on [126]).

axle are united comparable to a bicycle. Furthermore, the following assumptions are met [78, 104, 124]:

- The bicycle model is a two-dimensional representation of driving dynamics.
- The vehicle's center of mass is located on ground level. Therefore, vehicle rolling and pitching are not considered causing dynamic wheel loads.
- The wheels are united per axle in the vehicle's centered longitudinal axis.
- Furthermore, no longitudinal acceleration and thus a constant velocity is applied at the moment of observation.
- The cornering forces to the wheels are linearized.
- Wheel's casters and righting moments are not considered.
- Also, the wheel's tangential forces are not considered.

After defining the assumptions met about the bicycle model to get a linearized geometrical model, all symbols and relations for the vehicle's dynamic are described in the following. They are based on the principle of linear momentum and the principle of angular momentum.

C_m	describes the center of mass assumed to be located at ground level.
C_{FA}, C_{RA}	are the centers of the front and rear axles, respectively.
l	is the total distance between C_{FA} and C_{RA}.
l_{FA}, l_{RA}	are l's partial distances between C_m and C_{FA} or C_{RA}, respectively.
X_V, Y_V	describe the vehicle's own coordinate system according to [40].
ψ	is the vehicle's rotation related to the world's coordinate system.
δ	is the steering angle at front wheel.
β	describes the vehicle's attitude angle.
α_{FA}, α_{RA}	are the slip angles of the front and rear wheels, respectively.
v	is the vehicle's velocity.
v_{FA}, v_{RA}	are the velocities for the front and rear wheels, respectively.
$F_{(y)FA}$, $F_{(y)RA}$	are the cornering forces for the front and rear wheels, respectively.
C_A	is the ideal velocity pole for a slip angle free drive.
C_{VP}	is the velocity pole considering a slip angle during a drive.

The following symbols are not shown in Figure 6.5.

m	describes the vehicle's mass.
J_Z	is the vehicle's moment of inertia.
$\dot{\psi}$	is the yaw rate.
$c_{(\alpha_{FA})FW}$	is the front wheel's skew stiffness for α_{FA}.
$c_{(\alpha_{RA})RW}$	is the rear wheel's skew stiffness for α_{RA}.

In the following, geometrical relations between the identified values are defined.

$$\begin{aligned}
\alpha_{FA} &= \delta - \arctan\left(\frac{l_{FA}\dot{\psi} - v\sin(\beta)}{v\cos(\beta)}\right) && (6.1)\\
&\underset{\alpha_{FA} \ll 4^\circ}{\approx} \delta - \beta - l_{FA}\frac{\dot{\psi}}{v}.\\
\alpha_{RA} &= -\arctan\left(\frac{v\sin(\beta) - l_{RA}\dot{\psi}}{v\cos(\beta)}\right) && (6.2)\\
&\underset{\alpha_{RA} \ll 4^\circ}{\approx} -\beta + l_{RA}\frac{\dot{\psi}}{v}.\\
\beta &= \psi - \arctan\left(\frac{\dot{y}}{\dot{x}}\right). && (6.3)
\end{aligned}$$

The next equations describe the dynamic relations as a state space representation.

$$\begin{aligned}
\dot{\mathbf{x}} &= \mathbf{A}\cdot\mathbf{x} + \mathbf{b}\cdot u && (6.4)\\
\mathbf{y} &= \mathbf{c}^T\cdot\mathbf{x} + d\cdot u.
\end{aligned}$$

In Equation 6.4, the state space model which describes the dynamic driving behavior according to [136] is shown. **A** describes the system's state matrix, **b** denotes the system's input matrix and **c** describes the system's output matrix. **d** is used to describe the feed forward matrix. The vector **x** describes the system's current state vector and u the system's input which reflects the steering angle in this case.

$$\begin{aligned}
\mathbf{x} &= \begin{pmatrix}\beta\\ \dot{\psi}\end{pmatrix} && (6.5)\\
\dot{\mathbf{x}} &= \begin{pmatrix} -\frac{c_{(\alpha_{FA})}+c_{(\alpha_{RA})}}{mv} & \frac{c_{(\alpha_{RA})}l_{RA}-c_{(\alpha_{FA})}l_{FA}}{mv^2} - 1\\ \frac{c_{(\alpha_{RA})}l_{RA}-c_{(\alpha_{FA})}l_{FA}}{J_Z} & -\frac{c_{(\alpha_{RA})}l_{RA}^2+c_{(\alpha_{FA})}l_{FA}^2}{J_Z v}\end{pmatrix}\cdot\mathbf{x}\cdot\begin{pmatrix}\frac{c_{(\alpha_{FA})}}{mv}\\ \frac{c_{(\alpha_{FA})}l_{FA}}{J_Z}\end{pmatrix}\cdot u.
\end{aligned}$$

In Equation 6.5, the input vector for the system and the state space representation for the aforementioned geometrical relation for the driving dynamics are shown. As already mentioned, β describes the vehicle's attitude angle and $\dot{\psi}$ describes the yaw rate. Using the aforementioned model and equations, the vehicle's state can be numerically approximated

according to [136] which is shown in Equation 6.6.

$$
\begin{aligned}
\mathbf{x}(t_{(k+1)}) &= \Phi(\Delta t) \cdot \mathbf{x}(t_k) + \Gamma(\Delta t) \cdot u(t_k) \\
\mathbf{y}(t_k) &= \mathbf{c}^T \cdot \mathbf{x}(t_k) + d \cdot u(t_k) \\
&\Leftrightarrow \\
\mathbf{x}(t_{(k+1)}) &= \sum_{k=0}^{\infty} \frac{(\mathbf{A}\Delta t)^k}{k!} \cdot \mathbf{x}(t_k) \sum_{k=0}^{\infty} (\frac{\mathbf{A}^k (\Delta t)^k}{(k+1)!} \cdot (\Delta t)\mathbf{B}) \cdot u(t_k) \\
\mathbf{y}(t_k) &= \mathbf{c}^T \cdot \mathbf{x}(t_k) + d \cdot u(t_k).
\end{aligned}
\tag{6.6}
$$

The accuracy of the numerical approximation depends obviously on the number of chosen summands for the individual sums. Regarding the initially met assumptions, the resulting matrices from these sums are constant because they depend only on the constant time step Δt, the chosen accuracy which defines the upper limit K for the sum's index k and the current values of the state and input matrix. Thus, these numerical approximated matrices must be updated whenever the velocity v changes.

However, due to the simplifications which are met by these assumptions, the bicycle model for describing a vehicle's driving dynamic is only valid for lateral accelerations less than $0.4\frac{m}{s^2}$ [104]. For getting more precise results especially in limit ranges for the vehicle dynamics, the linearized bicycle model must be modified using non-linear equations describing the forces which are applied to the wheels on the front and rear axles [6]. Due to the intended application as outlined in Chapter 8, this more accurate model is not regarded but the described bicycle model for the position provider is chosen to show the applicability of the concepts described in this thesis.

$$
\dot{x}_{\text{pos}} = \begin{pmatrix} v\cos(\psi - \beta) \\ v\sin(\psi - \beta) \end{pmatrix}. \tag{6.7}
$$

To update the position data for the simulation, the changing of the vehicle's position is calculated as shown in Equation 6.7. Using a given initial position x_0 this changing can be integrated to provide absolute position information.

6.4.4 Monocular Camera Provider

The first raw data provider realized in the simulation of the system's context is a monocular camera provider. Its software architecture is shown in Figure 6.6.

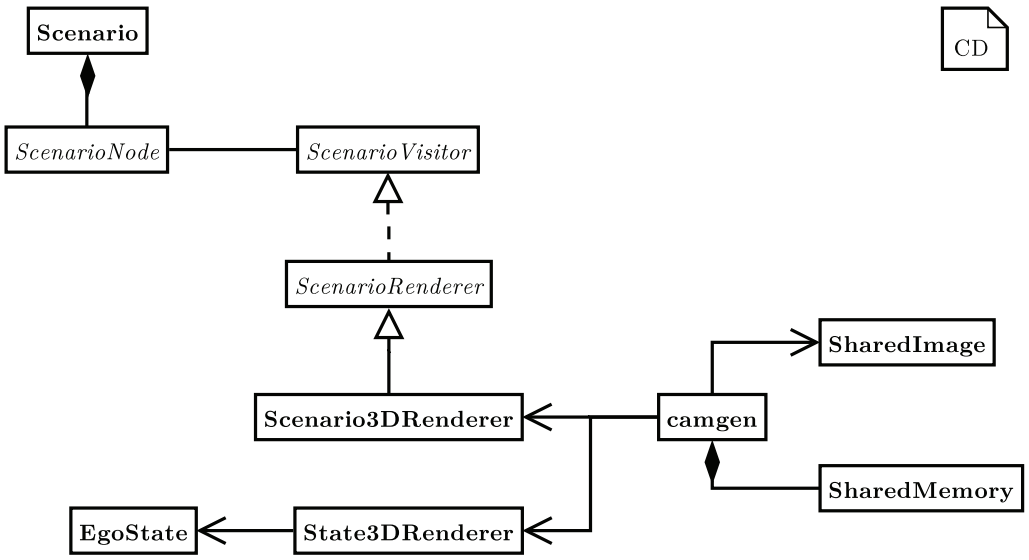


Figure 6.6: Software architecture for the monocular camera provider. This provider implements a simple monocular camera sensor model by using the specified system's context which is described by `Scenario` to render an OpenGL context. From this context, an image dump is provided for user-supplied image processing algorithms using a shared memory segment.

The realization of the monocular camera provider is quite easy reusing some concepts provided by `libcore` and `libhesperia`. Its main principle is to use the current state of the system's context for rendering into a predefined field of view for the virtual monocular camera. Since the `Scenario3DRenderer` can simply be used for generating an OpenGL render-able scene, the `State3DRenderer` renders dynamic elements from the system's context like the position and orientation of the AGV itself. These information are necessary because the virtual monocular camera is "mounted" on the AGV itself for example. Therefore, OpenGL's scene view camera is placed where the AGV is currently located and is translated to its virtual mounting position.

The OpenGL's scene view camera is used to grab the actual view of the scene into a `SharedMemory` segment. This shared memory is used to share the image's data between independent processes without serializing and sending the data using sockets for examples. However, shared memory can only be used among processes on the same computer. Regarding the pipes and filters data processing chain mentioned above, it is

assumed that basic feature detection is implemented on the computer to which the camera is attached to.

For informing other processes about the shared memory segment, the data structure `SharedImage` is used. This datum contains information about the image's dimensions and color depth as well as the name of the shared memory to which other process should attach to read the image. This data structure containing an image's meta-information is simply sent to the ClientConference.

Since the virtual monocular camera would grab the images as fast as possible, the mutual exclusion concept of shared memory segment can simply be used by the feature detection process to control the image acquisition. Therefore, the feature detecting process simply locks the shared memory and thus prevents the virtual monocular camera to grab new images while detecting features in the current frame is still incomplete.

6.4.5 Stereo Camera Provider

Based on the aforementioned monocular camera provider, a simple stereo camera provider can by realized easily. Therefore, two virtual monocular cameras are set up to operate as a combined stereo camera system. Thus, the rendering algorithm for the OpenGL scene is split up into two independent rendering cycles. For the example of an AGV, the first pass reads the current position and orientation of the vehicle to determine its absolute position in its system's context. Afterwards, the first monocular camera of the stereo camera pair is positioned according to its specified mounting position relative to the AGV's current position and the scene is rendered into a `SharedMemory` segment which is twice the size of the size of one single monocular camera image. Followed by the second pass, the second monocular camera is positioned alike and its current view into the OpenGL scene is rendered into the second half of the `SharedMemory` segment. Finally, the two combined images are broadcasted using a `SharedImage` in an analog manner comparable to the monocular camera provider.

6.4.6 Single Layer Laser Scanner Provider

In the following, the raw data generation for a single layer laser scanner using a GPU is described. Due to the use of a GPU and as shown in Figure 6.7, a concept using the three-dimensional representation for the scenario to be rendered with OpenGL like the aforementioned camera provider is necessary to produce data for a single layer laser scanner as described in the following.

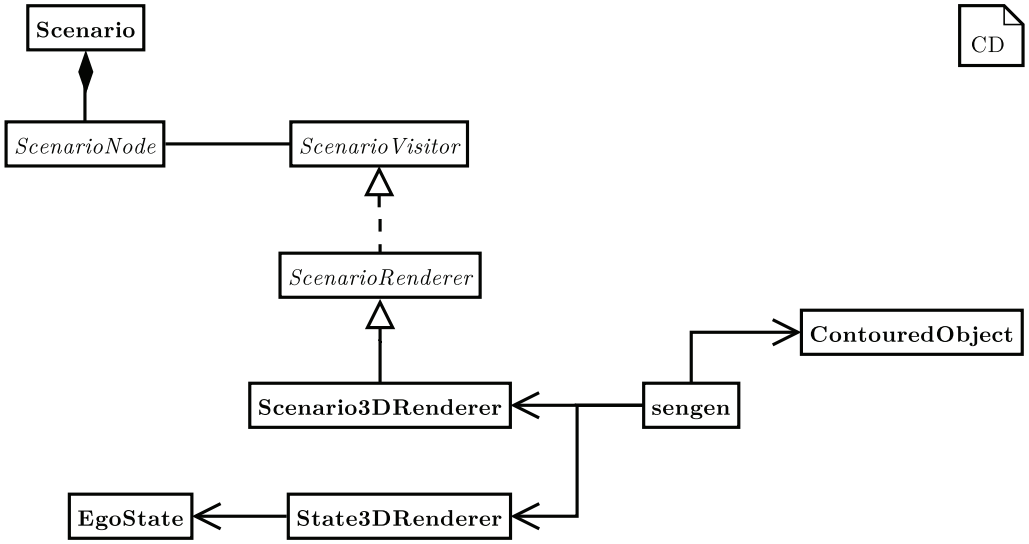
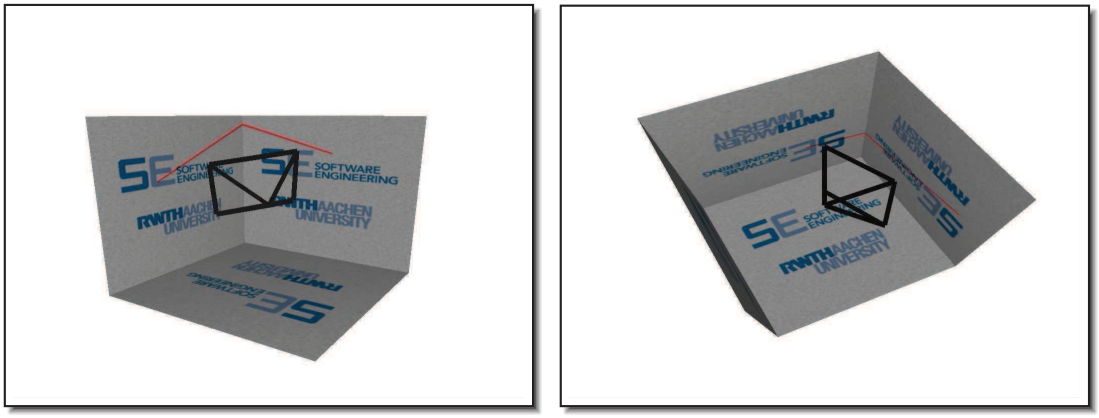


Figure 6.7: Software architecture for the single layer laser scanner provider. Comparable to the aforementioned sensor model for a camera, this provider also bases on the specified system's context described by `Scenario` for rendering an OpenGL context. However, contrary the aforementioned camera provider, this context is modified by a special shader program which is executed on a GPU which generates distance information. In a pre-processing stage, this context is evaluated by an image analyzing algorithm to retrieve these distances to providing them to user-supplied applications on higher layers.

6.4.6.1 GPU-based Generation of Synthetic Data

The main idea behind the raw data generation for a single layer laser scanner is the use of projective textures on a modern GPU. The principle behind projective textures is depicted in Figure 6.8. In the figure on the left hand side, a red line is shown in the upper half of a cube, while the picture on the right hand side shows the same line in the lower half of the cube after moving it down; both lines are projected into the scene. The projector is indicated by the dark lines.

Their basis are *projectors* for projecting a predefined texture into the rendered scene. The texture is placed right in front of the projector using the transformation T as shown in Equation 6.8. P_p is the projection matrix and V_p describes the view matrix; V_e^{-1} describes the inverse of the camera matrix to get back into the world's coordinate system to get finally the texture coordinates for projection.



(a) Projector shining in the upper half of the cube.

(b) Projector shining in the lower half of the cube.

Figure 6.8: Principle of projective textures using a projector (based on [18, 50]): A special texture is defined used as a foil which is placed directly in front of the camera's position from which the scene should be rendered. Using the equation specified in Equation 6.8, the content of this texture is projected into the scene. For simulating a single layer laser scanner, this texture contains a single line as shown in this figure which is projected into the scene.

$$T = \begin{pmatrix} \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot P_p \cdot V_p \cdot V_e^{-1} \tag{6.8}$$

Using this transformation, several independent projectors can be defined in the scene. After defining the principles behind projective textures, the algorithm for computing distances from a projected texture is outlined in Figure 6.9 which consists of an image synthesis followed by an image analysis problem. The former is marked with dark gray, while the latter is marked with light gray.

6.4.6.2 Algorithm for Computing Synthetic Raw Data

First, the shader programs are loaded into the GPU. These programs are used to implement the actual projection for a texture on one hand, and to determine the distances of the scene's vertices to the viewing position on the other hand. The program is split into one associated with the *Vertex Processor* and one associated with the *Fragment Processor*. An

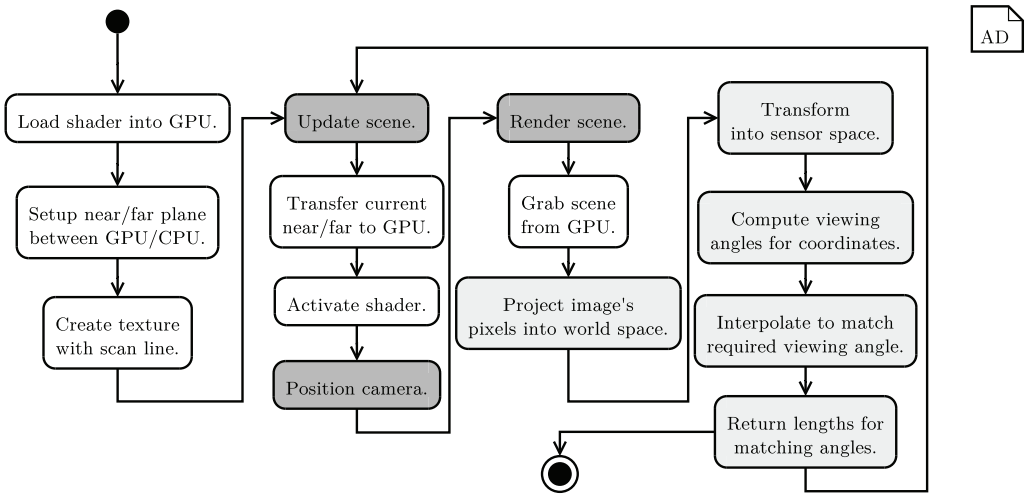


Figure 6.9: Outline for algorithm to compute distances using projected textures.

overview of the OpenGL processing pipeline can be found at [90].

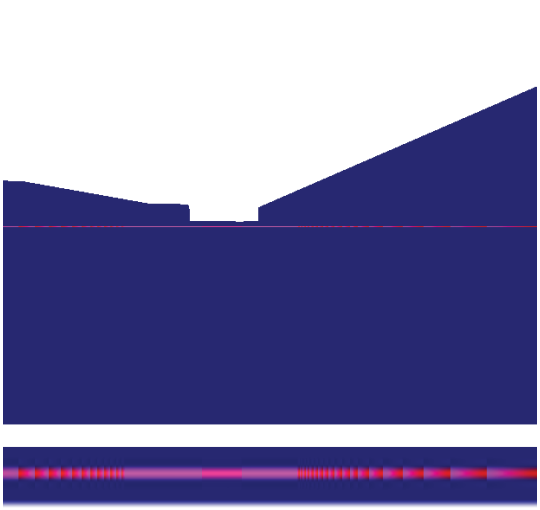
The former program encodes a vertex' depth value into a provided texture. The latter program reads the actual pixel's color representing its distance to the viewing position, computes the z-buffer value using the actual valid near- and far-clipping-planes, and sets the pixel in the resulting image according to the z-buffer value as the pixel's R-channel, the real distance's integral value as the pixel's B-channel, and the fractional value as the pixel's G-channel after texture projection.

After initializing the GPU with the vertex- and pixel-shaders, two variables of the fragment shader containing the current values for the near- and far-clipping-planes are connected to the CPU program. Next, the texture to be projected must be defined. As already shown in Figure 6.8, this texture contains the *scan line* to be projected into the scene since the fragment shader modifies the resulting image only if a scan line was found marked by the texture pixel's value in the R-channel which is set to 255.

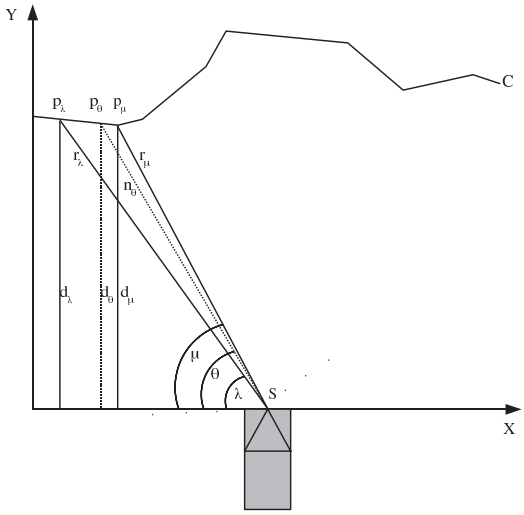
Next, the repeatedly executed distance computation loop is started. The first step is to update the current OpenGL scene by manipulating its elements according to the current state of the system's context. Following, the current valid values for the near- and far-clipping-planes are transferred to the fragment shader using the previously connected variables and the shader program is activated on the GPU for the current rendering cycle. The phase for the image syntheses is completed by positioning the camera using quaternions to the current position and rotation retrieved from the recently received `EgoState` for rendering the actual scene's elements.

After the rendering is completed, the image analysis phase starts with reading the com-

puted image from the GPU. The resulting image produced by the GPU is shown in Figure 6.10a, where in the upper half all surroundings' elements which are not irradiated by the projector are simply colored in blue. In the center of the image are the resulting pixels computed by the shader program. For the sake of clarity, the resulting pixels are magnified below the image.



(a) Resulting image read from GPU after applying the shader program: In the image above, the resulting scene which is read back from the GPU is shown; the lower part magnifies the relevant part from the image which contains the distances encoded by the shader program.



(b) Viewing angles and distances: Due to the discretization, only the rays r_λ and r_μ can be calculated; however, the nominal ray n_θ is required which must be calculated.

Figure 6.10: Visualization for the output of the sensor model for a single layer laser scanner. This sensor model is realized in an application which uses a shader program on the GPU for calculating the distances.

For illustration, sensor's raw data for producing distances for a single layer laser scanner with a FOV of $\frac{\pi}{2}$ containing 91 rays from $-\frac{\pi}{4}$ to $\frac{\pi}{4}$, whose mathematical model is depicted in Figure 6.10b should be realized. The mounting position S of the scanner is located in the center of the image. The arbitrary line C denotes distances to the viewing position for the surroundings irradiated by the scanner. Therefore, this line contains all intersection points from the single layer scanning plane with the surroundings.

Due to the discretization in the resulting image, only the distances d_λ and d_μ to the viewing plane denoted by the X-axis could be computed for the imaginary rays r_λ and r_μ for example. However, the nominal ray n_θ reflecting one of the scanner's rays must be computed. The distances returned to the viewing plane denoted by d_λ and d_μ are encoded in the resulting image's pixels. The pixel's R-channel containing the depth value for a

scene's vertex is used to compute the world coordinates for the given x- and y-position in the image coordinate system using Equation 6.9.

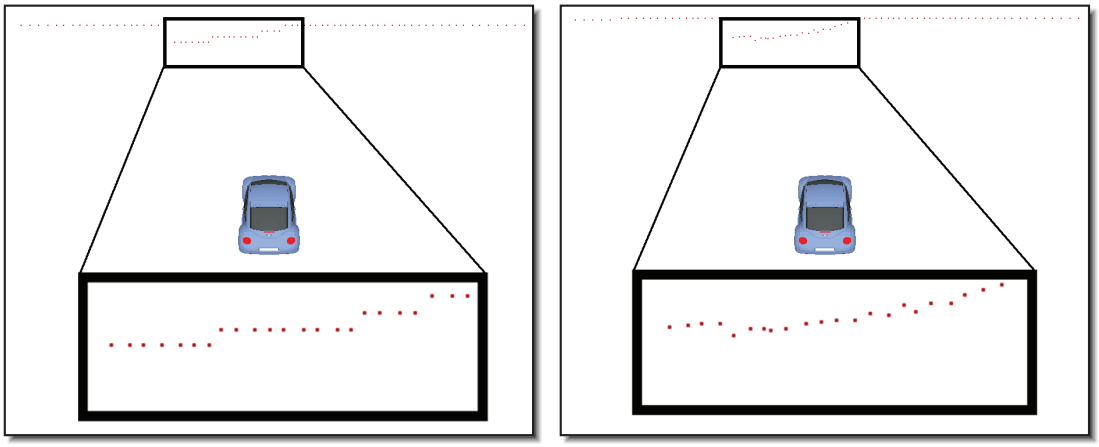
$$C = (P \cdot M)^{-1} \cdot \begin{pmatrix} \frac{2(x-\hat{x})}{\overline{w}} - 1 \\ \frac{2(y-\hat{y})}{\overline{h}} - 1 \\ 2z - 1 \\ 1 \end{pmatrix} \tag{6.9}$$

The product of $P \cdot M$ denotes the projection in world coordinates and the triple (x, y, z) contains the image's x- and y-position and also the computed depth value as z. The quadruple $(\hat{x}, \hat{y}, \overline{w}, \overline{h})$ contains information about the image's width and height as well as the offset of the image's logical origin $(0, 0)$ regarding to the screen's origin. Using this equation, the positions p_λ and p_μ can be computed directly.

However, since p_θ could not be measured directly, an interpolation for the desired nominal ray n_θ is necessary. Therefore, all angles along the viewing plane using the points' distance to the viewing plane and its lateral distance to the sensor's mounting position S are computed using $\phi_{(x)} = \arctan(\frac{d_{(x)}}{S_x - x})$. All computed angles $\phi_{(x)}$ are stored with their corresponding distance $d_{(x)}$ in a map. Due to the dependency on the depth value from the near- and far-clipping-planes used for computing the world coordinates causing aliasing effects, the vertex's real distance is encoded in the pixel's B- and G-channel. This value is finally used to optimize the computed and interpolated distance for the nominal ray n_θ as shown in Figure 6.11.

To compute the correct distances for the nominal rays, the map is used the find the both best matching rays denoted by r_λ and r_μ around the nominal ray n_θ minimizing the difference between the angles (θ, λ) and (θ, μ). The computed difference for both pairs is furthermore used to weight the interpolated distance d_θ using d_λ and d_μ as shown in Equation 6.10.

$$d_\theta = \frac{\parallel \theta - \lambda \parallel}{\parallel \lambda - \mu \parallel} d_\lambda + \frac{\parallel \theta - \mu \parallel}{\parallel \lambda - \mu \parallel} d_\mu \tag{6.10}$$



(a) Computed distances using the z-buffer depth value scaled regarding to near- and far-plane.

(b) Optimized distances using the actual distances from w-buffer.

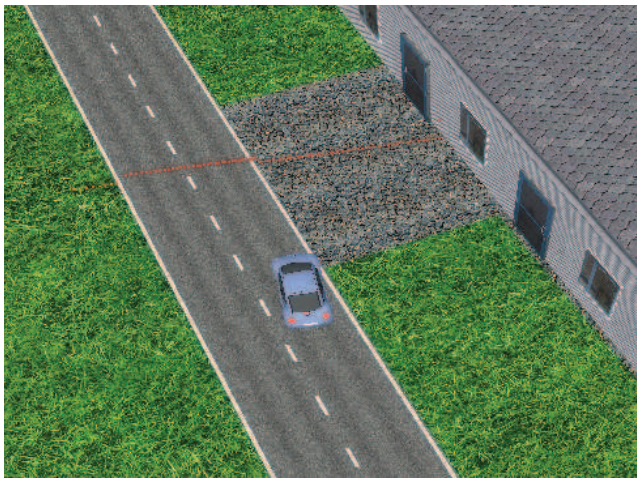
Figure 6.11: Aliasing effect when solely using the z-buffer demonstrated at a curved target: On the left hand side, the rays from the laser scanner are hitting the target with only discrete distances which provides inaccurate distances. This is caused by the decreasing accuracy for increasing distances to the ray-emitting source [32]. On the right hand side, the z-buffer combined with the w-buffer is used to optimize the calculated distances which reduces the aliasing effect.

6.4.6.3 Example

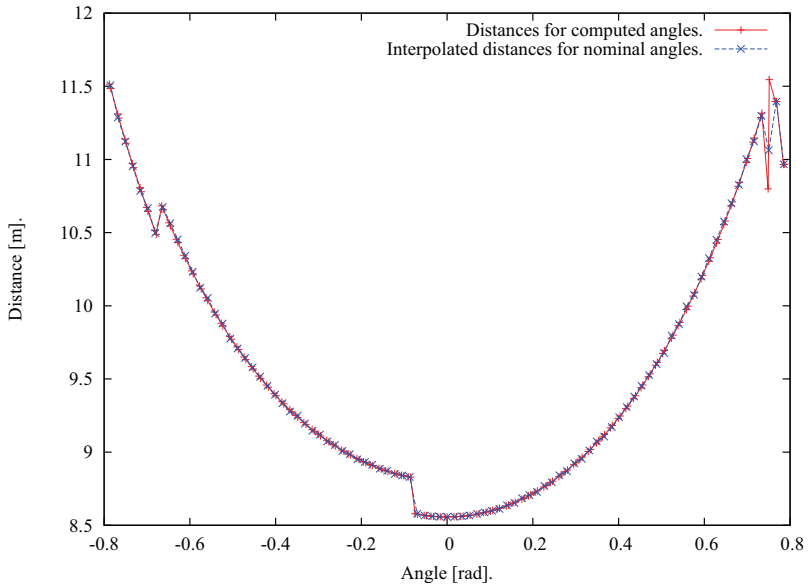
In Figure 6.12a, a situation in the modeled surroundings is shown. On the right hand side in Figure 6.12b, the results for a single layer laser scanner with a FOV of $\frac{\pi}{2}$ mounted 1.55m above the ground on the center of the vehicle's roof looking at 8m in front of the vehicle are shown. The computed distances with correct angles are plotted in blue with an X on its graph. In red with a + on its graph, the interpolated distances to match the sensor's nominal angles are plotted. In the plot, the elevation of the road can be seen easily between $\phi = -0.65$rad and $\phi = -0.1$rad. Furthermore, the red graph describing interpolated distances matches well the blue graph containing computed distances and angles. Even at the wall of the house, the interpolation for nominal rays produces reasonable results between the actually computed distances.

6.4.7 Sensor-fusion Provider

For providing pre-processed data like output from a sensor-fusion, a mid-level data provider for generating `Obstacles` is described in the following. It is called mid-level data provider because it does not produce raw data for sensors of a specific type but can be used for results of a sensor-fusion fusing several raw data producing sensors. The



(a) Situation with activated single layer laser scanner.



(b) Plot of interpolated distances for nominal angles and computed angles for measured distances: The graph marked with + shows the computed angles while the graph marked with x depicts the reconstructed nominal angles.

Figure 6.12: Visualization of computed distances.

`Obstacle` data structure derives directly from `PointShapedObject` and can be used to describe arbitrarily shaped objects from the system's context detected by different sensors like radar or laser scanners. This datum consists of a position, rotation if applicable, acceleration, and velocity and moreover, a polygon used for describing its outer shape.

6.4.7.1 Generation of Synthetic Data

Mostly, the outer shape is only a contour because sensors normally cannot detect the rear side from objects. Since later processing stages could assume the contour is part of an arbitrarily shaped polygon and simply connect the open vertices at the beginning and end of the contour, the polygon as a subset of contours is used.

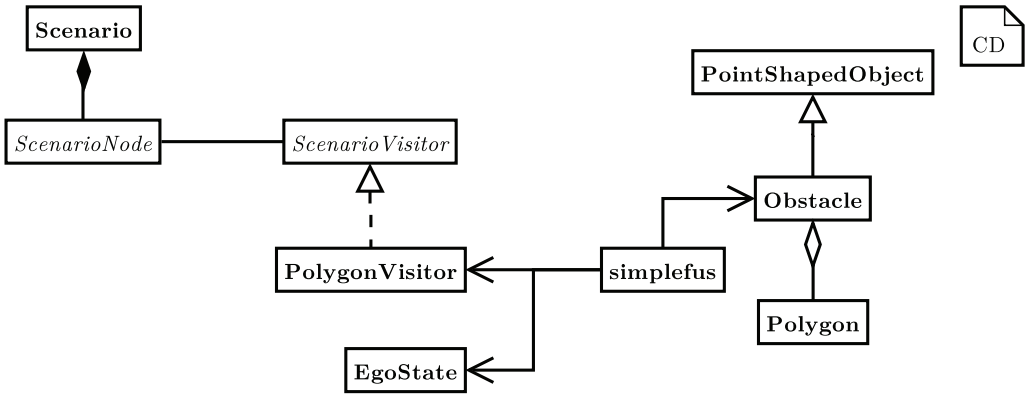


Figure 6.13: Software architecture for the sensor-fusion provider. Comparable to both aforementioned sensor providers, this one also bases on the `Scenario` specification. For generating an abstract representation from the SUD's surroundings, a visitor is defined which traverses the scenario's AST to gather information from polygons. These polygons are used to calculate intersections with a specified viewing area which represents an ideal sensor.

Like all previously described providers, the production of pre-processed data is operating on the `Scenario` using a `PolygonVisitor` for extracting all polygonal shaped objects as shown in Figure 6.13. Furthermore, it uses the `EgoState` for localization in the world.

6.4.7.2 Algorithm for Computing Synthetic Raw Data

The purpose of the following algorithm is to generate contours for the `Obstacle` data structure. First, the sensor's FOV is defined by specifying several coordinates in the vehicle's coordinate system. Next, for every update on the `EgoState`, the local FOV is translated to the current position and rotation in the world using the data provided by `EgoState`.

For every polygon p_j in the list extracted by the `PolygonVisitor`, the overlapping areas with the FOV polygon are computed. Therefore, for every edge $e_{i_{FOV}}$ of the FOV's

polygon the intersection point $I_{e_{ij}}$ with every edge of p_j is computed. Since these intersection points are on the outer line of the FOV, these points belong to the overlapping polygon by definition and are added to the polygon list P'_j for the polygon p_j. Next, all vertices v_j are tested if they lie inside the FOV. If they are inside the FOV, they are also added to P'_j.

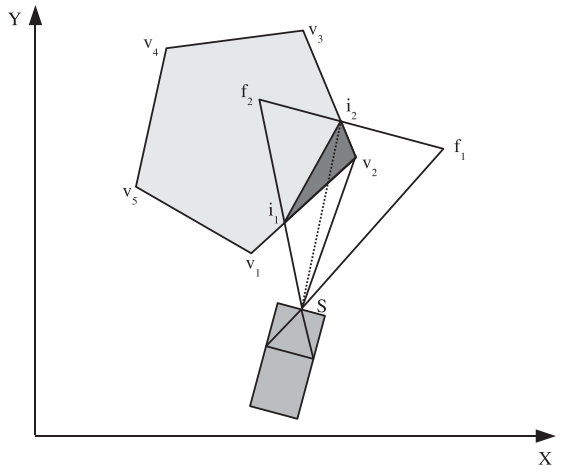
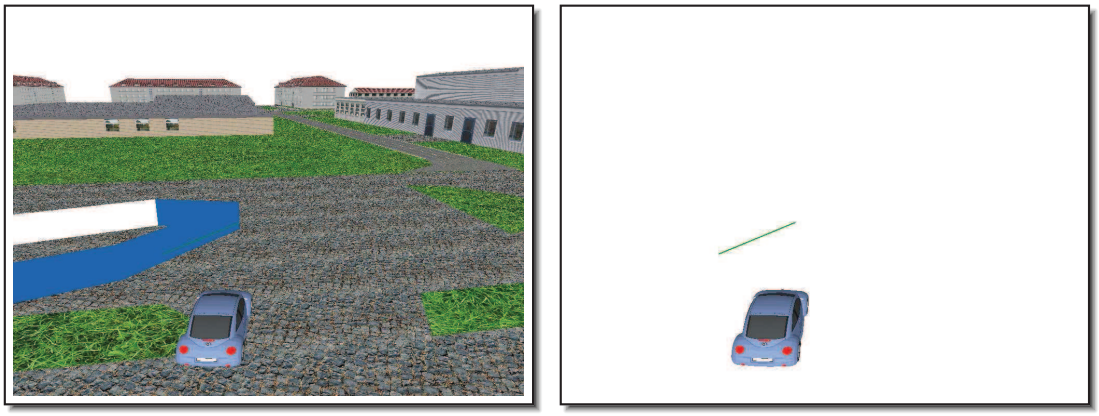


Figure 6.14: Overlapping polygons with visibility lines resulting in a contour line between i_1 and v_2: The dark gray triangle i_1, v_2, i_2 shows the invisible area within the sensor's FOV from point S for the polygon v_1, v_2, v_3, v_4, v_5. Because the line $\overline{Si_2}$ crosses this invisible area, the point i_2 is not part of the outer contour.

Now, the overlapping polygon described by the vertices P'_j was found. Next, the vertices must be reduced by the vertices which cannot be seen by the sensor's mounting position S. Therefore, for every vertex v_j from the polygon P'_j, the line $\overline{Sv_j}$ describing the visibility line from S to v_j is tested for intersection points with any other edge from P'_j. If no intersection point could be found, the vertex v_j can be seen directly from the sensor's mounting position; otherwise, this vertex is removed from P'_j. Finally, all vertices in P'_j are sorted with ascending viewing angles regarding the sensor's mounting position to ensure the correct detection order. In Figure 6.14, the previously outlined algorithm is depicted.

6.4.7.3 Example

In Figure 6.15a, results of the algorithm are shown for a FOV of 60°. As shown in Figure 6.15b while hiding the surroundings, only the measured object's contour line is shown.



(a) Visualization of polygonal obstacles.

(b) Visualization of polygonal obstacles with hidden surroundings.

Figure 6.15: Visualization for the output of the sensor-fusion provider.

6.4.8 Dynamic Context Provider

As outlined in Section 4.4, the stationary system context is augmented by dynamic objects as defined in a situation which also uses a DSL. On the example of AGVs, this dynamic context can be used to model different traffic situations e.g. with other vehicles on intersections.

In Figure 6.16, the general architecture for the dynamic context provider is shown. Comparable to the aforementioned providers, this one also uses the `Scenario` data structure which can have one or more situations defined in `Situation`. In the latter data structure, the type, shape, and behavior of a dynamic object according to the DSL as outlined in Section 4.4.2 are defined. To control a dynamic object, the `PointIDDriver` is used which realizes the dynamic object's driving on an a priori defined route consisting of several consecutive way-points provided by `Scenario`.

Every instance of a `PointIDDriver` is managed by `DynamicContextProvider` which supervises depending other objects, updates the position data, and distributes the system state of dynamic objects. On one hand, the system state can be directly used in user-contributed applications; on the other hand, it can be "detected" by one of the aforementioned providers like the camera or the single layer laser scanner provider. Therefore, the abstract data structure for a dynamic object is rendered using the concepts outlined in Section 5.4.5. Thus, information from the SUD's system's context can be provided either on low-level using sensors' raw data or on high-level directly using the dynamic object's data structure.

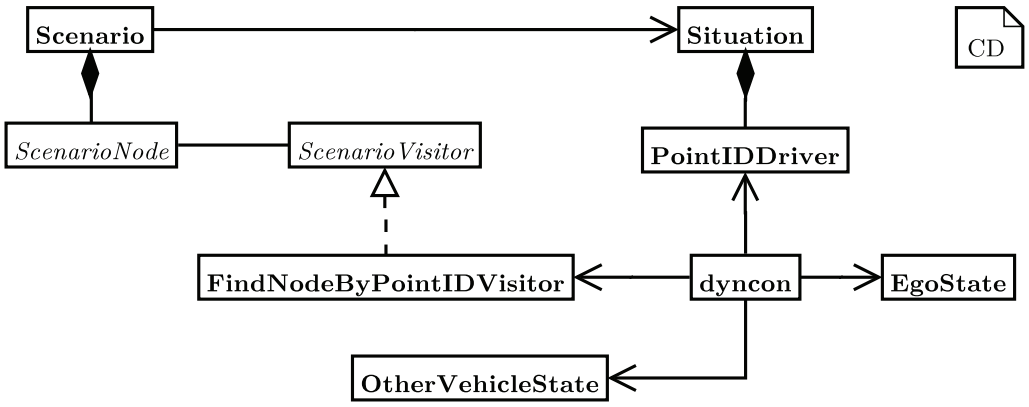


Figure 6.16: Software architecture for the dynamic context provider. Comparable to the already presented providers, this provider bases on the `Scenario` specification as well. Moreover, it uses a `Situation` to get the specification of the dynamic context. To create the necessary models for the dynamic context, a concrete instance of the DSL is evaluated and the required objects with their associated behavior `PointIDDriver` are set up. The data provider computes continuously updated information using the `OtherVehicleState` data structure. These objects can either be used directly in high-level user-contributed applications by evaluating their attributes for example or they can be rendered into an existing OpenGL scene. In the latter case comparable to the stationary surroundings, the dynamic objects can be "detected" using the aforementioned providers.

7 Monitoring and Unattended Reporting

In this chapter, a non-reactive visualization environment allowing interactive inspections of a running sensor- and actuator-based autonomous system as well as unattended reporting using the framework $\mathcal{H}$esperia are presented. First, some general considerations and design drivers are discussed. Following, some realization aspects for an interactive inspection application called *Monitor* are outlined. Finally, unattended inspections allowing automated test runs which are used in the next chapter are presented.

7.1 General Considerations and Design Drivers

Next, selected design drivers for a running system's inspection are outlined.

- *Non-reactive inspection.* Any monitoring or reporting application must not interact with a running system to avoid interferences. This implies that no running application takes note of a monitoring or reporting application's existence. Furthermore, no application must send additional data explicitly to such an application.
- *Intuitional interface.* Obviously, a monitoring application which is intended to be used by humans must provide an intuitional interface. Moreover, for the unattended usage described in Section 7.3, interfaces to be used by a test engineer must be unambiguous.
- *Transparent usage.* Any monitoring or reporting application should be used both with the real running sensor- and actuator-based autonomous system as well as with previously recorded data and with data generated by a system simulation.

Due to the design and architecture of the framework $\mathcal{H}$esperia, at least the first and the last requirement can be realized with ease. Since the entire communication between applications implemented using the framework $\mathcal{H}$esperia is realized using the concept ClientConference as outlined in Section 5.4.1, any monitoring or reporting application can simply

join the same communication group. Thus, every message exchanged in a ClientConference is automatically received by a monitoring or reporting application.

Moreover, the concept ClientConference can not only be used on live systems but also for replaying previously recorded data using the tools already mentioned in Section 5.6. Additionally, the real system can be transparently substituted by a virtualized model as described in Chapter 6 and Chapter 8. Thus, a monitoring or reporting application realized with the framework $\mathcal{H}$esperia can be used to inspect and analyze a running system.

7.2 Interactive Monitoring

As already mentioned in Section 5.4.5, the framework $\mathcal{H}$esperia provides a device-independent visualization. Thus, both two-dimensional representation and three-dimensional visual feedback can be easily realized using the concepts provided by the framework. In this section, the visualization for the stationary surroundings is outlined.

7.2.1 Monitoring Component `monitor`

In Figure 7.1 the architecture of the non-reactive interactive inspection component called *Monitor* is shown. Since it is realized using the concepts provided by the framework $\mathcal{H}$esperia, all requirements discussed at the beginning could simply be realized.

The component itself derives from `ConferenceClientModule` and implements the interface `DataStoreManager`. Using this interface, different data-stores as already outlined in Section 5.3.2.3 can be registered at a running ClientConference. The application monitor uses the `PlugInProvider` to query existing plug-ins to be used for system inspection. The `PlugInProvider` returns a list of available `PlugIns` as well as one special plug-in called `MasterPlugIn` if desired by the user.

The `MasterPlugIn` can be used to substitute the standard handler called `FIFOMultiplexer` for handling incoming `Containers`. By default, every incoming container is simply distributed to all running plug-ins using their implemented interface `ContainerListener`. By activating the `MasterPlugIn`, a buffered multiplexer allowing suspend, resume, replay, step forward, step backward, and save to disk can by applied to a running system.

In Figure 7.2, the application itself is shown. It is realized as a Multi Document Interface (*MDI*)-application using the GUI framework Qt [22]. In the window, all available

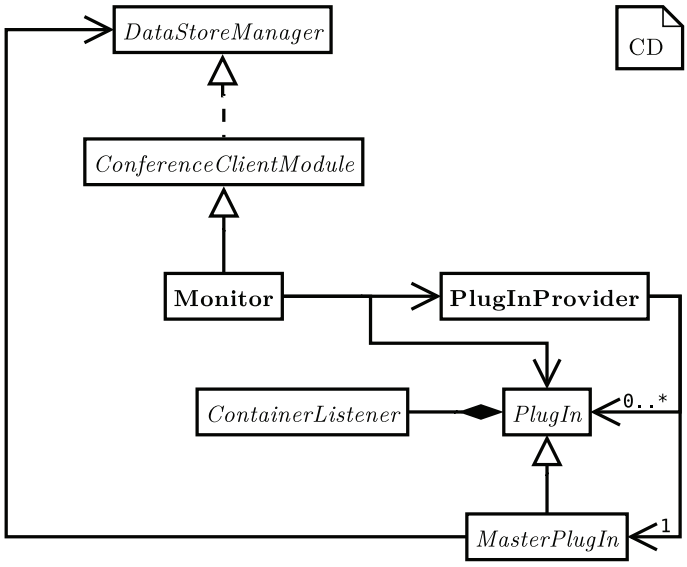


Figure 7.1: Architecture of component “Monitor”. The application consists of several independent plug-ins which are fed with incoming `Containers` automatically. Thus, they can realize arbitrary visualization tasks; furthermore, due to the plug-in concept, this application can be extended easily.

plug-ins reported by the `PlugInProvider` can be selected. Every plug-in can be executed several times if necessary. In the bottom window in area “6”, the control plug-in for the `BufferedFIFOMultiplexer` is shown allowing the user to interrupt the `Container`’s distribution to all running plug-ins while further filling the available buffer in background. Furthermore, the replay of available `Containers` in the buffer can be controlled using self-explanatory buttons. Moreover, the current content of the buffer can be selectively saved to disk in the same format as the component `recorder` would save the data. Thus, captured interesting situations during the inspection of the running system can simply be stored for further analysis or playback using `player`.

In the center window in part “1”, all currently running plug-ins are shown. In the upper left corner in area “2”, a `PacketLogViewer` showing a chronologically ordered, textual representation of all received `Containers` is activated. On the upper right corner in area “3”, a surroundings’ visualization component which is described in detail in the following is running. On the bottom left corner in area “4”, a viewer for currently available `SharedImages` which are exchanged between independent processes using shared memory is shown; these synthetic images are created by the *Camera Provider* as described in Section 6.4.4. On the bottom right corner in part “5”, statistical data of the consumption of their assigned run-time frequency of all running applications is shown.

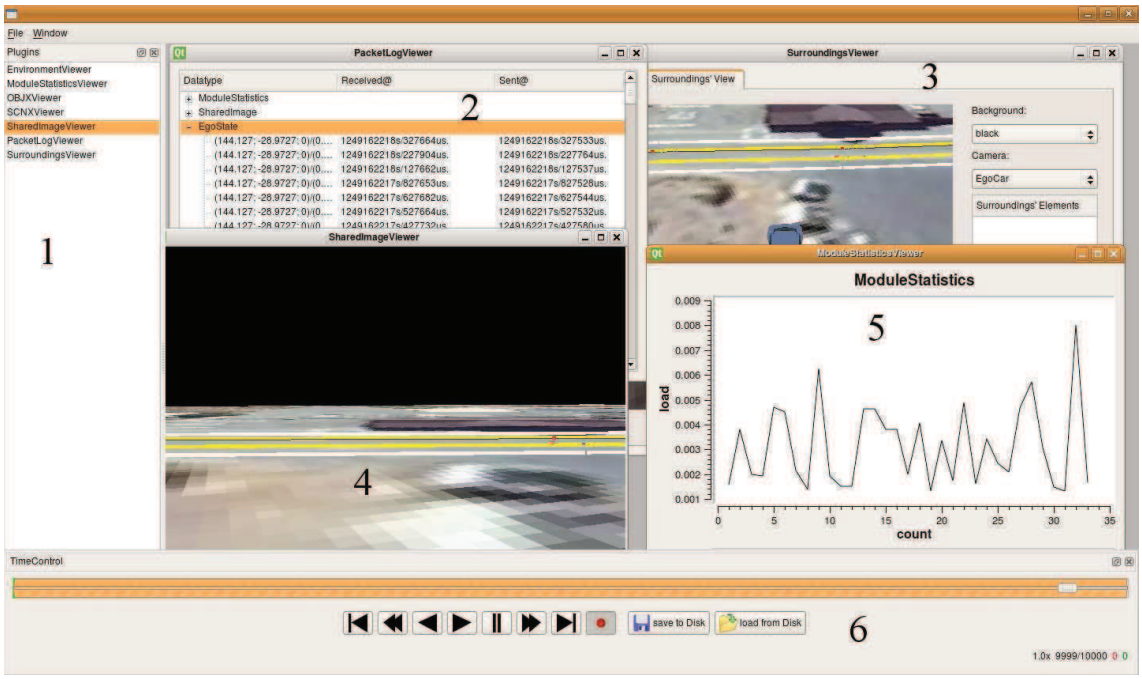


Figure 7.2: Non-reactive system inspection using component "Monitor": On the left hand side in area "1", a list of all available plug-ins is shown. In the upper left hand side in part "2", a trace of all received `Containers` is shown while on the upper right hand side marked with "3", a freely navigatable 3D visualization of the current scenario is rendered. On the lower left hand side in area "4", the visualization of the camera provider producing synthetic images is shown. The lower right hand side in part "5" finally plots statistical information about the applications' life-cycles. In area "6", a control bar for controlling the buffer which stores all captured `Containers` is available which can be used to suspend or replay the current buffer.

7.2.2 Visualization of Stationary Surroundings

In the following, the use of the aforementioned concept for device-independent data visualization is outlined for drawing the stationary surroundings. In Figure 7.3 its architectural implementation is shown using also the GUI framework Qt.

The main data structure to visualize is `Scenario` consisting of several `ScenarioNodes` and thus representing an ASG from a parsed SCNX file. For traversing this ASG, any object of a class implementing the interface `ScenarioVisitor` can be used. This interface is implemented by `ScenarioRenderer` which traverses all visualizable elements of the ASG by performing a type conversion for each node visited during traversal to call type-dependent visiting methods.

Inside these type-dependent visiting methods, commands for drawing the current node's visualization using the interface `Renderer` is used. This interface provides primitive

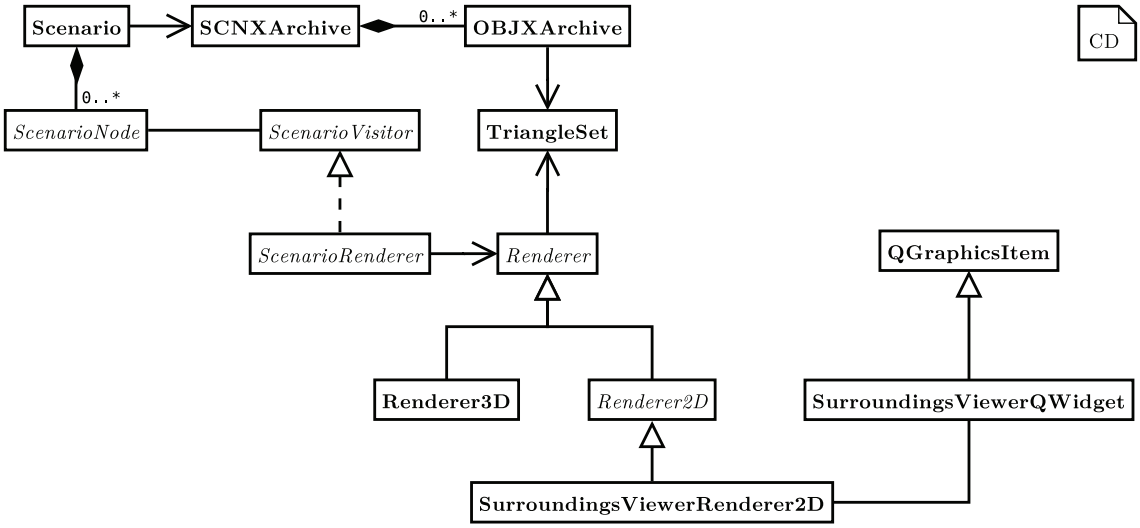


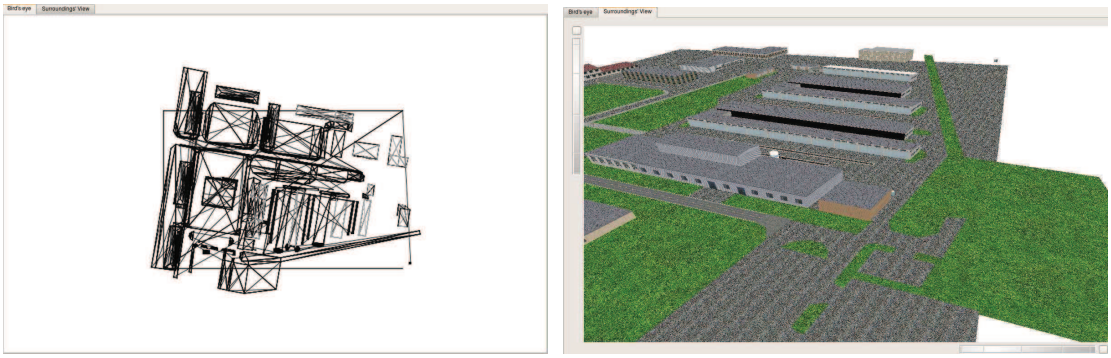
Figure 7.3: Device-independent data visualization for stationary surroundings: A given scenario is traversed for mapping the render-able information from the surroundings' elements to the drawing primitives provided by the interface `Renderer` as already mentioned in Section 5.4.5. The 2D visualization is implemented using a drawing context from Qt which is also used to develop the "monitor" application itself.

drawing instructions like `drawPoint`, `drawLine`, or `drawPolyLine`. Every method accepts a point or a set of points from $\mathbb{R}^3$. Depending on the concrete implementation of `Renderer`, these input values are visualized using a two-dimensional view called *bird's eye view* using `Renderer2D` or a freely visitable three-dimensional representation using `Renderer3D`. The latter one simply maps these calls to primitive drawing operations using OpenGL which itself is embedded either using the OpenGL Utility Toolkit (*GLUT*) providing rudimentary operations for creating a GUI or by more enhanced windowing tool-kits like Qt.

The former one which maps the primitive drawing operations to get a two-dimensional representation is still an abstract class which omits the real drawing methods. As shown in Figure 7.3, this class is derived to `SurroundingsViewerRenderer2D` implementing these pure virtual methods using `QGraphicsItem` provided by Qt. The representation using this alternative simply omits the z-coordinate when getting called in the regular repainting method of the windowing toolkit.

As already mentioned before, an SCNX file can also contain complex models provided by 3D modeling software. Currently, the Wavefront format is supported as mentioned before as compressed OBJX files containing files for describing faces consisting of triangles and its normals, material information, and images for the model's textures. For getting these files into a device-independent representation, the class `TriangleSet` is used to

describe a set of triangles and their associated materials. Thus, a complete model consists of a list of several sets of triangles. These list of triangle sets is passed to `Renderer` which simply implements the appropriate methods in one of its subclasses.



(a) Simplified two-dimensional visualization ("bird's eye view").

(b) Three-dimensional visualization.

Figure 7.4: Resulting representation using the concept of device-independent visualization. The camera on the right hand side is located in the lower left corner of the two-dimensional image pointing to its upper right corner.

Results for a two-dimensional and a three-dimensional representation are shown in Figure 7.4. In the figure on the left hand side, the mapping of a three-dimensional representation of a complex model describing a three-dimensional surroundings is shown. Therefore, `SurroundingsViewerRenderer2D` simply flattens the triangles of the model to the ground layer. The figure on the right hand side shows the visualization using the same ASG as input data structure but the `Renderer3D` as visualization engine. Thus, a freely visitable representation is realized.

Obviously, a bird's eye view could by achieved by either letting the camera pointing along the z-axis or by computing a perspective projection using a 3D visualization. However, both solutions require technically a three-dimensional context for visualization. With the concept presented before, a pure two-dimensional representation using the same unmodified input data can be realized.

7.2.3 Visualization of Dynamic Elements

To visualize dynamic elements like the own vehicle, sensor's raw data, or virtual objects for illustrating algorithm's intermediate steps, an analog concept as already outlined before is realized. Its architectural concept is presented in Figure 7.5.

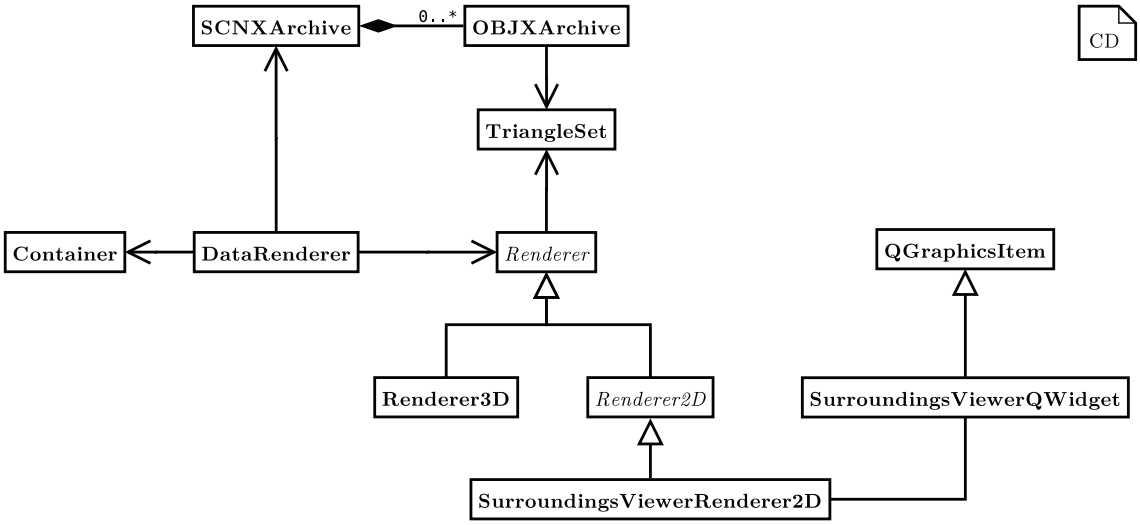


Figure 7.5: Device-independent data visualization for dynamic elements: Comparable to the stationary visualization, the scenario data is used to retrieve information about complex model provided by 3D modeling programs. Furthermore, all sent `Containers` can simply be visualized by a centralized mapping to the drawing primitives of interface `Renderer` which is carried out in the class `DataRenderer`.

Comparable to the concept already mentioned before, a device-independent data visualization is implemented. Therefore, the interface `Renderer` encapsulating primitive drawing operations is simply reused. The data to be visualized is retrieved from a running ClientConference by broadcasting `Containers` containing serialized objects. Thus, a `DataRenderer` is provided which simply uses a concrete 2D or 3D renderer implementing the interface `Renderer` for mapping a `Container` into a visual representation.

For an intuitional representation of some surroundings' elements like the own vehicle, models produced by a 3D modeling program can be used. Therefore, the `DataRenderer` has access to these models using an SCNX archive and maps the model into a device-independent list of triangle sets.

Combining both concepts, stationary surroundings enriched by dynamic elements can be simply realized by applying the `ScenarioRenderer` first. Afterwards, the current state of dynamic elements is drawn using the `DataRenderer`. Furthermore, the strict separation between drawing primitives realized by a concrete visualization application and the concrete representation of an object provided by the framework $\mathcal{H}$esperia, further data structures can be easily visualized without the need for modifying the visualization application by simply adding the necessary mapping to primitive drawing operations.

7.3 Unattended and Automatic Monitoring of an SUD for Acceptance Tests

Besides interactive monitoring for directly supporting developers, reporting of the system's quality is necessary to evaluate its maturity. For realizing automated acceptance tests as outlined in Chapter 3 an interface is required to evaluate *repeatedly* and *unattendedly* the system's behavior over time. Moreover, this interface shall allow a similar usage like unit tests for being combined with continuous integration systems. In the following, the software architecture for evaluating a system under test which completes the simulations of the system and the system's context as already outlined in Section 6.3 is presented.

7.3.1 Architecture for the Reporting Interface

In Figure 7.6, the general software architecture for evaluating running systems under test is shown, which completes Figure 6.3 from Section 6.3.2. Comparable to the class `SystemFeedbackComponent` which is used to compute values for the system's context to feed back information to the system under test, `SystemReportingComponent` derives from `SystemContextComponent`.

However, `SystemReportingComponent` does not have an association to an instance of `SendContainerToSystemUnderTest` preventing them to send any data to the system under test. Thus, any instance of a subclass implementing the method `void report(const core::wrapper::Time&);` gets automatically all `Containers` sent between any applications from the system under test as well as all data sent from the `SystemFeedbackComponents` to the system under test. But contrary to the aforementioned class, an instance of `SystemReportingInterface` can only evaluate the incoming data without interfering with the rest of the running simulation of the system under test and all `SystemFeedbackComponents`.

Thus, regarding *separation of concerns*, `SystemFeedbackComponents` compute necessary information to operate the system under test with the desired level of details, while `SystemReportingComponents` can evaluate the system under test by inspecting all sent `Containers`. Therefore, not only the required subclasses of `SystemReportingComponent` can be easily composed and added to a `RuntimeEnvironment`, but these subclasses can also be applied after an executed simulation during a post-processing stage without changing their code. Hereby, any necessary `SystemReportingComponent` can be applied to the captured and

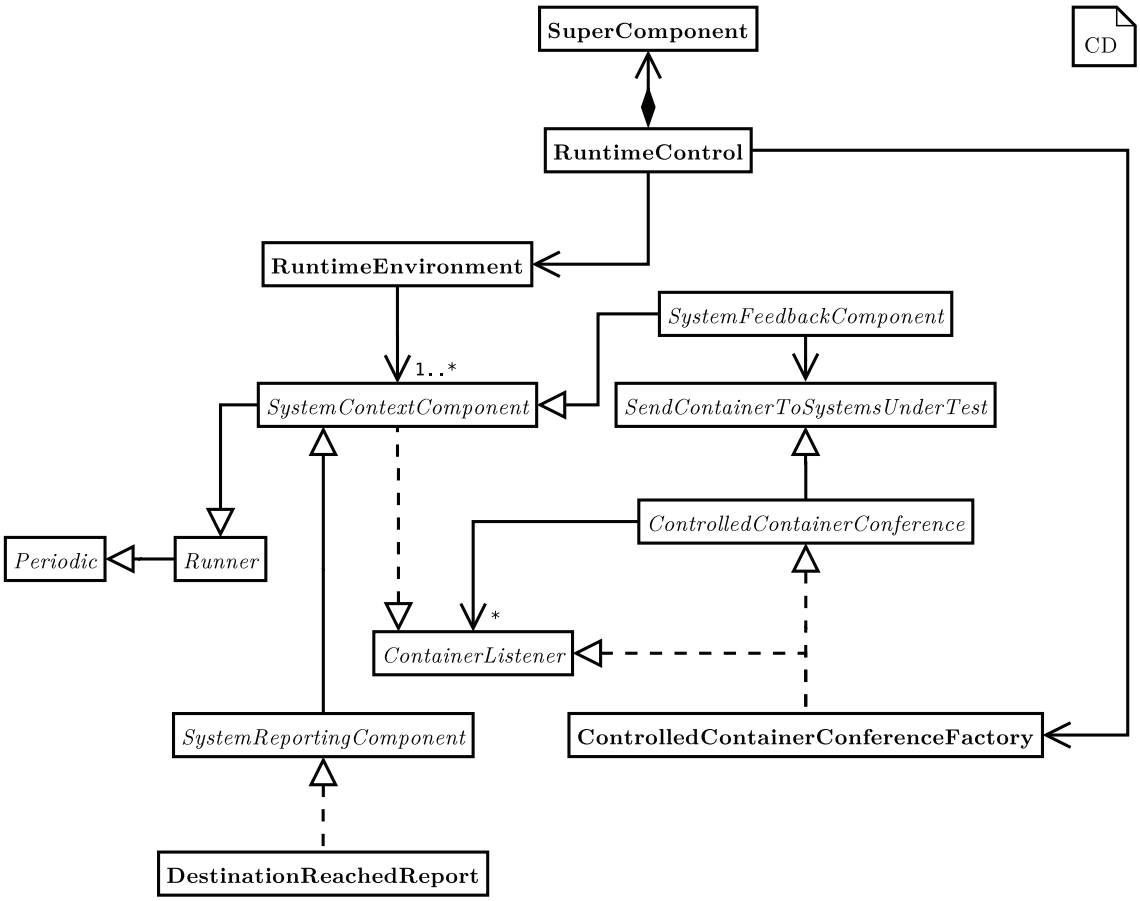


Figure 7.6: Software architecture for reporting components which evaluate the system's context. All reporting components derive from `SystemReportingComponent` which allows a specified frequent scheduling by `RuntimeControl`. Furthermore, these components are automatically receiving all sent `Containers` for evaluation.

recorded data during the previous simulation run because it simply evaluates a given stream of `Containers`. Thus, even after a complex system simulation, additional or time-consuming evaluations are simply possible.

7.3.2 Usage of Reporting Interface

In the following, various scenarios for using the outlined reporting interface `SystemReportingComponent` are described. Therefore, subclasses are derived from this class which implement aspects that can be evaluated independently during a running system simulation for acceptance tests for example.

7.3.2.1 Evaluating the System: Destination Reached Report

In the following, an exemplary usage of the reporting interface is shown. The goal for the implemented subclass of `SystemReportingComponent` called `DestinationReachedReport` is to monitor continuously the current position of the vehicle and to report finally whether it has reached a predefined goal. This reporter can also be added simply several times to monitor the passing of a sequence of given destinations.

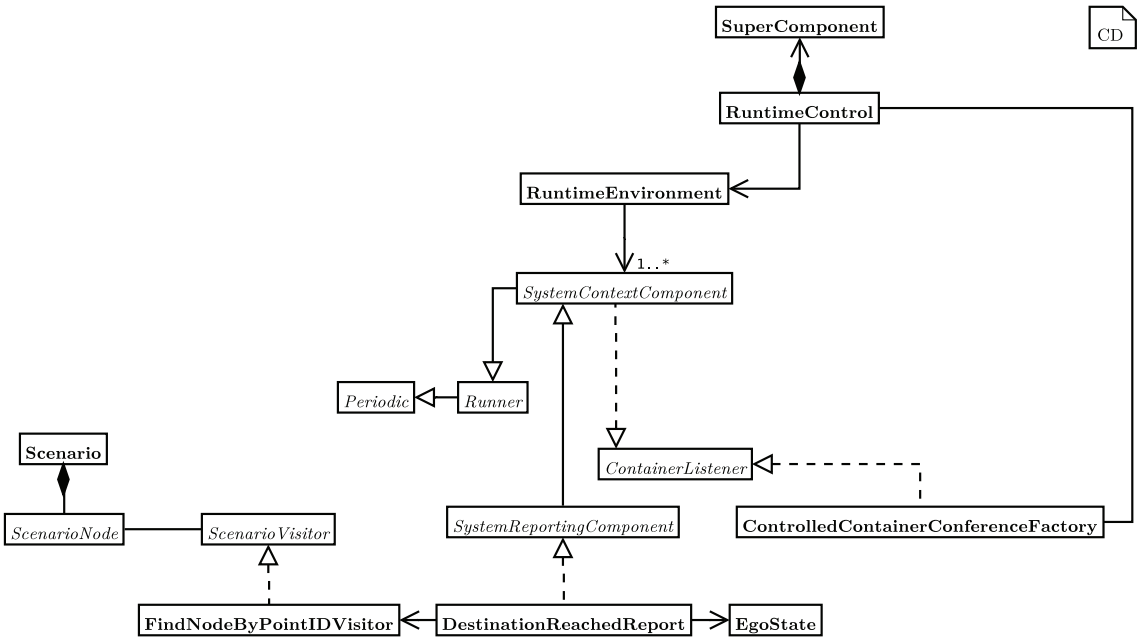


Figure 7.7: Software architecture for reporting whether a given destination was successfully reached. The `DestinationReachedReporter` implements the interface `SystemReportingComponent` to receive automatically all send `Containers`. Furthermore, it uses the formally specified scenario for getting information about available way-points which can be used as destinations for an AGV.

As shown in Figure 7.7, the `DestinationReachedReport` derives from `SystemContextComponent` to get all data sent between any application from the system under test and `SystemFeedbackComponents`. Furthermore, this class uses the DSL for getting information about the digital map consisting of identifiable way-points describing absolute Cartesian coordinates. Using a way-point's identifier passed to `DestinationReachedReport` at construction, the visitor `FindNodeByPointIDVisitor` traverses the ASG constructed from the given instance of the DSL describing the stationary surroundings to find the identifier of the desired destination. Using this identifier, the associated position is retrieved from the ASG and stored for further usage.

During a system simulation run, the instance of `DestinationReachedReport` is continuously called. Upon activation, the instance iterates through its buffer containing all sent data and inspects all `Containers` containing information about the current vehicle's state, e.g. its position, orientation, and velocity. The current vehicle's position is used to compute the distance to the desired destination. As soon as the computed distance is less than a given threshold, the instance of the class `DestinationReachedReport` returns true after finishing the system simulation.

Thus, besides the software framework $\mathcal{H}$esperia no further tooling is required to evaluate an SUD unattendedly and in an automated manner. Therefore, these reporters which realize the metrics based on the customer's acceptance criteria can simply be specified as unit tests as shown in Listing 7.1. These unit tests themselves can be executed regularly using a continuous integration system like CruiseControl [38].

```
#include "cxxtest/TestSuite.h"

class SimpleTestSuite : public CxxTest::TestSuite {
  public:

    void testReachingDestination() {
      // 0. Setup system's configuration.
      stringstream config;
      config << "global.scenario = file://Scenarios/↩
      RichmondFieldStation.scnx" << endl;

      // 1. Setup runtime control.
      DirectInterface di("225.0.0.100", 100, ↩
      sstrConfiguration.str());
      VehicleRuntimeControl vrc(di);
      vrc.setup(RuntimeControl::TAKE_CONTROL);

      // 2. Setup application.
      const string START_WAYPOINT = "1.4.2.4";
      const string DESTINATION_WAYPOINT = "1.5.1.6";

      // Define the actual SUD.
      SimpleDriver mySimpleDriver(START_WAYPOINT, ↩
      DESTINATION_WAYPOINT);

      // 3. Define the SUD's system's context.
```

```
      const float FREQ = 5;
      SimplifiedBicycleModel bicycleControl(FREQ, config.str↩
      ());

      // 4. System's reporting component.
      const float THRESHOLD_DESTINATION = 1; // 1m threshold.
      DestinationReachedReport destinationReached(config.str↩
      (), DESTINATION_WAYPOINT, THRESHOLD_DESTINATION);

      RecordingContainer recorder(FREQ, "simpleTestSuite.rec"↩
      );

      // 5. Compose the simulation.
      RuntimeEnvironment rte;
      rte.add(mySimpleDriver);
      rte.add(bicycleControl);
      rte.add(destinationReached);
      rte.add(recorder);

      // 6. Run application under supervision of ↩
      RuntimeControl for maximum 180s.
      TS_ASSERT(vrc.run(rte, 180) == RuntimeControl::↩
      APPLICATIONS_FINISHED);

      // 7. Check if the destination was finally reached.
      TS_ASSERT(destinationReached.↩
      hasReachedDestinationWaypoint());

      // And finally clean up.
      vrc.tearDown();
    }
};
```

Listing 7.1: Integration of customer's acceptance criteria using reporters in unit tests.

On the example of a unit test realized with CxxTest[156], a simple executable test specification was created which evaluates whether the SUD fulfills the customer's requirements by using the customer's acceptance criteria as the continuously applied metric. In this case, the SUD is an autonomously driving vehicle and the metric evaluates if the vehicle has finally reached its destination.

First, in line 8 et seqq. the currently valid system configuration is specified. Afterwards, in line 12 et seqq. the scheduler for the system simulation is set up. In this case, the application is directly under control of the `RuntimeControl`; another implementation provides a command-line interface which allows an interactive evaluation of the SUD if desired. In line 21, the actual SUD is set up while in the following lines its system's context is specified which consists of the simplified bicycle model as specified in Section 6.4.3. In line 28 et seqq. a metric which reflects the customer's acceptance criteria is set up. This criterion is continuously applied to the running SUD for gathering information.

In line 34 et seqq. the `RuntimeEnvironment` is composed for defining which components must be scheduled. In this example, the `RuntimeEnvironment` consists of the SUD, its system's context, and the `DestinationReachedReport` which evaluates a customer's acceptance criterion.

The system simulation itself is started in line 41 for a maximum duration of 180s. Thus, this method call blocks for a maximum duration of 180s. If this hard deadline is missed, the call returns with a return code which is not equal to `APPLICATIONS_FINISHED` and thus, it describes the reason for the failure; this method call also returns immediately if an exception occurs. The return code of this method call is only fulfilled when the system simulation could successfully be executed within the specified time limit.

In line 44, the fulfillment of the specified metric is checked. Afterwards, the test case is cleaned up in line 47. The last call is not mandatory because the instance of `VehicleRuntimeControl` is automatically cleaned up when it gets destroyed when leaving the current scope.

For further inspections for example in case of a failed test case, in line 31 a recording component which is included in `libcontext` is created. This component is also added to the `RuntimeEnvironment`. Thus, it automatically receives all sent data during an unattended system simulation and all received `Containers` are stored in a data file. This data file can be read later on using the application `Monitor` as mentioned in Section 7.2 for a manual step-by-step in-depth analysis.

7.3.2.2 Evaluating the System: Distance to Route Report

Besides the aforementioned implementation for validating if the vehicle has reached its intended destination, its distance to a given or the optimal route can be monitored. Therefore, the `SystemReportingComponent` named `DistanceToRouteReport` is implemented which extends the reporting component mentioned before.

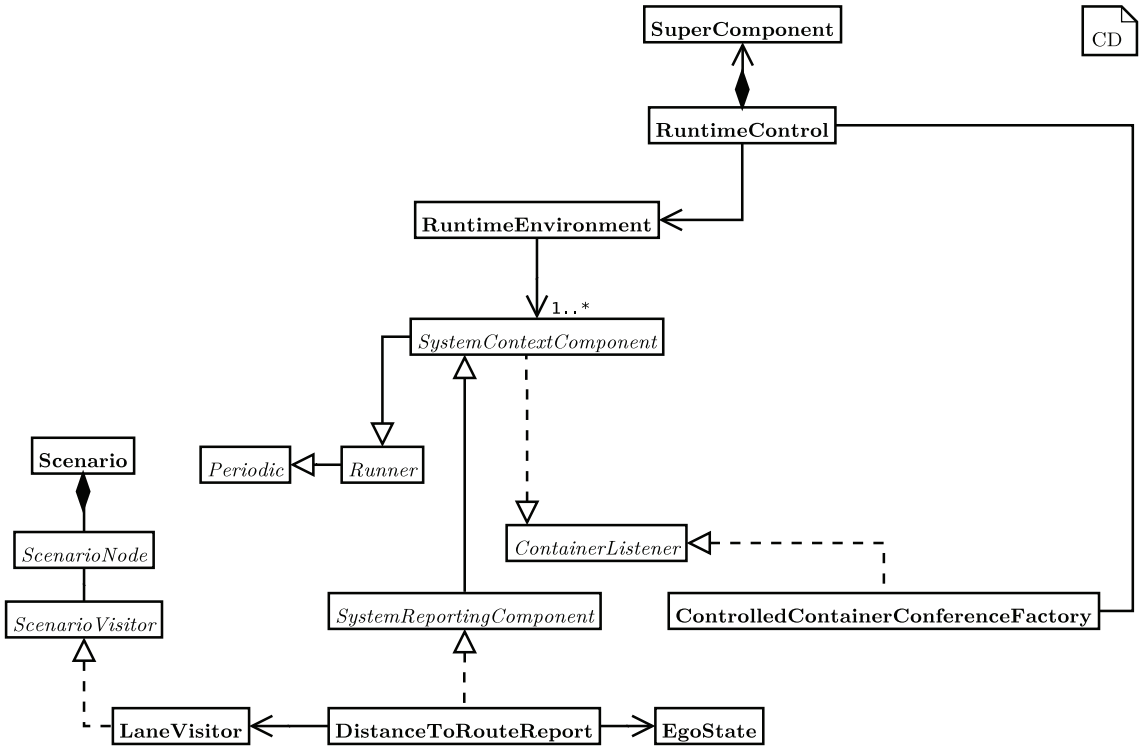


Figure 7.8: Software architecture for a `SystemReportingComponent` to evaluate whether the vehicle's distance to an optimal route is continuously less than a given threshold. Therefore, comparable to the aforementioned component, `DistanceToRouteReport` automatically receives all distributed `Containers` and evaluates the current vehicle's position and orientation to a pre-calculated given or to the optimal route using a `LaneVisitor` which traverses the road network.

In Figure 7.8, the software architecture of the `DistanceToRouteReport` is shown. Comparable to the aforementioned reporting component, it also derives from `SystemContextReport` to receive automatically all distributed data. Moreover, it relies on the DSL to calculate the optimal route between two given points from the road network. Therefore, it uses a `LaneVisitor` which traverses the road network to find an optimal route between the given points during the initialization phase of this reporting component. Alternatively, it can use a user-contributed route to evaluate the vehicle's distance to the poly-line of that given sequence of way-points.

During a running system simulation, the `DistanceToRouteReport` continuously evaluates the currently incoming vehicle's position and orientation to calculate the vehicle's distance to the regarded route's segment. As long as this distance is less than a given threshold, the `DistanceToRouteReport` reports true after finishing the current system simulation; however, when the vehicle's distance is greater than the specified threshold for the first time, this reporter not only returns false but also provides information

about the vehicle's position and orientation whenever it violates the specified boundary.

7.3.2.3 Evaluating the System in its Context: Distance to Objects Report

The both aforementioned reporting components evaluate the system's behavior without regarding its system's context. Therefore, a further reporting component named `DistanceToObjectsReport` supports the acceptance tests by evaluating the system's distance to any existing object in the system's context.

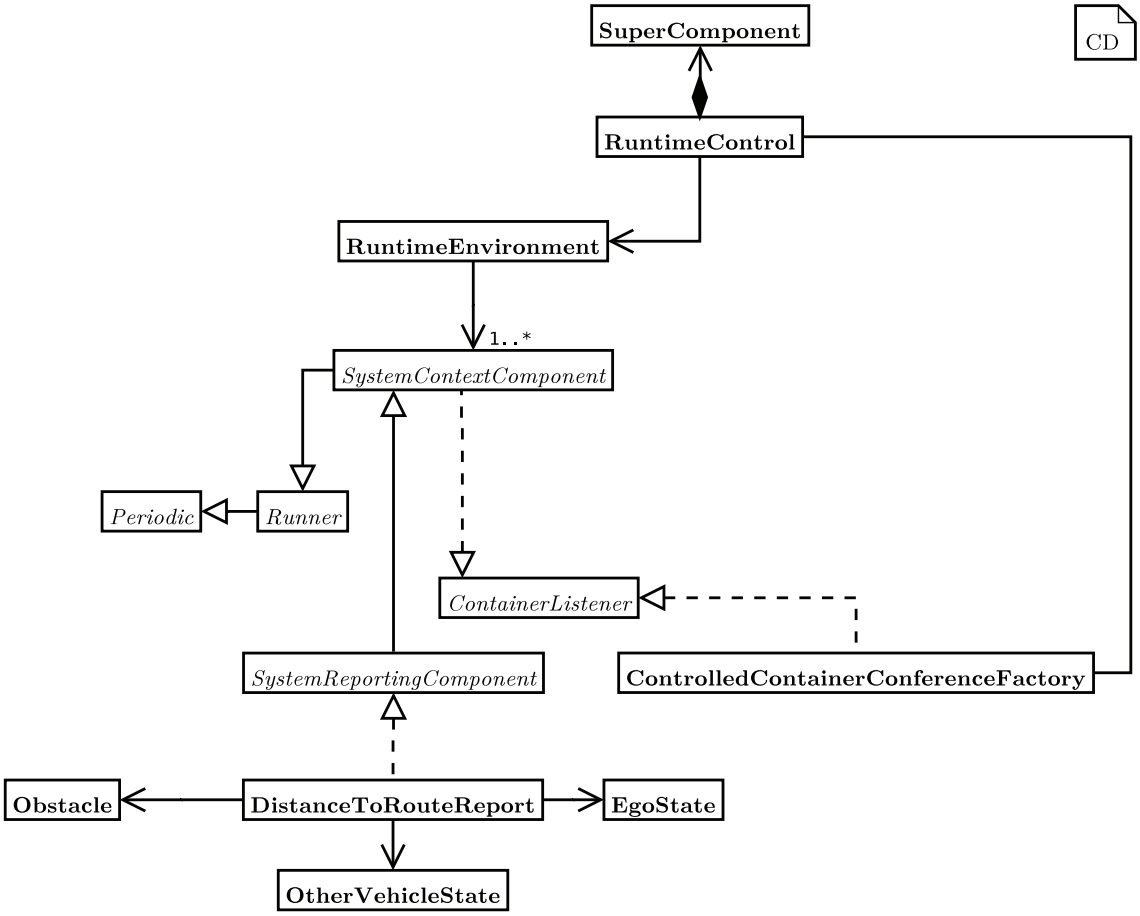


Figure 7.9: The class diagram depicts the software architecture for a component which continuously evaluates the system's behavior within its system's context. Therefore, the `DistanceToObjectsReport` evaluates the data from the system's context namely `Obstacle` and `OtherVehicleState`. For both, the Euclidean distance is calculated; moreover, for the former the polygonal shape is also evaluated to compute the distance which is compared to a user-specified distance. The distributed data is received automatically as mentioned in Section 7.3.

Alike the both already described reporting components, `DistanceToObjectsReport` also derives from `SystemReportingComponent` as shown in Figure 7.9.

Therefore, it receives automatically all distributed `Containers`. This data contains all necessary information for calculating the distances which are compared with the user-specified threshold. The first information is distributed by `OtherVehicleState` which is used to describe positions of other vehicles as outlined in Section 6.4.8. Further information is provided by `Obstacle` which additionally contains a polygonal description of the object's shape. For all points of both objects, the Euclidean distance is calculated and compared to the user-specified threshold. Moreover, for the polygonal shape of `Obstacle` the perpendicular points are computed and compared as well to consider the distances towards any side of the object as well.

Using this reporter, the system's behavior with stationary and dynamic elements from its system's context can be simply evaluated. Moreover, the quality of a sensor data fusion module producing an abstract representation based on sensor's raw data like a laser scanner or a monocular camera as described in Section 6.4.6 and Section 6.4.4 can be evaluated as well.

In the case study presented in Section 8.2.5, some of the aforementioned reporting components are used to evaluate and ensure the quality of a software component on a example of a real autonomous vehicle.

8 Case Study and Evaluation

In this chapter, an example for practical use of the tools and framework described in the previous chapters is presented. First, a benchmark for the framework $\mathcal{H}$esperia is presented. Following, its application for an AGV is described.

8.1 Benchmark for $\mathcal{H}$esperia

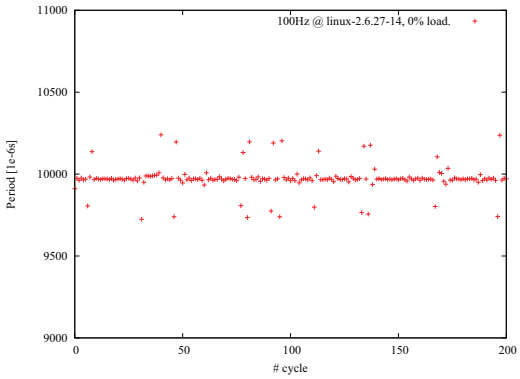
In the following, some benchmarks for outlining the performance of the framework $\mathcal{H}$esperia are shown. Its real-time capabilities and communication features are of substantial interest due to the intended application in the area of real-time data processing applications.

8.1.1 Performance of the Timing

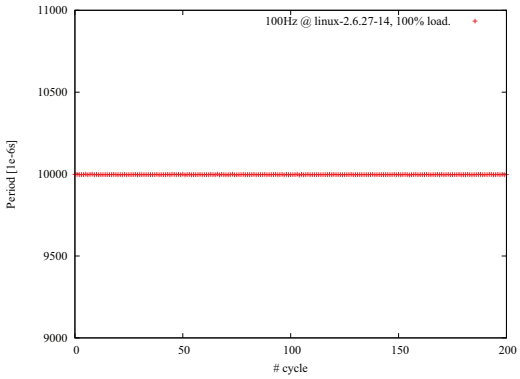
First, the schedulability and timing in the framework $\mathcal{H}$esperia is discussed to illustrate its real-time capabilities. As already outlined in Chapter 5, the real-time implementation depends directly from the OS which is chosen for using $\mathcal{H}$esperia. For carrying out the following tests, a regular Linux 2.6.27-14-generic kernel without real-time extension for the non-real-time tests was chosen whereas a Linux 2.6.27-3-rt kernel which includes the preempt-rt patch set [107] was chosen which allows soft-real-time capabilities [63]. In Figure 8.1, the timing on the former Linux kernel without a real-time extension is shown.

In Figure 8.1a, a process with a specified frequency of 100Hz shall be scheduled every 10ms. Furthermore, there is no other system load. For 200 cycles, it can be seen that the process is scheduled for about every 9.971ms with a standard deviation of 0.065ms. In the worst case, the process is scheduled after 10.24ms violating the specified timing.

On the right hand side in Figure 8.1b, the same process is shown with the same frequency. However, at this time the process spawns as many other threads as CPUs available causing 100% system load. Now, the scheduler of the operating system prefers this process over others due to its required computation time. Thus, the average scheduling time is at

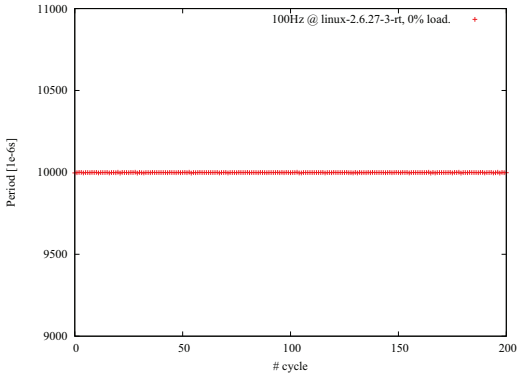


(a) Benchmark for the timing for a process with a 100Hz thread and no further system load on a Linux 2.6.27-14-generic kernel: The specified frequency of 100Hz could not be fulfilled reliably.

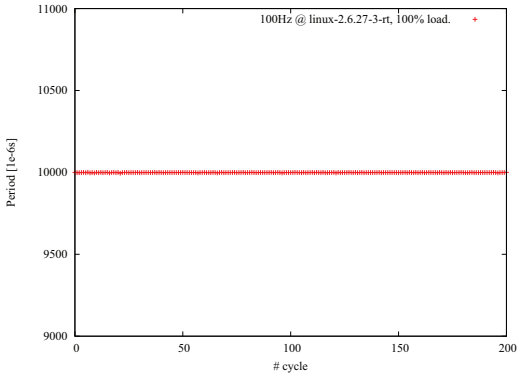


(b) Benchmark for the timing for a process with a 100Hz thread with two other threads causing 100% system load on a Linux 2.6.27-14-generic kernel: Due to the increased system load which is caused by the two spawned threads the Linux kernel scheduler prefers even the 100Hz thread more often.

Figure 8.1: Benchmark for the timing of the framework $\mathcal{H}$esperia for the Linux kernel 2.6.27-14-generic.



(a) Benchmark for the timing for a process with a 100Hz thread and no further system load on a real-time Linux 2.6.27-3-rt kernel: On a Linux kernel with preempt-rt real-time extensions the specified frequency of 100Hz is fulfilled with a very low jitter even at no system load.



(b) Benchmark for the timing for a process with a 100Hz thread with two other threads causing 100% system load on a real-time Linux 2.6.27-3-rt kernel: The same setup as mentioned before is fulfilled reliably on a Linux kernel with preempt-rt real-time extensions.

Figure 8.2: Benchmark for the timing of the framework $\mathcal{H}$esperia for the Linux kernel 2.6.27-3-rt.

9.997ms with a standard deviation of 0.00498ms. Now, the worst cast schedule time is at 10.0ms.

In Figure 8.2, the same process is executed on the same computer system now running the Linux 2.6.27-3-rt kernel with preemption patches. These patches are available for all major Linux distributions.

On the left hand side in Figure 8.2a, the process is running without spawning other threads; thus, the system load is nearly at 0%. The process is scheduled every 9.9988ms on average with a standard deviation of 0.00358ms. The worst case scheduled execution was at 10.01ms.

In Figure 8.2b, the process is spawning other threads causing a system load of 100%. Now, the scheduler prefers this process and the average scheduled execution time is at 9.9998ms. The standard deviation is about 0.00329ms and the worst case scheduled execution was at 10.01ms.

As a result, it can be seen that the Linux kernel with preemption patches is a good choice if no commercial real-time operating system is available. Moreover, when using the framework $\mathcal{H}$esperia, a convenient way to realize real-time applications by simply specifying the required frequency and implementing a specified abstract method is provided; anything else is handled transparently for the user-contributed application by the software framework. However, it is obvious to obey design patterns like avoiding expensive memory allocations or locking for critical sections between different threads when developing real-time applications [43].

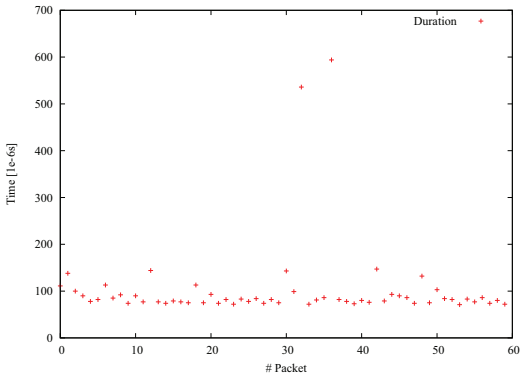
8.1.2 Performance of the Communication

After discussing the schedulability and timing, the communication in the framework $\mathcal{H}$esperia is evaluated to show that broadcasted UDP packets which are used for a ClientConference can be used to realize a fast and convenient way for several communicating applications. As already mentioned in Section 5.4.1, a ClientConference is realized as a UDP multi-cast transferring the atomic type `string` which itself contains the serialized data structure `Container`.

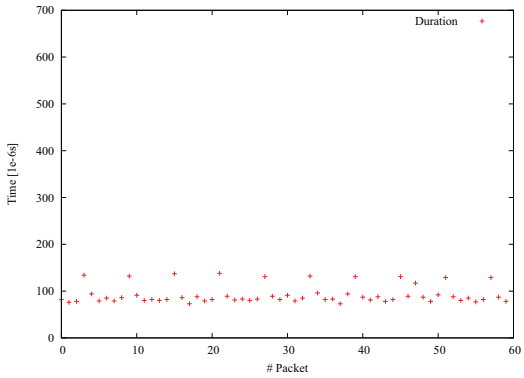
8.1.2.1 Local Communication on One Computer

In the following, the atomic type `string` of the fixed size of 256 bytes is sent between different applications and the duration is measured. For these tests, the communication performance of the framework $\mathcal{H}$esperia on a local computer is evaluated.

First, as shown in Figure 8.3, a packet of 256 bytes is sent with 1Hz from one sender to one receiver. On the left hand side in Figure 8.3a, the executed processes are running



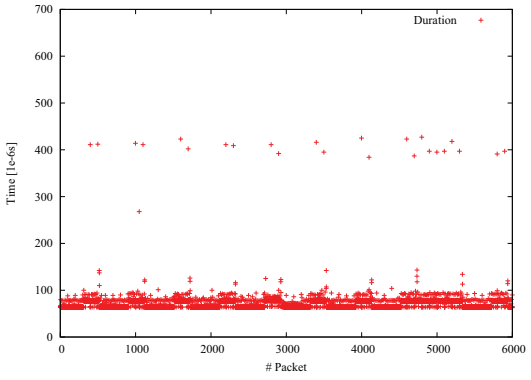
(a) Benchmark for one sender sending data at 1Hz and one receiver running as user processes on the same computer: Some packets are delayed up to 0.6ms.



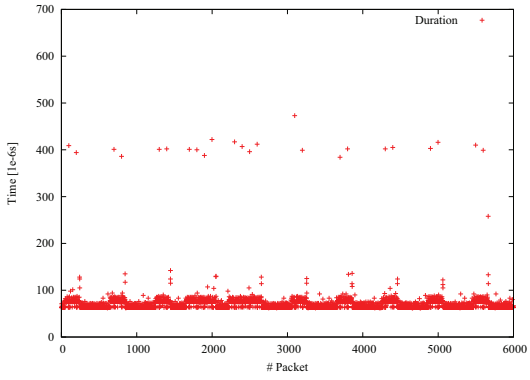
(b) Benchmark for one sender sending data at 1Hz and one receiver running as privileged processes on the same computer: When the same test is carried out using a privileged user the average duration is approximately at 0.0936ms without the aforementioned peaks.

Figure 8.3: Benchmark for one sender sending data at 1Hz and one receiver running on one computer.

as non-privileged processes. It can be seen, that the average duration is approximately 0.11ms with some peaks at nearly 0.6ms. However, when both processes are executed as privileged processes, these peaks seem to disappear and the average duration is around 0.0936ms. In both cases, no packets are lost.



(a) Benchmark for one sender sending data at 100Hz and one receiver running as user processes on the same computer.

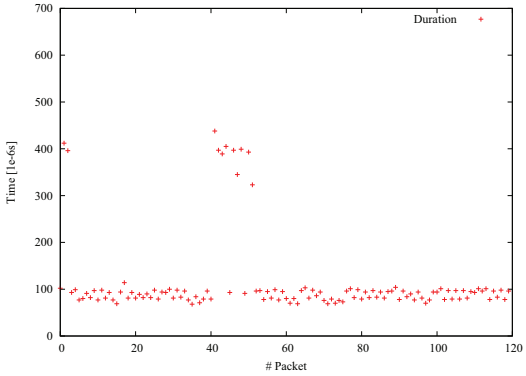


(b) Benchmark for one sender sending data at 100Hz and one receiver running as privileged processes on the same computer.

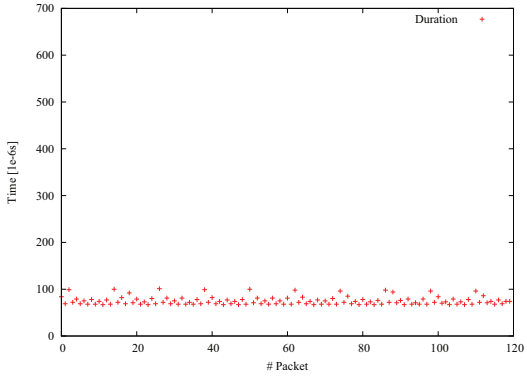
Figure 8.4: Benchmark for one sender sending data at 100Hz and one receiver running on one computer: On average the transmission duration is approximately at 0.07ms with no significant differences at all.

The same setup is shown in Figure 8.4. On the left hand side in Figure 8.4a, the sender is

sending at 100Hz constantly 256 bytes to one receiver. There, the average duration was about 0.0715ms with a peak at 0.427ms. On the right hand side, both processes are run as privileged processes. Now, the average transmission duration is at 0.075ms with a peak at 0.472ms. Thus, no significant difference can be remarked. In both cases, no packets are lost as well.



(a) Benchmark for two senders sending data at 1Hz and one receiver running as user processes on the same computer: The average duration is around 0.126ms with a worst case duration at approximately 0.438ms.

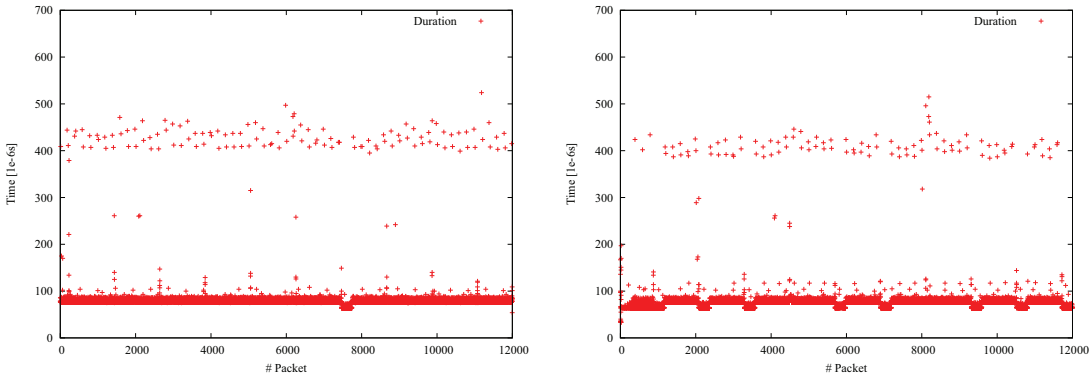


(b) Benchmark for two senders sending data at 1Hz and one receiver running as privileged processes on the same computer: In this case, the average duration is at approximately 0.076ms.

Figure 8.5: Benchmark for two senders sending data at 1Hz and one receiver running on one computer.

In Figure 8.5, the same test using two senders which are sending at 1Hz and one receiver is shown. On the left hand side in Figure 8.5a, two sending processes and one receiving process running as non-privileged process are shown. On average, a UDP packet needs nearly 0.126ms for transmission, in the worst case up to 0.438ms. On the right hand side in Figure 8.5b, when all three process are running as privileged processes, the average duration for transmitting a UDP packet of 256 bytes payload is for about 0.0756ms while the maximum duration is 0.13ms. In every case, no packet got lost.

The same setup is shown in Figure 8.6 where also two sending and one receiving process are measured. On the left hand side in Figure 8.6a, two senders which are sending at 100Hz running as non-privileged processes are shown. On average, the duration is nearly at 0.0874ms, while the worst case duration is slightly over 0.5ms. On the right hand side, all three processes are run as privileged processes. The performance is nearly the same as in the previous case: The average duration for transmitting a `string` is 0.0788ms, while the worst case duration is also slightly greater than 0.5ms. Like in all other cases, no packets were lost as well.



(a) Benchmark for two sender sending data at 100Hz and one receiver running as user processes on the same computer.

(b) Benchmark for two sender sending data at 100Hz and one receiver running as privileged processes on the same computer.

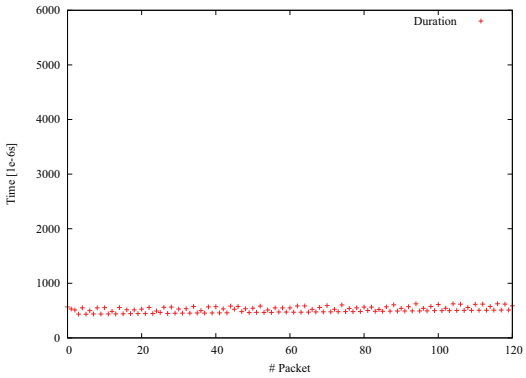
Figure 8.6: Benchmark for two senders sending data at 100Hz and one receiver running on one computer: On higher network traffic on the local network device, no significant difference between privileged and non-privileged processes can be deduced.

Altogether, the overall results show that the use of UDP for data transmission is a reasonable choice even at higher network loads because no packets were lost at any benchmark. Furthermore, even on an increasing network bandwidth's consumption, the worst case transmission duration is around 0.5ms for only a small amount of packets. Furthermore, the transmission duration is reduced when the processes are executed using a privileged user account.

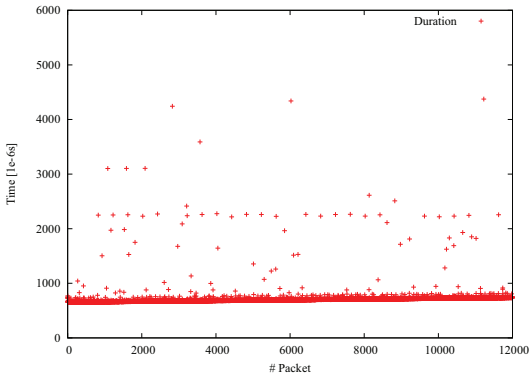
8.1.2.2 Communication on Two Computers

In the following, the performance of the framework $\mathcal{H}$esperia for the communication between two computers is analyzed. Therefore, both computers were synchronized using either Network Time Protocol (*NTP*) [103] or Precision Time Protocol (*PTP*) [84]. The results for each protocol, which obviously depend on further running processes which also use the network for communication, are shown below.

In Figure 8.7, two sending processes running on one computer and one receiving process running on another computer are shown. Each process is executed as a non-privileged process. On the left hand side in Figure 8.7a, both senders are sending at 1Hz. The average transmission duration is approximately 0.591ms, while the worst case duration is at 0.768ms. On the right hand side in Figure 8.7b, both senders are running at 100Hz. In this case, the average transmission duration increases to 0.778ms, while the worst case duration is at 4.3ms. In both cases, no packets were lost.

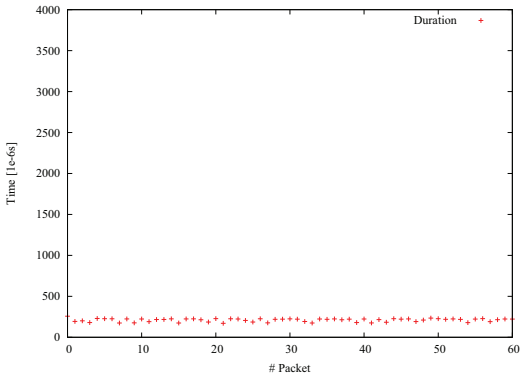


(a) Benchmark for two senders sending data at 1Hz and one receiver running as user processes on two computers synchronized using NTP: The average transmission duration is at approximately 0.591ms while the worst case transmission duration is around 0.768ms.

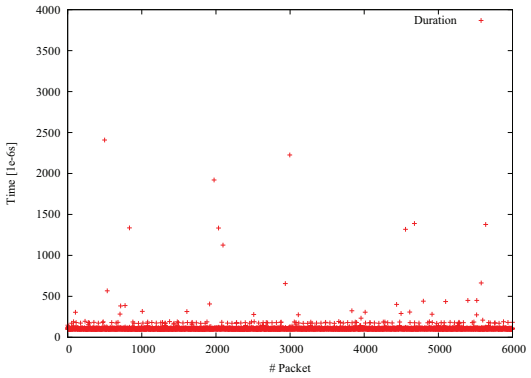


(b) Benchmark for two senders sending data at 100Hz and one receiver running as privileged processes on two computers synchronized using NTP: On a higher network bandwidth's consumption the average transmission duration is increasing to 0.778ms while the worst case transmission duration is at nearly 4.3ms.

Figure 8.7: Benchmark for two senders sending data at 1Hz and 100Hz and one receiver running on two computers.



(a) Benchmark for two senders sending data at 1Hz and one receiver running as user processes on two computers synchronized using PTP: The average transmission duration is at approximately 0.209ms while the worst case transmission duration is at 0.429ms.

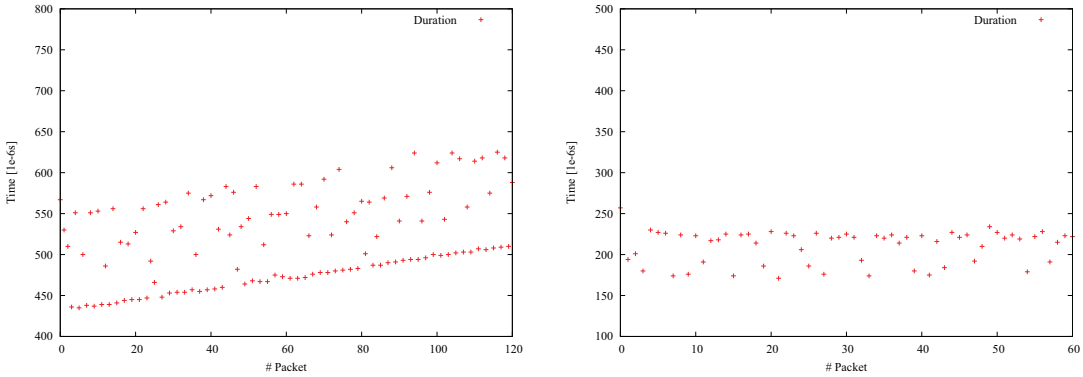


(b) Benchmark for two senders sending data at 100Hz and one receiver running as privileged processes on two computers synchronized using PTP: On higher bandwidth's consumption the average transmission duration is at approximately 0.112ms and the worst case transmission duration is around 2.5ms.

Figure 8.8: Benchmark for two senders sending data at 1Hz and 100Hz and one receiver running on two computers.

In Figure 8.8, a setup using one sender and one receiver each running on a separate computer is shown. On the left hand side in Figure 8.8a, the sender is sending with 1Hz and running as a regular non-privileged process. Here, the average transmission duration is

for about 0.209ms, while the maximum duration is at 0.429ms. On the right hand side in Figure 8.8b, the same setup with the sender running at 100Hz is shown. In this case, the average duration is nearly 0.112ms, while the worst case is slightly less than 2.5ms.



(a) Benchmark for two senders sending data at 1Hz and one receiver running as user processes on two computers synchronized using NTP.

(b) Benchmark for two senders sending data at 1Hz and one receiver running as user processes on two computers synchronized using PTP.

Figure 8.9: Comparison between NTP and PTP: The plot on the left hand side is translated to the bottom to allow a more intuitional comparison because the measured durations on the left hand side have an additional offset. However, this inaccuracy is within the NTP's specification which is under ideal conditions at least a multiple of 1×10^{-6}s according to [103].

Another important remark about NTP and PTP is shown in Figure 8.9. On the left hand side in Figure 8.9a, the communication between two computers using UDP for transportation of 256 bytes payload, which are synchronized using NTP is shown. It is very obvious that the synchronized time is drifting and the transmission duration is increasing. In this case for 120s the transmission duration increases from about 0.43ms to 0.51ms. On the right hand side in Figure 8.9b, the same setup is shown using PTP is shown. In this case, no drift is remarkable and the transmission duration is nearly constant.

Altogether, the results show that even for the communication between several computers the UDP is a reasonable choice. However, depending on the desired use case, the time synchronization between all participating computers is important and in time-critical environments which are limited to local networks, the PTP is a reasonable choice to ensure only a very low drift in the independent clocks.

8.2 Application for $\mathcal{H}$esperia on an Autonomous Ground Vehicle

The framework $\mathcal{H}$esperia was tested on an AGV at University of California, Berkeley in summer 2009. The description of this test is presented in the following. First, the vehicle and its model are presented in detail. Following, the test site and its modeling is described. Finally, the development and deployment of a simple algorithm for controlling the vehicle to locate itself on a digital map and navigate safely a route computed from this digital map is shown.

8.2.1 Ford Escape Hybrid – ByWire XGV

The vehicle used for testing the concepts implemented in the framework $\mathcal{H}$esperia is a 2008 Ford Escape Hybrid Sports Utility Vehicle (*SUV*) as shown in Figure 8.10a. For getting access to the steering wheel, acceleration, and brake system, the vehicle was modified into a so-called drive-by-wire system which allows open-loop control [157]. The same platform was also used by team VictorTango in the 2007 DARPA Urban Challenge and proved reliability [5].

As shown in Figure 8.10a, a stereo vision camera system and a single layer laser scanner are mounted on the AGV's roof. Furthermore, besides a Wireless-Local Area Network (*W-LAN*) antenna, a NovAtel GPS antenna for localization is also mounted on the highest position to avoid shadowing. In Figure 8.10b, the AGV's trunk is shown. In the trunk, two racks are mounted in a shock-proof manner to carry all vehicle's computers and power systems. The rack on the left hand side carries the TORC PowerHub system providing 6,000 watt, followed by a waveform generator for providing a synchronization signal for the stereo camera system. Finally, the NovAtel Synchronized Position & Attitude Navigation (*SPAN*) IMU HG-1700 is mounted on top the waveform generator for highly precise localization.

The rack on the right hand side contains five computers for data processing. Four computers mounted in half-2U cases and pair-wisely grouped provide an Intel Core 2 Quad CPU each along with 4GB RAM and 160GB SATA HDD for logging purposes. A Compact Flash (*CF*) card containing the operating system and the actual processes are used to avoid malfunctions due to heavy movements and accelerations. The 2U server on the rightmost side is a dual Intel Core 2 Quad with 8GB RAM and an 80GB SSD. This server is responsible for data acquisition from the IMU like position data, vehicle's orientation, and velocity, and for generating the steering and acceleration commands using a given tra-

(a) Ford Escape Hybrid – ByWire XGV's sensors: On the roof there is a stereo vision system alongside with a single layer laser scanner mounted for perceiving the vehicle's surroundings in front of the car.

(b) Ford Escape Hybrid – ByWire XGV's trunk: The incoming data is processed on several computers which are mounted in the rack on the right hand side of the trunk. On the left hand side, an IMU for providing highly precise position data is mounted as well as a waveform generator to provide a synchronization signal to trigger the independent cameras of the stereo vision system.

Figure 8.10: Overview of the Ford Escape Hybrid – ByWire XGV's sensors and trunk.

jectory. Furthermore, this computer runs the non-reactive visualization application to be shown both on the Liquid Crystal Display (*LCD*) mounted under the roof for passengers on the vehicle's back seats and on the small display of the car radio for the driver and the fellow passenger.

Based on the general system architecture for sensor- and actuator-based autonomous systems as already shown in Section 2.2, Figure 8.11 depicts the specific system architecture implemented in the 2008 Ford Escape Hybrid – ByWire XGV. The input message `Vehicle State` is a so-called heartbeat pulse message sent by the ByWire Real Time Controller indicating that it is operating properly. The AGV localizes itself using the aforementioned NovAtel SPAN system providing highly precise position data with an accuracy of $\sim \pm 1.8m$ and orientation data with an accuracy of $\sim \pm 0.02rad$ using the message `Position`. Both data is sent over UDP while the former is encapsulated using the JAUS message `HEARTBEAT_PULSE` [89] and broadcasted into the vehicle's sub-net, the latter is sent using a proprietary data structure using UDP. For avoiding additional message routing caused by the `NodeManager` as required in the JAUS specification, the XGV directly communicates with the required component bypassing the JAUS standard. Thus, freely available implementations like the OpenJAUS package [113] cannot be used

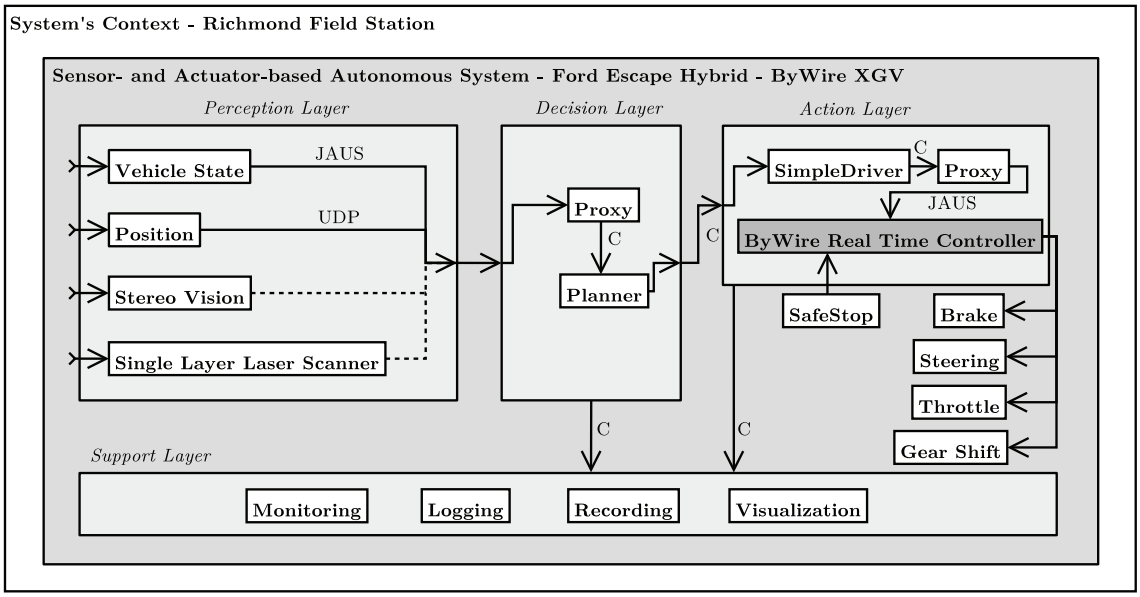


Figure 8.11: System architecture implemented in the 2008 Ford Escape Hybrid – By-Wire XGV. `JAUS` indicates data which is encapsulated into the `JAUS` protocol, `UDP` indicates data which is sent using a proprietary protocol, and `C` indicates data wrapped into a `Container` data structure; therefore, the application `proxy` from the framework $\mathcal{H}$esperia is used. Furthermore, the components *Planner* and *SimpleDriver* are realized as one combined application using the framework $\mathcal{H}$esperia as well. The applications from the support layer are described in Section 5.5, 5.6, and 7.2 and base also on the framework $\mathcal{H}$esperia.

with the vehicle without modifications.

Beyond, a stereo vision system along with a single layer laser scanner system are used for perceiving the vehicle's surroundings. For the demonstration shown in this chapter, these systems are not used for the vehicle control. Therefore, in Figure 8.11 these connections are depicted dashed. However, the vehicle model presented in Section 8.2.3 provides all sensor's raw data for the developers.

Following the perception layer, the decision layer processes all acquired data from the vehicle. Since the data itself is available in different data formats, the `proxy` application receives the data and translates it to the `Container` data format used in $\mathcal{H}$esperia. In Figure 8.11, the `Container` format is indicated by `C`. After translating the data, the planner checks if the vehicles has reached the next available way-point from the initially planned route. If no more way-points are available, the vehicle is stopped. The `SimpleDriver` which actually contains the planning algorithm uses the current position and orientation data to compute the next necessary steering commands to be sent to the car as described in Section 8.2.4. Since the vehicle can only be controlled using the aforementioned JAUS

messages, the `proxy` application translates the computed steering commands into JAUS-compliant messages again.

On both layers, all received and sent messages from and to the system can simply and non-reactively be visualized using the `monitor` as described earlier, since all communication is wrapped into `Containers`. Furthermore, all data sent within a `ContainerConference` can simply be stored for further analysis using `recorder`.

8.2.2 Test Site "Richmond Field Station"

The Ford Escape Hybrid – ByWire XGV was tested on the "Richmond Field Station", a research and testing facility located about 6mi northwest of the University of California, Berkeley. For providing a digital map to the planning algorithm of the AGV on one hand, and to model the system's context on the other hand, the previously defined DSL was used.

Figure 8.12: Model of the system's context for Richmond Field Station projected on an aerial image; image credit: University of California, Berkeley.

In Figure 8.12, an aerial image of the Richmond Field Station with roads of the digital map is shown. For creating a digital map with an intuitional representation for the user, a clearly identifiable land mark from the northbound oriented image was chosen and set as WGS84 origin coordinate for the underlying coordinate system. For the Richmond Field Station, the coordinate (37°54'56.16"N, 122°20'5.14"W) was chosen. Using this reference point, 76 way-points marked as red in Figure 8.12 were recorded using a highly

precise GPS system. These coordinates where projected into a Cartesian coordinate system as already described in Section 4.2.3.

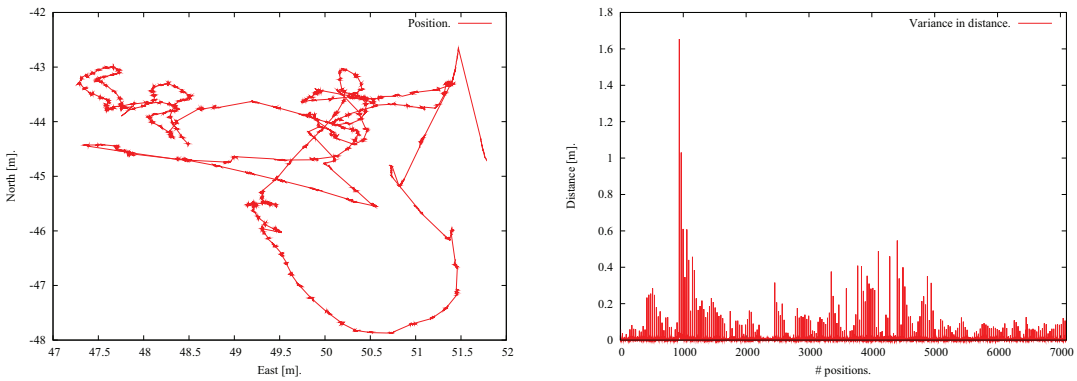
Grouping the recorded way-points to skeleton points of lanes, two-lanes roads forming a loop containing an intersection were created. Furthermore, these lanes where enriched with virtual lane markings indicating a yellow lane marking in the middle of the road forming a double yellow lane marking. The lane marking on the road's sides are simply defined as white.

8.2.3 Modeling the Ford Escape Hybrid XGV

In the following, general considerations for the modeling of the AGV Ford Escape Hybrid XGV are given to provide a reasonable model for the software development. Therefore, both a model of the vehicle's position and of the sensors to perceive the surroundings are presented.

8.2.3.1 Model of the Vehicle's Position

As already mentioned in Section 8.2.1, the vehicle's position, orientation, velocities, and accelerations are provided by a NovAtel GPS receiver combined with a NovAtel HG-1700 IMU. To derive an appropriate model to simulate this data for matching the reality by reducing its perfectly computed quality by artificial noise, the vehicle was placed on a parking spot with a good satellite's visibility to record the position provided by the IMU over a long period of nearly one hour while the vehicle was not moved at all.



(a) Absolute position reported by the IMU.

(b) Position's variance.

Figure 8.13: AGV's absolute position provided by the IMU over time and its variance.

In Figure 8.13a, the absolute vehicle's position is plotted. It can be easily seen that the absolute position varies between 47.25m and 51.75m in the East direction and between -48.00m and -42.50m in the North direction. Furthermore, the position's variance is between 0.01m up to 1.65m for an interval of 1s.

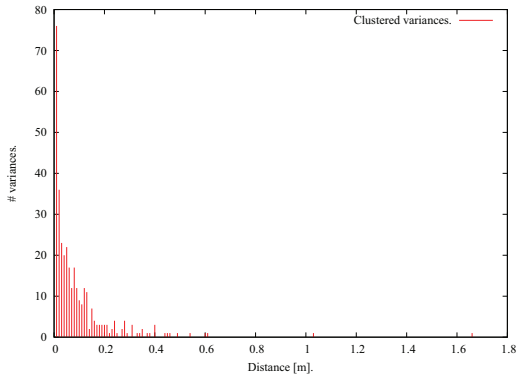


Figure 8.14: Clustered variances for the IMU provided positions.

After determining the IMU's variances for the position data, this data can be clustered to determine the frequency of every occurring variance. These clustered variances are shown in Figure 8.14. Nearly 24% of the position data varies about 0.01m during 1s; further 24% of the data varies up to 0.04m during 1s. Altogether, 70% is scattered up to 0.1m and 29% of the position varies between 0.1m and 0.5m. Only two outliers greater then 1m at 1.03m and 1.66m were recorded.

Using the data gathered from the IMU, the computed positions for the AGV in the simulation can be modified using the noise described above. Thus, the data gets a better relation to the reality. Further analysis might be carried out to model correlations between these variances but this modeling is out of scope for this example.

8.2.3.2 Model of the Vehicle's Sensors to Perceive its System's Context

Although the algorithm which is described in the following in Section 8.2.4 does not rely on information from the AGV's system's context except for GPS data, the model for its sensors to perceive its surroundings as shown in Figure 8.10 is outlined. As already mentioned, the XGV uses a stereo vision system combined with a single layer laser scanner to gather information from its system's context.

To simulate these sensors, the framework $\mathcal{H}$esperia provides models in `libvehiclecontext` as described in Section 6.4.5 and in Section 6.4.6. Thus, besides the position data, sensors' raw data for the stereo vision system as well as the

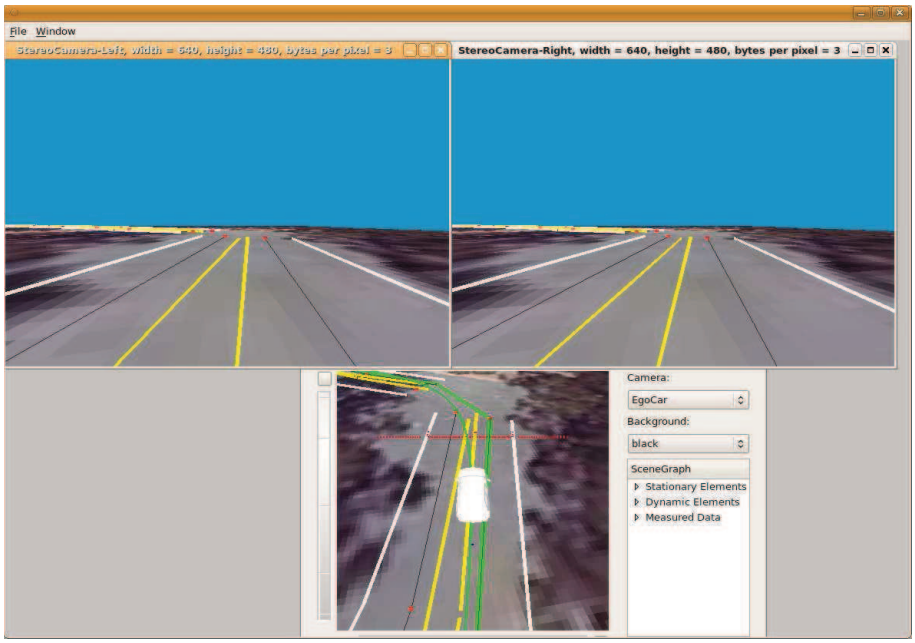


Figure 8.15: Non-reactive visualization of the Ford Escape Hybrid – ByWire XGV's sensors' model: In the upper area both raw images from the stereo vision system are shown. Below these images, the chasing camera for the AGV which is indicated by the white vehicle is shown. This camera is continuously following the vehicle's movements and thus, from this perspective, the scan line for the single layer laser scanner can be seen in front the car.

single layer laser scanner can be computed automatically to support both the interactive development as well as the unattended system simulation of perception algorithms. The non-reactive visualization of the sensors for perceiving the AGV's surroundings is shown in Figure 8.15 using the application `monitor`.

8.2.3.3 Performance of the Vehicle's Models for System Simulations

The performance of the selected AGV's models including the position provider but even for the providers which perceive the AGV's surroundings depends on various factors. First, the performance depends on the chosen number of sensors for a specific sensor type and the internal sensor model's complexity. For example, the computation of consecutive vehicle's position data is less complex compared to the simulation of a single layer laser scanner.

Following, the performance is limited by the complexity of the system's context. Using the position provider again the system context for computing the next vehicle's position depends only on the previous vehicle's state if a planar surface is assumed. When us-

ing a non-planar surface the computation is getting more and more complex apparently. However, the sensor model for vision-based sensors including the single layer laser scanner algorithm depends from the number and complexity of elements which are used in the OpenGL 3D scene. For example, this complexity depends on the number of triangles used to model a specific object in the scene. All these triangles including possible textures and lighting conditions must be rendered for one computation cycle before the specific model of the sensor can be calculated.

Furthermore, the performance also depends on the current computing platform on which all models are calculated. As mentioned in Section 6.4.6, some computing-intense algorithms are using not only the CPU but also the GPU to distribute the computation load. Moreover, the system simulation itself can be decomposed and distributed over several independent computing nodes as outlined in Section 6.4.2.

All aforementioned considerations must be regarded if the system simulation should be run interactively during the development. This is due to the independently running user-contributed application which is not under the control of the system simulation. Contrary to these interactively running system simulations, the aforementioned considerations does not need to be regarded in unattended system simulations for evaluating an SUD because not only the system's context but also the user-contributed applications are executed under the control and supervision of the system simulation as outlined in Section 6.3. Thus, the SUD and its system's context are independent for the real time and work entirely on the virtual system time. Therefore, the necessary computation time to perform one single step Δt_{sim} for even computing-intense algorithms may take a long time which means $\Delta t_{\text{sim}} > \Delta t_{\text{real}}$. But also the opposite case when the required models for the system and its system's context are less complex which is expressed by $\Delta t_{\text{sim}} < \Delta t_{\text{real}}$ can be realized resulting in a system simulation which is running faster than in real time.

8.2.4 Velocity and Steering Control Algorithm

In the following, a simple control algorithm implemented for evaluation purposes using the framework $\mathcal{H}$esperia is presented which was inspired by [155] but significantly modified to be used for urban environments containing sharp curves. First, some general consideration are discussed to outline the main idea behind the velocity and steering control algorithm. Following, variants and their effects of different interpolation techniques applied to a given set of way-points to be followed by the AGV are discussed. Finally, results of the velocity and steering control algorithm are presented gathered from test runs in the simulation using the $\mathcal{H}$esperia framework and from test runs performed by the real

AGV itself.

8.2.4.1 General Considerations and Design Criteria

The realized velocity and steering control algorithm is inspired by human driving. Empirically observed, humans drive by trying to minimize the lateral distance of a virtual but fixed point in front of the vehicle. This point is on the elongated vehicle's driving direction and thus, it is called *draw-bar*. This principle is the base for the steering control algorithm.

The velocity control algorithm bases on the same model but relies on the inverted effect: A decreasing distance for the fixed point in front of the vehicle results in an increasing acceleration because the vehicle's orientation is similar to the orientation of current route's segment and, thus, the vehicle can drive with a higher velocity. However, when the distance of this fixed point related to the current route's segment is increasing, the vehicle's velocity must be reduced because its orientation is getting more and more dissimilar to the segment's orientation. This controller has the following effects: When the vehicle is approaching a curve, it reduces its velocity and when it is leaving the curve it increases its velocity again after passing the curve's apex. In Figure 8.16, geometrical relations for both control algorithms are depicted.

As shown in Figure 8.16, the vehicle is simplified to a linear bicycle model with infinite tire stiffness [69]. In that figure, the path which the vehicle should follow is denoted by P. The vehicle's rotation in the world is denoted by ψ and the desired steering angle for the front wheels is named δ. The first draw-bar is named l_S and second one is named l_V. Both draw-bars has not necessarily the same length. Since it is desired that the AGV should reduce its velocity before it enters a curve, the latter should be longer than the first draw-bar l_S as shown in the figure. Both lengths are parameters which can be parametrized and are constant for the run-time.

As already mentioned, the idea behind the first draw-bar l_S is to determine the distance between the perpendicular point P_S on the path and D_S. This metric is used by the controller to compute the necessary steering angle δ as described in Equation 8.1.

$$\begin{aligned} \upsilon \tan(\mu) = x_S \quad &\underset{\upsilon>0}{\Leftrightarrow} \quad \tan(\delta - (\lambda - \psi)) = \frac{x_S}{\upsilon} \\ &\Leftrightarrow \quad \delta = (\lambda - \psi) + \arctan(\frac{x_S}{\upsilon}). \end{aligned} \tag{8.1}$$

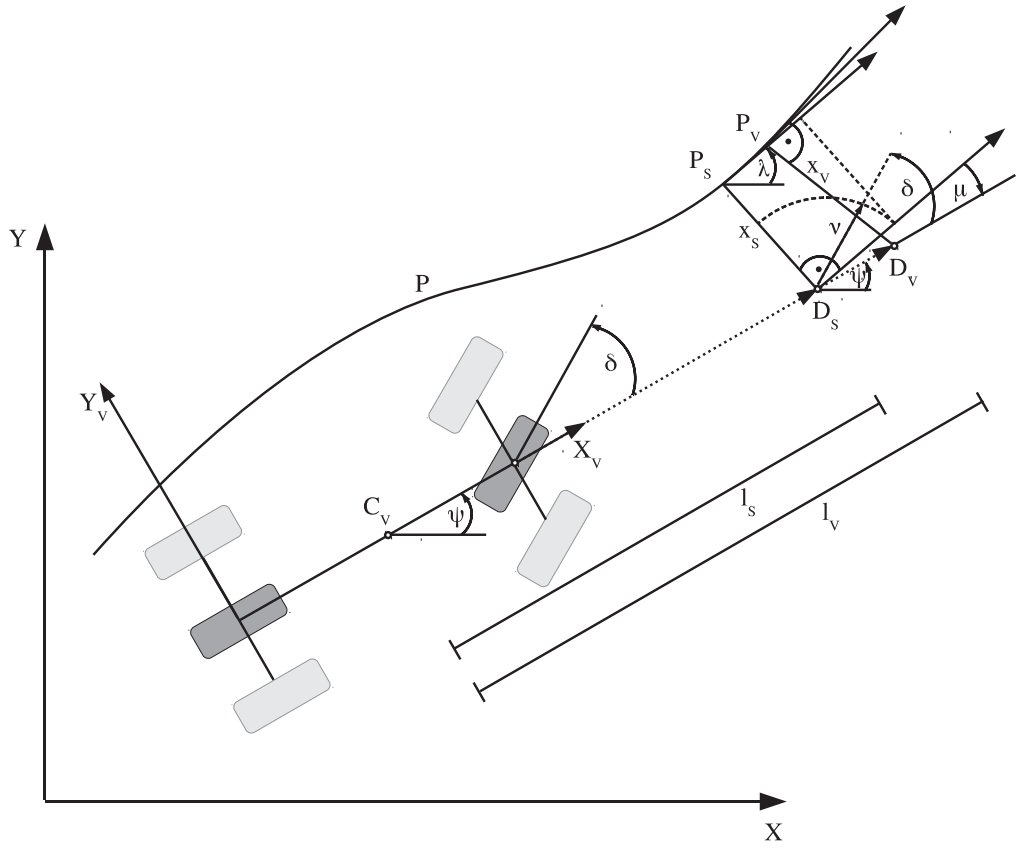


Figure 8.16: Geometrical relations for the control algorithm: The control algorithm based on two independent draw-bars. The first draw-bar called l_s is used for computing the distance x_s to the planned path P, while the latter has a greater distance to the vehicle and is called l_v for computing the distance x_v. The distance x_s is used to steer the vehicle depending on the distance, while the distance x_v is used to adjust the vehicle's velocity by reciprocally proportionally evaluating its value.

The angle ψ denotes the vehicle's rotation around the Z-axis and υ describes the vehicle's velocity, while λ denotes the orientation of the current path' segment containing the perpendicular point relative to the world. Its difference μ describes the relative delta between the vehicle's orientation and the current path' segment. To get the necessary steering angle to steer the vehicle towards the path, the angle μ can be used computing its tangent using the current draw-bar's distance to the path. Its final implementation is shown in Equation 8.2 where k describes the controller's gain.

$$\delta = (\lambda - \psi) + \arctan(k\frac{x_S}{\upsilon}), \text{ with } \upsilon > 0. \tag{8.2}$$

While the draw-bar point D_S should be minimized by the controller to steer the vehicle towards the path P, the distance of the draw-bar point D_V to its perpendicular point P_V on the path is used to control the vehicle's velocity. Therefore, the resulting velocity simply reciprocally proportionally depends to its distance: The greater the distance of D_V relative to P_V, the lower is the vehicle's speed. This is evident since an increasing distance describes an increasing curvature of the path. Thus, the vehicle lowers its velocity while entering a curve and increases the velocity again while leaving the curve after passing its apex. For reducing the vehicle's velocity right before a curve, the draw-bar's length l_V is greater than the draw-bar's length l_s for computing the steering angle. The final implementation is shown in Equation 8.3.

$$v = \begin{cases} v_{max}, & \frac{v_{max}}{\|\overline{D_S P_S}\|} > v_{max} \\ v_{min}, & \frac{v_{max}}{\|\overline{D_S P_S}\|} < v_{min} \\ \frac{v_{max}}{\|\overline{D_S P_S}\|}, & \text{otherwise.} \end{cases} \tag{8.3}$$

8.2.4.2 Computing and Optimizing a Route

The route is planned using an A$*$ search in a directed graph providing Euclidean distances as edges' weights which guarantees a resulting optimal route regarding the edges' weights if a route exists. The directed graph is built from the stationary surroundings data structure using a `LaneVisitor`. Thus, this visitor traverses the ASG and examines the current `ScenarioNode` whenever it encounters a `Lane`. Depending on the associated `LaneModel`, either the start and end point of an `Arc` or `Clothoid` are added as vertices to the graph connected by a directed edge containing the Euclidean distance between both coordinates. Moreover, if the visitor encounters a `Connector`, both semantically connected nodes are either added to the graph or an additional edge is inserted into the graph.

The graph itself is implemented using the *Boost Graph Library* wrapped and provided by the $\mathcal{H}$esperia framework. For generating the directed and weighted graph, only one single method `void updateEdge(const VertexData &v1, const VertexData &v2, const EdgeData &e);` is provided. Its implementation handles the insert of missing nodes or the update of existing edges as already mentioned in Section 5.3.2.5.

The resulting route consists of a list of vertices describing absolute positions in the world. This route can be further optimized which influences directly the controller's behavior which is discussed in Section 8.2.4.3. The different types of optimization are related to

the interpolation of the path segments between two points of the route. The effects of different types of interpolations are shown in Figure 8.17.

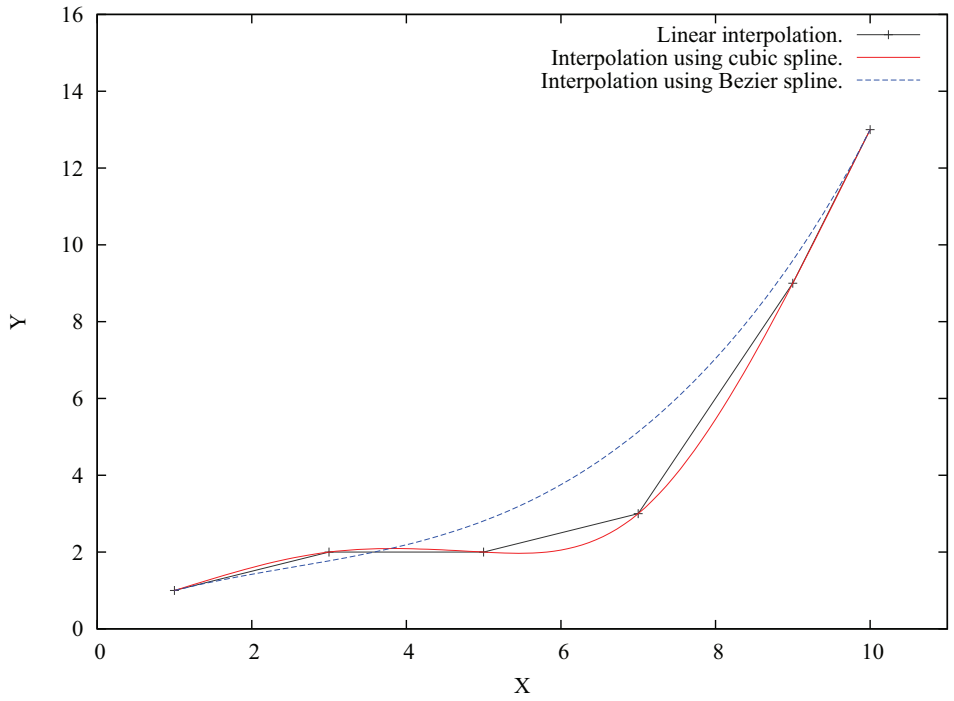


Figure 8.17: Effects for linear, cubic splines, and Bézier curves: While cubic splines pass all provided nodes, Bézier curves do not due to its definition.

In that figure, six arbitrary chosen way-points are connected using linear, cubic splines, and Bézier curves respectively which are described in detail in [8]. Obviously, linear interpolation is the simplest possibility to connect two points pairwisely using a straight line. Thus, actually no optimization is applied to the route.

$$f_{i-1}^{(1)}(1) = f_i^{(1)}(0) \wedge f_{i-1}^{(2)}(1) = f_i^{(2)}(0). \tag{8.4}$$

The next applicable optimization are cubic splines drawn as a red line in the figure. Cubic splines use a third order polynomial pairwisely applied to two knots. Cubic splines are C^1- and C^2-continuous as shown in Equation 8.4.

Thus, a cubic spline shows continuous behavior in its knots. Furthermore, the cubic spline intersects all given knots by definition as shown in Figure 8.17. Thus, the AGV would oscillatingly drive on the path optimized using cubic splines.

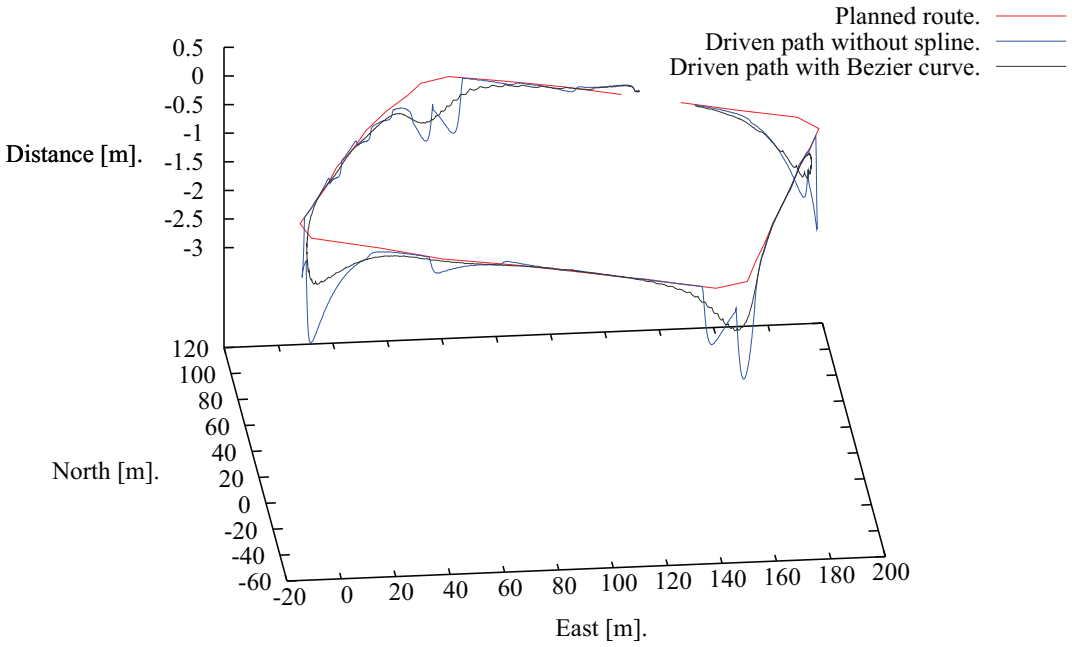
For avoiding this oscillating behavior, the route is optimized using Bézier curves instead. Bézier curves lay inside the knots' convex hull and thus do not necessarily continue through more than the first and the last knot. As shown in Figure 8.17, the Bézier curve only intersects the first and the last given way-point.

8.2.4.3 Performance in Simulation

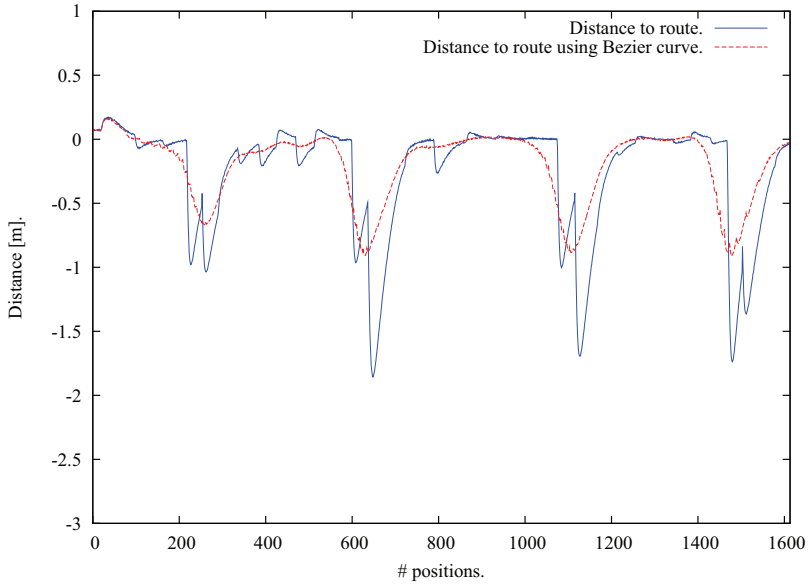
In the following, results of the velocity and steering control algorithm for simulated test runs running at 20Hz non-real-time are discussed. For all experiments, the velocity draw-bar had a length of 15m while the draw-bar for controlling the steering angle had a length of 5m. The position, orientation, velocity, and distance to the planned route for the steering draw-bar are plotted. The experiments were carried out for different velocities namely $v \approx 1.6\frac{m}{s}$, $v \approx 2.3\frac{m}{s}$, and $v \approx 3.0\frac{m}{s}$.

The following Figures 8.18, 8.19, and 8.20 show the results for different given velocities. All figures named (a) show the planned route with the red line which started in the northern part of the Richmond Field Station and followed counterclockwisely the outer lane for one round.

The solid blue line shows the driven path using linear interpolation with the distance to the planned route plotted in the Z-axis, while the dashed gray line shows the results for the route optimized using Bézier curves. All figures named (b) plot the distance to the planned or optimized route for the linear interpolation and the Bézier curve respectively.

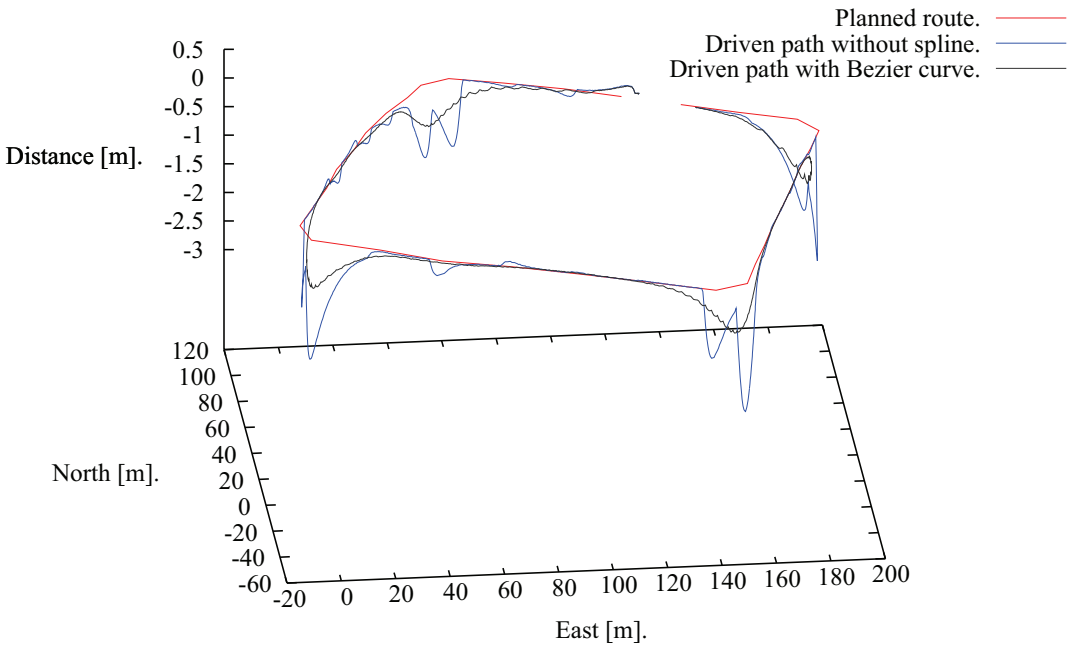


(a) Planned route with driven path for $v \approx 1.6\frac{m}{s}$.

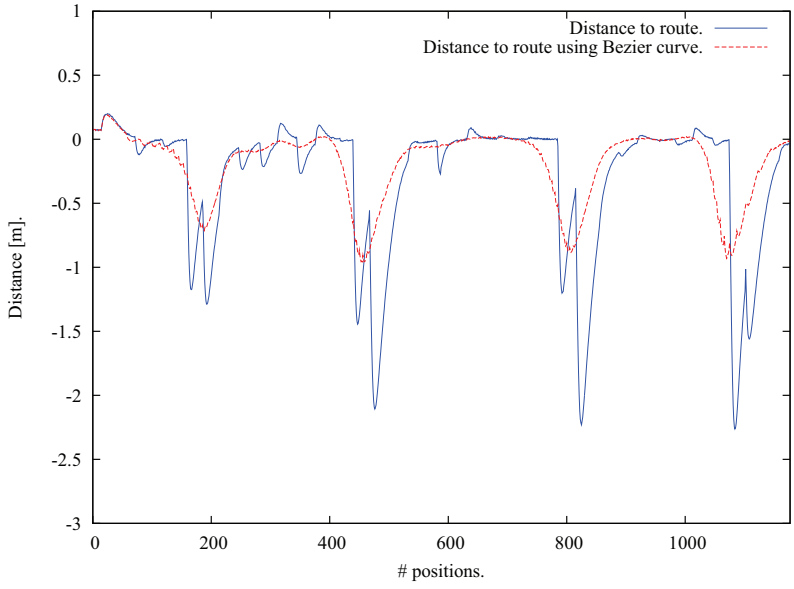


(b) Linear distance to planned path for $v \approx 1.6\frac{m}{s}$.

Figure 8.18: Performance of the draw-bar controller in the simulation for a velocity of approximately 1.6m/s.

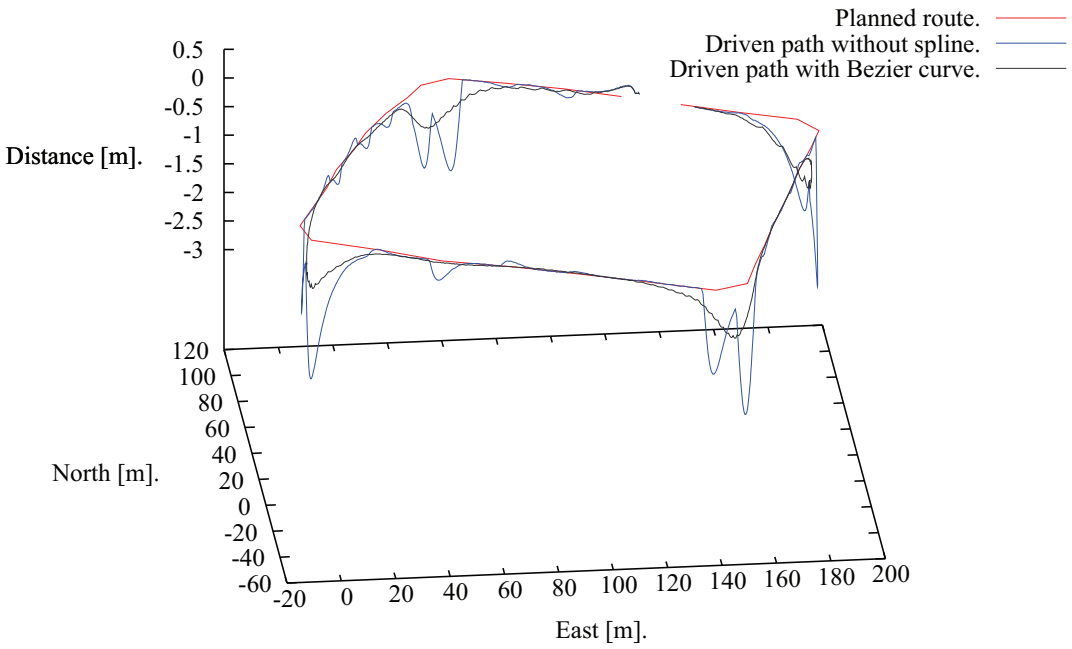


(a) Planned route with driven path for $v \approx 2.3\frac{m}{s}$.

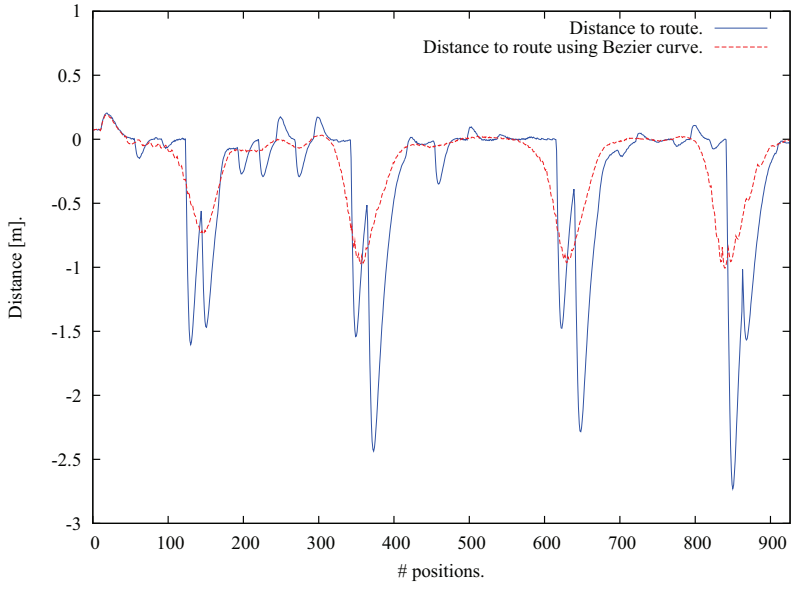


(b) Linear distance to planned path for $v \approx 2.3\frac{m}{s}$.

Figure 8.19: Performance of the draw-bar controller in the simulation for a velocity of approximately 2.3m/s.



(a) Planned route with driven path for $v \approx 3.0 \frac{m}{s}$.

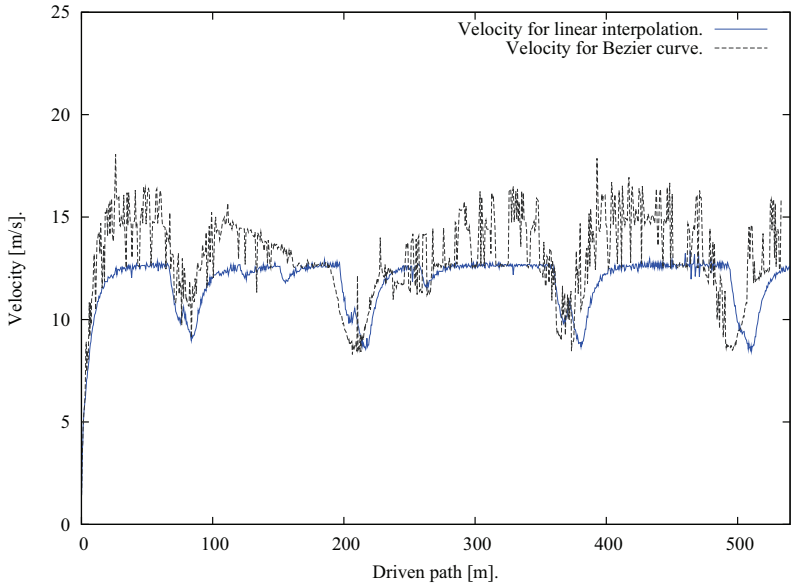


(b) Linear distance to planned path for $v \approx 3.0 \frac{m}{s}$.

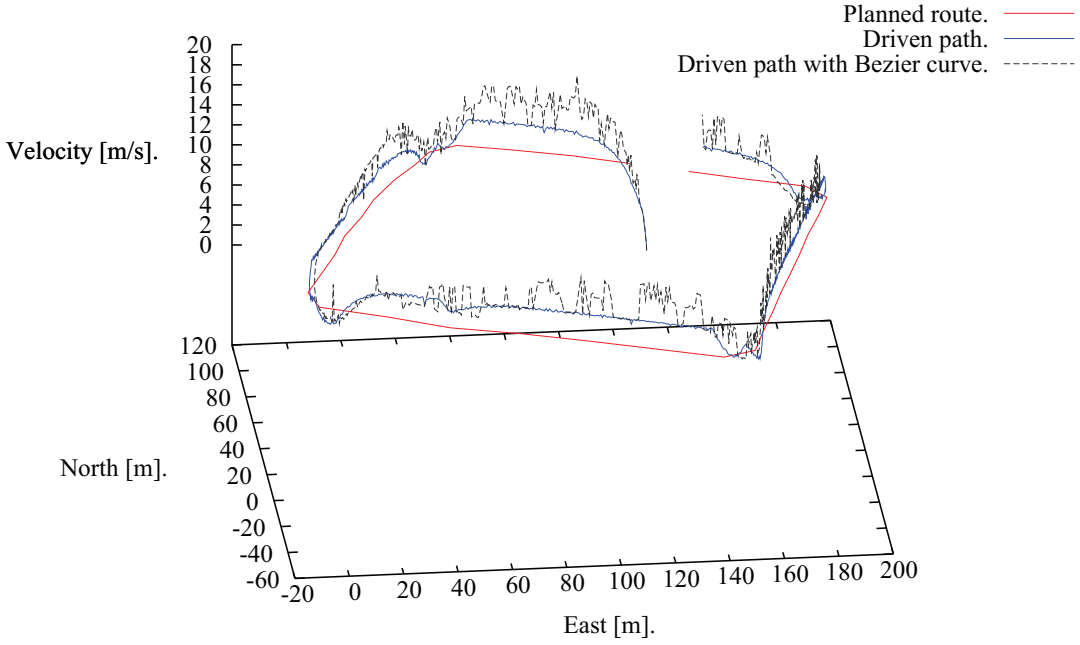
Figure 8.20: Performance of the draw-bar controller in the simulation for a velocity of approximately 3m/s.

First of all, the proposed velocity and steering control algorithm follows the planned route both for the linear interpolated case and the Bézier curve optimized route as well. Moreover, it can be easily seen, that the draw-bar steering control algorithm tends to minimize the distance $\overline{D_S P_S}$ as expected. However, with increasing velocities, the maximum error provoked by using non-optimized routes increases from -1.86m to $-2.73m$. Using Bézier curves, the maximum error increases from $-0.92m$ to $-1.01m$ only and thus is significantly lower.

In Figure 8.21, the results for the velocity control algorithm are shown. The figure on the left hand side shows the velocity profile plotted over the driven way. For the linear interpolated route, the corrections commanded by the velocity control algorithm due to changing distances in $\overline{D_V P_V}$ are nearly constant in the straight segments of the course. This is caused by the steering control algorithm which reduces the error in $\overline{D_S P_S}$ towards a straight line. Only in the curves of the track, the velocity is reduced significantly until the velocity draw-bar passes the apex; then, the velocity is increased again, which is shown in the plot on the right hand side showing the vehicle's velocity relative to the planned route shown as red line.



(a) Velocity profile over driven path for linear and B-spline interpolation for $v \approx 3.0 \frac{m}{s}$.



(b) Velocity relative to planned route for linear and B-spline interpolation for $v \approx 3.0 \frac{m}{s}$.

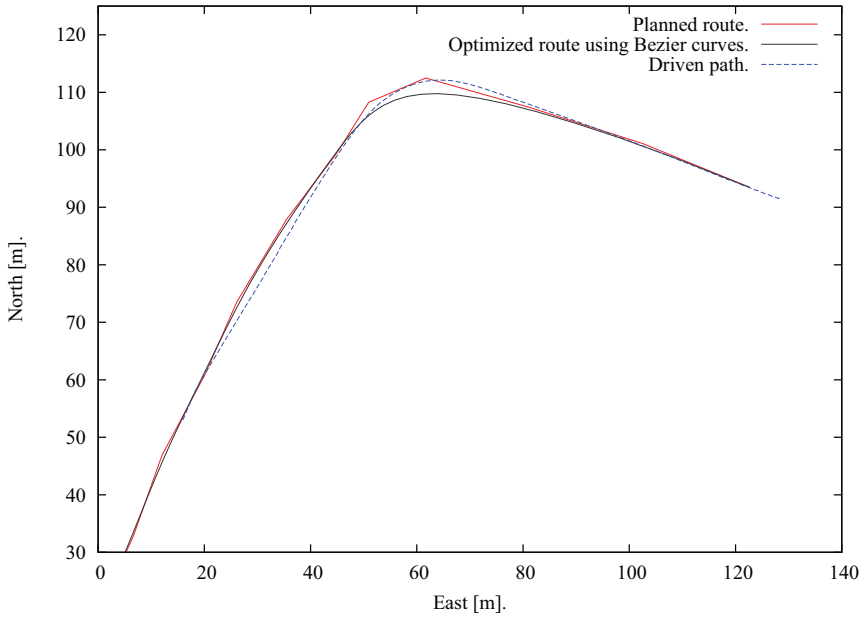
Figure 8.21: Performance of the draw-bar velocity controller in the simulation: The velocity of the vehicle is adjusted often due to a continuously changing distance x_v for the velocity draw-bar.

Contrary, for the Bézier curve optimized route, the steering control algorithm is continuously correcting the draw-bar's distance $\overline{D_S P_S}$ due to the curved course. The same applies for the velocity control algorithm which is continuously correcting the velocity depending on the distance $\overline{P_V D_V}$ of its draw-bar. However, the average velocity for the Bézier curve optimized route is $v_{\text{Bézier}_{avg}} = 2.542\frac{m}{s}$, while for the linear interpolated route, the average velocity is only $v_{\text{linear}_{avg}} = 2.328\frac{m}{s}$. The minimum velocities neglecting the initial ones representing the first acceleration are $v_{\text{Bézier}_{min}} = 1.684\frac{m}{s}$ and $v_{\text{linear}_{min}} = 1.702\frac{m}{s}$, respectively; the maximum velocities are $v_{\text{Bézier}_{max}} = 3.614\frac{m}{s}$ and $v_{\text{linear}_{max}} = 2.67\frac{m}{s}$. Thus, with the proposed algorithm and the Bézier optimization, an approximately 9% higher average velocity results for the AGV; moreover, the maximum velocity is also nearly 35% higher than in the non-optimized variant.

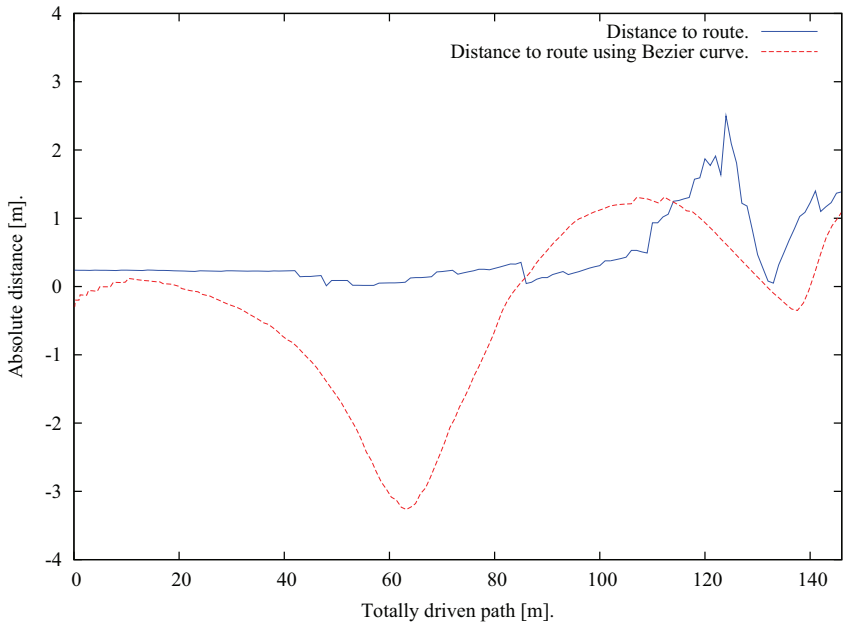
Altogether, the aforementioned plots were derived interactively using the software framework $\mathcal{H}$esperia. Thus, the quality of the integrated velocity and steering control algorithm could be evaluated interactively to support its development and for optimization. In the following, this algorithm was applied to the real vehicle.

8.2.4.4 Performance in Reality

The aforementioned velocity and steering control algorithm was evaluated on the AGV Ford Escape Hybrid XGV with a slightly higher velocity at $v = 4.1\frac{m}{s}$, with a length of $7m$ for the steering draw-bar and a velocity's draw-bar's length of $15m$ as in the simulation. Its performance running at 20Hz non-real-time is shown in Figure 8.22.



(a) Performance of the velocity and steering control algorithm in reality for $v \approx 4.1 \frac{m}{s}$: The red line interconnects the single way-points of the road network and the black line depicts the optimized route using a Bézier curve. The finally driven blue line is the actual position of the AGV in reality.



(b) Linear distance relative to the planned route for the velocity and steering control algorithm in reality for $v \approx 4.1 \frac{m}{s}$.

Figure 8.22: Performance of the velocity and steering control algorithm in reality.

In the figure on the left hand side, the red line shows the way-points of the planned route connected by straight lines, while the black line shows the optimized route using a Bézier curve. The dashed blue line shows the position of the AGV itself relative to the planned and optimized route which follows the initially planned and optimized route.

As shown in Figure 8.22b, it can be seen that the AGV follows the optimized route with a minimum error of $-0.001m$, an average error of $0.3m$, but with a maximum error of $1.31m$ to the left hand side and a maximum error of $-3.27m$ to the right hand side of the route optimized using Bézier curves. Despite the maximum errors seem to be that high, the AGV follows pretty well the initially planned route as shown in Figure 8.22b by the blue curve. Altogether, the algorithm itself could be applied successfully to the real vehicle using the software framework $\mathcal{H}$esperia. It can be further optimized by adjusting the vehicle's model which is used in the simulation to get a more precise behavior for the vehicle in the simulation compared to the reality; however, these optimizations are not in the scope for this thesis whereas the actual *unattended* system simulation and evaluation are of substantial interest. Therefore, for preserving the algorithm's quality without testing it over and over again its evaluation shall be automated as already outlined in Section 7.3. This final step is described in the following.

8.2.5 Automating Test Runs

For assuring the algorithm's quality, the concept of an automatic and unattended test drive shall be applied. Therefore, several *reporter*s are necessary to define which are running unattendedly, continuously, and automatic during a system simulation for continuously evaluating the SUD. For the example, the SUD is the aforementioned steering and velocity control algorithm and its system's context is represented by the simplified bicycle model as mentioned in Section 6.4.3.

The first reporter is `DestinationReachedReport`. The main goal of the class is to evaluate whether the vehicle has reached an arbitrarily defined destination point. Therefore, this class uses the visitor `FindNodeByPointIDVisitor` to traverse the ASG describing the stationary surroundings to retrieve the node containing all data about the given destination identifier. Using this node, its coordinates are used to compute its distance to the current `EgoState` provided by `ControlledContainerConferenceFactory` using the `SystemContextComponent`'s `ContainerListener`. If the computed distance is less than a given and constant threshold specified at construction of the `DestinationReachedReport`'s instance, this reporter returns finally true when

its method `bool hasReachedDestinationWaypoint() const;` is called; otherwise, false will be returned.

The next reporter is used to evaluate if `simpledriver` not only chooses but also drives along the shortest route to given destination point from a given start point. Therefore, an instance of the class `ChoosingShortestRouteReport` uses the ASG to retrieve all `Layers`, `Roads`, `Lanes`, and `LaneModels` to construct a graph representing a digital map. Using this map and the start and destination way-points, the shortest route is determined using the A*-algorithm resulting in a list of coordinates retrieved from the digital map. Comparable to the previous reporter, an instance of this class continuously compares the current valid `EgoState` with the list of coordinates representing the order in which the route must be driven. Every time the coordinate at the head of the list is successfully passed by computing a distance which is less than a given and constant threshold, it is removed from the list and its successor is used for further evaluations. Finally, this reporter would return true if all coordinates were passed in the correct order with a distance less than the specified threshold.

Following, the next reporter called `DistanceToRouteReport` is used to determine continuously the distance of the vehicle to the chosen route. Therefore, comparable to the previous reporter, this reporter computes the route between a given start and destination way-point. Furthermore, it uses two consecutive coordinates from which the second one must be in front of the vehicle to compute a straight line. Using the current position provided by `EgoState` which derives from `Position` describing any point-shaped object with a position, orientation, velocity, and acceleration, its perpendicular to this straight line is computed. As long as the distance between the vehicle's position and its perpendicular with respect to the route is less than a given and constant threshold, the reporter would finally return true indicating that the vehicle's distance was alright at any time during the simulation of the system's context.

As already mentioned in Section 6.3, these reporters are simply registered at the `RuntimeEnvironment` which contains all instances to be scheduled and executed during a simulation run. Since all reporters return a Boolean result in their simplest case, the concept of `RuntimeControl` and `RuntimeEnvironment` enabling unattended simulation runs combined with the concept of the aforementioned reporters can be easily used with the well-known unit test environments like CxxTest [156]. Moreover, these unit tests and the entire software build process were set up for an automated software build using a *continuous integration system* like CruiseControl [38]. Therefore, every modification made to the software, and especially to the source of `simpledriver`, was evaluated to the test cases specifying virtualized test runs.

8.2.5.1 Conclusion

According to proposed methodology in Chapter 3, an algorithm which processes continuous input data for a sensor- and actuator-based autonomous system was specified, developed, and applied to its real hardware environment. Furthermore, for preserving its quality for further modifications, extensions, or optimizations the algorithm was embedded into automated system simulations without any modifications by using the concepts provided by the framework $\mathcal{H}$esperia. Because these system simulations are conceptually similar to unit tests, they could be automated using recent continuous integration systems. However, due to the fact that the evaluated algorithm requires complex and especially continuous input data from a complex system's context which are generated from the specified DSL as outlined in Section 4.4.

However, considering instable position information provided by the IMU as well as for velocities which are higher than the evaluated $v \approx 4.1\frac{m}{s}$, more information from the surroundings like lane markings or road boundaries like curbs should be used to safely realize velocities necessary for urban environments. Though, this is out of scope for this thesis.

8.3 Further Applications of the Framework $\mathcal{H}$esperia

In the following, the applicability of the framework $\mathcal{H}$esperia as well as some tools realized using the framework are outlined for the development of systems consisting of sensors and actuators.

8.3.1 Sensor Data Collection

One of the most commonly used tasks during the development of systems using sensors for perceiving their surroundings is sensor data collection using all mounted sensors. This task can be supported easily by defining a data structure which can be serialized using a `Container` and which describes the sensor data. Additionally, a component must be provided which is continuously reading the sensor's raw data and filling the data structure. Depending on the amount and frequency of the sensor's raw data, the data can be written directly to disk or broadcasted into a ClientConference to be recorded by `recorder`.

For later analysis, this data can be replayed easily using `monitor` to inspect the data visually or for further processing using `player`. For the former, a device-independent data visualization as described in Section 5.4.5 must be provided which can be done

easily since most sensors either provide points or contour data as raw data; for cameras, `SharedImages` can be directly used. Using the tools mentioned in Section 5.6, the data can be replayed either continuously or stepwisely.

Since synchronous data is inherently important for this task, all used computers must be synchronized before recording the data to get correct timestamps for the captured data. Since the framework *Hesperia* serves only as communication framework for this task using the internal computer's clock for time stamping the data, time synchronization must be setup before collecting data. For synchronizing computers using software solutions, NTP or PTP as already mentioned in Section 8.1.2.2 are available for example.

8.3.2 Virtual Sensor Data Collection

Comparable to the sensor data collection, the system consisting of different sensors and actuators can be virtualized as described in Section 8.2.3. Providing the customer's scenarios formally specified using the DSL described in Section 4.4.1, a virtual system context can be generated wherein the system to be developed can be freely placed. Using components from the system simulation as outlined in Chapter 6 for generating the required sensor's raw data, the same tools as mentioned before can be used to record and replay the captured data for further inspection or processing.

Depending on the use case, time synchronization can be necessary. If the virtual sensor data collection is made interactively or while using several independent computers, all computers involved must be synchronized to use the same valid system time as already mentioned in Section 8.1.2.2. Otherwise, if the data collection is purely virtually made using unattended test runs for example, the communication as well as the time for the running system is controlled entirely by `libcontext`. Thus, the data collection itself is independent from the real time.

8.3.3 Application-Dependent Additive Sensor Data Generation

Having previously captured system's data, this data can be easily enriched by additional sensor's raw data to generate a set of different data collections providing different subsets of available sensors. Therefore, the previously recorded data is replayed using `player` while additional sensor's raw data is generated using e.g. a virtual camera or a single layer laser scanner. The resulting data is recorded again using `recorder` for producing an enriched data collection. For example, this feature is relevant for evaluating different sensor's mounting positions or sensor variants while using the same test drive.

8.3.4 Evaluation Runs

In the following, the applicability of the framework $\mathcal{H}$esperia and the techniques developed and described before for different use cases of evaluations for the case of automotive software engineering are shown.

8.3.4.1 Situational Evaluations

The most obvious use case are *situational evaluations*. Using the framework $\mathcal{H}$esperia, different situational evaluations for the traffic can be evaluated. The evaluation can be made on one hand purely virtual in the simulation for the system's context at different levels of details providing both low-level sensor's raw data and high-level data structures using an abstract description of the surroundings. Moreover, on the other hand, the evaluation can be made risklessly while saving valuable resources using virtual sensor's raw data while running applications realized with the framework $\mathcal{H}$esperia on the real sensor- and actuator-based autonomous system in reality. Thus, complex or even system context's risky situations can be evaluated repeatedly with identical conditions.

8.3.4.2 Alternate Sensor Configurations

Comparable to the aforementioned evaluation, identically repeatable situations can be evaluated for different mounting positions for one or several sensors. Furthermore, different types or amounts of sensors can be evaluated to explore the best sensor setup for fulfilling the customer's requirements for the SUD's behavior in its intended system's context.

8.3.4.3 Sensor- and Actuator-based Autonomous System in the Loop

Completing these evaluation runs regarding different but identically repeatable system context situations, planning or control algorithms can be evaluated safely on a real vehicle at different stages. On one hand, real sensor data can be used for deriving decisions to be realized by the system's actuators. But instead of using the real actuators, the system is modified accordingly by an operator. On the other hand, virtual sensor's raw data can be provided for the algorithms for evaluating the system's real actuators.

8.3.5 Illustrating a Sensor- and Actuator-based Autonomous System's Performance

Another non unusual task is to illustrate the autonomous system's performance for documentation or presentation tasks. This task can be easily supported by the framework *Hesperia* using the concept of device-independent data visualization. Since the same concept is also used for realizing the non-reactive data visualization as described in Section 7.2, it is simply reused by `rec2video` to implement a tool for rendering a sequence of images from a given autonomous system's recorded data. These images can be rendered into a video file with a desired quality.

9 Related Work

In this chapter, related work for this thesis with a focus on currently available frameworks for developing distributed component-based embedded software is presented. Then, development and test environments especially for automotive software are outlined. For both aspects, the supplied documentation for a specific solution was mainly used.

9.1 Frameworks for Distributed Component-Based Embedded Automotive Software

In the following section, a brief selection of available programming frameworks for distributed component-based embedded software is presented. The frameworks are evaluated regarding the following aspects:

- *Compliance to standards.* Each framework should rely on standards for use on different hardware or different operating systems.
- *Provision of usage patterns.* Each framework should provide so-called "best-practice" usage patterns to enforce a similar component design on source code level. Furthermore, running applications should be decoupled or only loosely coupled to avoid a priori knowledge about communication dependencies.
- *Support for non-reactive communication inspection.* Each framework should offer possibilities to inspect component communication for on-line monitoring of the running system or offline playback for previously recorded data to support the developer's work and to realize system evaluations.

9.1.1 Elektrobit Automotive GmbH: Automotive Data and Time Triggered Framework

Elektrobit Automotive GmbH provides the Automotive Data and Time Triggered Framework (*ADTF*). This framework can be used to create applications consisting of software

components which act as filters which operate on incoming time-stamped data streams to produce output streams. Hereby, every filter defines a set of typed input- and output pins. Every input pin can read data from one specific data type called `IMediaSample` identified by a defined `MediaType` and its `MediaSubType`. For example, let the former describe a video type, then the subtype denotes whether it contains compressed or uncompressed data. For transmitting data, a filter creates the filter-specific subclass derived from `IMediaSample` and serializes the data.

For convenience, a Graphical User Interface (*GUI*) supports the configuration of all desired filter components. Connected input- and output-pins from several filter components represent a directed graph called *filtergraph*. All filters can be loaded at run-time using a plug-in concept realized by the ADTF run-time kernel. The run-time kernel itself is responsible for the actual execution and the scheduling of a given filtergraph. Therefore, it manages the filter's state machine consisting of `Init`, `Start` for starting the stream processing, `Stop` for stopping the data processing, and `Shutdown`. The entire design of the ADTF is similar to Microsoft Component Object Model (*COM*), and its filter chain concept is inspired by Microsoft DirectShow.

The ADTF itself can be run on x86 hardware and can be used with Microsoft Windows, Ubuntu 7.04, and openSUSE 10.3. The framework provides access to different data sources: CAN, Flexray, Media Oriented Systems Transport (*MOST*), and Local Interconnect Network (*LIN*) using hardware provided by third party suppliers, TCP, UDP, cameras using either Microsoft DirectShow, BlueFox, or Video4Linux, and audio streams [46, 47, 132].

The main communication concept of the ADTF bases on unformatted data transfer for serialized data structures between components of a filtergraph. Thus, the `IMediaSample` is the generic type which is used to realize data exchange between components of the filtergraph. Besides, the ADTF provides meta-information for input data which is read from the supported hardware like CAN or cameras. However, no further data structures which support for example the description of the system's context on an abstract level which is necessary for intelligent algorithms to evaluate an SUD's system's context are provided. Obviously, this is not the intended scope of the ADTF because it provides an application-independent approach and therefore, a generic data exchange is provided which must be adapted for a specific use case.

Due to the explicit definition of synchronized streaming connections between several software components using the specified input- and output-pins, a non-reactive communication inspection is not possible because the inspecting component itself is directly part of the resulting application. Furthermore, the inspected preceding components take directly

notice from the inspecting component. Non-reactive monitoring about the exchanged data is only provided for gathering statistical information about the throughput of sent `IMediaSample` for example.

9.1.2 AUTOSAR

The AUTOSAR specification is intended to define a standard for software and system architecture for vehicles covering not only technical implementations but also the software and system development process in general to tackle the increasing number of software-intense ECUs. AUTOSAR consists of a multi-layer architecture, whose lowermost layer abstracts from the micro-controllers, whereas the second layer abstracts an entire ECU. On top of the ECU abstraction layer, a services layer provides state management for the ECU itself. Finally, the Run-time-Environment (*RTE*) integrates all aforementioned layers and provides rudimentary communication. Therefore, AUTOSAR supports synchronous and thus blocking 1:1 client/server communication as well as anonymous and asynchronous 1:n sender-receiver communication.

Furthermore, it realizes a so-called Virtual Functional Bus (*VFB*) concept abstracting all communication interconnections between running software components using the RTE. The main goal is to allow software and system development for automotive software components regardless to the final implementation [3, 51, 91].

Since the main focus of AUTOSAR is on the software and system development process for ECUs in general, it does not provide special support for sensor- and actuator-based autonomous systems at all or for their development. Thus, AUTOSAR could be used as rudimentary communication and abstraction layer, however, experiences using AUTOSAR for systems based on sensors which produce large amounts of raw data are missing.

9.1.3 OpenJAUS

OpenJAUS is an open source implementation of the Joint Architecture for Unmanned Systems (*JAUS*) specification initiated by Defense Advanced Research Projects Agency (*DARPA*) [89]. The main goal behind JAUS is to create autonomous systems for air, ground, water, and subsurface consisting of components which themselves are provided by different third party suppliers. Thus, this goal is comparable to the one of AUTOSAR.

A system designed and realized following the JAUS specification consists of several independent subsystems called nodes and controlled by a `NodeManager`. The

`NodeManager` is responsible for managing a node, providing information about available services on a specific node and for routing messages between different subsystems. Every node can run several different components which either acquire data from sensors, control actuators, or process data. From a specific component, several instances can be available at run time. Therefore, every instance must have a system-wide unique JAUS software address consisting of the subsystem identifier, the node identifier, the component identifier, and finally the instance identifier.

The JAUS software addresses are used by the `NodeManager` to route incoming and outgoing `JAUSMessages` between running instances. Messages are sent either upon an instance's specific request or periodically which is called a service connection. The JAUS specification itself does not specify the media to be used for communication. Thus, OpenJAUS currently uses UDP for its communication. Furthermore, the JAUS specification defines a set of messages both to query information about a running system based on JAUS and to control autonomous systems in general. In its most recent version, JAUS provides more than 150 messages ranging from information about the current velocity of an autonomous system to steering commands for controlling a running system.

Besides the OpenJAUS library, the JAUS Toolkit can be used as an extension for National Instruments LabVIEW [158]. This extension allows design and development of JAUS-compliant applications by easily dragging and dropping graphical elements wrapping JAUS messages. Thus, the entrance in the development of JAUS-compliant applications is simplified.

Despite its proven applicability in the 2007 DARPA Urban Challenge [5] and its standardization by Society of Automotive Engineers (*SAE*), the main problem of JAUS is its main goal at the same time. Enforcing compatibility with several independent third party suppliers means to rely on a formally defined and fixed message set. Extensions to this set are only possible by defining new messages with new identifiers duplicating the message to be refined or to define new messages unknown to other JAUS-compliant systems per definition. However, compared to the approach presented in Section 5.4.3 which provides a DSL to describe and maintain data structures in the software framework $\mathcal{H}$esperia, the messages used by JAUS must be maintained and extended manually.

Furthermore, due to its goal providing a library to be used with C and C++, the implementation provided by OpenJAUS does not use modern object-oriented concepts but realizes all messages using only C-style structs repeating identical code for every message. This limitation can be avoided using an alternative implementation of the JAUS specification provided by [163]. However, the concept of messages which are centrally routed by a `NodeManager` per node requires that messages which should be sent to instances run-

ning on another node must be sent to the local running `NodeManager` first which sends the message to the remote `NodeManager`. The remotely running `NodeManager` finally delivers the message to the required component's instance. Thus, additional latency is caused using this concept.

9.1.4 Orca/Hydro

Orca/Hydro is an open source framework for realizing component-based software development for robotics supporting C++ and Java which provides some algorithms for developing experimental robotics platforms. Orca itself relies on Internet Communications Engine (*ICE*), a framework for distributed communication for today's major programming languages [80]. ICE realizes optionally compressible communication using TCP and UDP and provides Specification Language ICE (*SLICE*) to specify component's interfaces. Furthermore, Orca uses ICE's `IceGrid Registry` to share service names among independent processes, `IceBox` for realizing the component's state machine, and `IceStorm` to broadcast published messages between several subscribers.

Orca provides a simplified API to its underlying ICE communication framework. This wrapper API allows the deployment of a user developed component either as a standalone application or as part of the aforementioned `IceBox`. Orca can be used with Linux, experimentally with Microsoft Windows, and with QNX.

Hydro offers drivers for reading data from a camera using OpenCV [25], a Global Positioning System (*GPS*) device or a single layer laser scanner. Furthermore, some algorithms for deriving disparity maps from stereo vision images as well as simple path planning algorithms are provided. Furthermore, a simple interface to the experimental robotics development suite Player/Stage/Gazebo as described in Section 9.2.7 is provided. Orca/Hydro was experimentally applied to an autonomous vehicle by a team in the 2007 DARPA Urban Challenge [100].

As directly stated by its design goals, Orca itself does not provide any architectural usage patterns besides the ones wrapped from ICE itself. Thus, along with Hydro, a similar technical approach to the filtergraph from the ADTF is provided combining components from both packages. However, ICE causes a similar additional latency for routing at runtime like the `NodeManager` used by OpenJAUS.

9.1.5 Evaluation of the Frameworks for Distributed Component-Based Embedded Automotive Software

In Figure 9.1, a qualitative evaluation of the aforementioned frameworks is provided. In this table the last three columns reflect one of the criteria mentioned above: *Compliance to standards*, *Provision of usage patterns*, and *Support for non-reactive communication inspection*. Every framework was qualitatively evaluated using the scale *high*, *medium*, and *low* which is denoted by the corresponding amount of black squares.

Framework	**Compliance to standards**	**Provision of usage patterns**	**Support for non-reactive communication inspection**
Automotive Data and Time Triggered Framework	■■■	■■■	■■
AUTOSAR	■■■	■■	■
OpenJAUS	■■	■■	■
Orca/Hydro	■	■■	■

Figure 9.1: Qualitative evaluation of the frameworks for distributed component-based embedded automotive software: *Automotive Data and Time Triggered Framework*, *AUTOSAR*, *OpenJAUS*, and *Orca/Hydro*.

A classification as *high* reflects that the considered framework fulfills to a large extent the given requirement; *medium* describes that only some aspects of a given criterion are fulfilled, while *low* means that the framework does not fulfill or only limitedly fulfills a requirement. Further information about every framework is provided within the respective section.

9.1.6 Other Robotics and Communication Frameworks

Besides the previously described frameworks which focus either directly on automotive software engineering or which have been applied to autonomous vehicles, several other tool-kits and frameworks addressing different aspects on robotics are available. Some of them are outlined briefly in the following.

A framework for realizing real-time communication using different communication patterns is Middleware for Robotic and Process Control Applications (*MiRPA*). The main focus of MiRPA is on real-time communication for distributed embedded software. Thus, it provides synchronous and asynchronous client/server communication as well as synchronous publisher/subscriber communication [53].

Another framework is Carnegie Mellon Robot Navigation Toolkit (*CARMEN*) written in C with Java support. This toolkit is intended to support the development of experimental robotics platforms running under Linux and consisting of GPS, sonar devices, infrared devices, and single layer laser scanner devices [34]. It is similar to Player/Stage/Gazebo as described in Section 9.2.7.

Aiming to provide a meta-operating system for any robotics platform, Willow Garage provides Robot Operating System (*ROS*) [176]. In general, ROS provides communication facilities, message sets, and data storage for graph-like, distributed processing nodes comparable to the ADTF for experimental robotics platforms. For supporting a wide range of robotics platforms, its strength is the abstraction from concrete sensors and actuators. This enables the reuse of algorithms for perceiving the robot's surroundings or for planning and motion control.

In [95], a communication framework is presented which implements a modified subset of the JAUS specification. The main focus is on optimizing the communication within a node using Inter-Process Communication (*IPC*), while the inter-node communication is realized using TCP. Thus, on one hand the communication inside a node could be improved and monitored using a watchdog supervising running processes. On the other hand however, this communication framework is neither compatible to OpenJAUS nor to the JAUS specification.

Common Object Requesting Broker Architecture (*CORBA*) is a specification for developing distributed heterogeneous applications. Using its Interface Definition Language (*IDL*) for specifying formally data structures and method signatures and an Object Request Broker (*ORB*), general purpose programming languages like C++ or Java can use the generated interface definitions to call remotely available objects. An open source real-time-capable implementation of CORBA is available by *The ACE ORB* [133]. Limited to Microsoft Windows only, Microsoft COM is comparable to CORBA [121].

Another approach for communication between distributed components is provided by [101]. This library, realized in C, can be used to generate concrete transmittable data structures using UDP multi-cast for C, Java, and Python using a data definition language. Although, this library originates from the team MIT's contribution to the 2007 DARPA Urban Challenge and thus proved its applicability for automotive software, no ready-to-use data structures for the context of autonomous ground vehicles are provided; contrary to the data definition language provided with $\mathcal{H}$esperia as described in Section 5.4.3, the data definition language of that library cannot be used to describe inheritable data. Furthermore, this framework transmits data using named channels wrapped around UDP multi-cast for one specific data type instead of typed messages. Thus, the sender and re-

ceiver must ensure to send only data from one specific type over one named channel to avoid malfunctions at the receiver.

In [48], some rudimentary data structures for easing inter-component communication are provided instead of a communication framework. Amongst others, data structures for describing the time, position, and orientation are provided. Furthermore, basic operations on these data structures like rotations or coordinate conversions are available.

Compared to the data description language as presented in Section 5.4.3, some similar frameworks are available. Google is using their own implementation called Protocol Buffers [72]. This toolkit provides a high-level language to describe serializable data. However, to avoid problems caused by evolving data structures, the developer must not change the so-called tag numbers for an attribute or add further required fields. The approach presented in this thesis provides a transparent concept to the user to avoid any misuse. Another similar approach is provided by bdec [13]. This tool requires a user-supplied XML specification from which a specific decoder for binary data is generated automatically. Contrary to the approach outlined here, that tool cannot generate encoders; moreover, using XML for language specification results in a less compact description. Additionally, the data description language as outlined in this thesis is not only applicable and usable with the software framework $\mathcal{H}$esperia. Instead, due to the concept of modular decorators which traverse the ASG to generate desired language-dependent data structures, these decorators can simply be extended to use this language in other contexts.

9.2 Software Development and System Testing

In the following section, a selection of currently available development environments for automotive software systems is presented. These systems are evaluated for the following aspects:

- *Supporting a virtual development process.* For reducing dependencies on real hardware on one hand and to provide identical development environments for all developers on the other hand, the development environment should virtualize the system to be developed including all of its necessary components like sensors and actuators. Additionally, the context of the system must be available for every developer to test the system's reactions on stimuli from the system's context.
- *Supporting the integrated development of low-level and high-level algorithms.* Due to increasing complexity in embedded systems caused by integrated low-level algorithms for control and high-level algorithms for perceiving and assessing the

system's context which use complex data structures, the development environment should support the realization of both kinds of algorithms.

- *Integration of the development and testing environment.* Supporting the aforementioned integrated development of low-level and high-level algorithms in a homogeneous manner, the testing and development environment should be integrated allowing both tests of single parts of the system under development and tests of the entire system without the need for different tools with different interfaces.

9.2.1 Driving Simulator at Deutsches Zentrum für Luft und Raumfahrt (DLR)

The Deutsches Zentrum für Luft und Raumfahrt (DLR) operates a hexapod driving simulator allowing to test driver assistance systems and their impact on the driver. Therefore, a real car can be mounted inside a cabin which itself generates the movements and accelerations depending on the driver's input. The vehicle's surroundings is projected 270° in front of the car allowing a realistic presentation. Thus, this driving simulator can be classified as Driver-in-the-Loop (*DiL*) simulation.

Contrary to the approach presented in this thesis, the system's focus is on preparing real vehicle test drives for driver assistance system in late stages of the development process. Therefore, it is rather inapplicable for early stages in a system development process demanding a virtualized interactive and unattended test environment.

9.2.2 Framework for Simulation of Surrounding Vehicles in Driving Simulators

In [112], a framework for generating realistic traffic on rural roads and highways is presented. The framework itself consists of a microscopic simulation for all objects around the own vehicle where a precise simulation is necessary, and a so-called mesoscopic simulation for vehicles in a greater distance to the own vehicle. Thus, a realistic behavior of vehicles around the own one can be achieved including following another car, changing lanes, or overtaking slowly moving vehicles.

The main focus of the framework is on realistic generation of traffic flows to be integrated in existing driving simulators. Compared to the approach described in this thesis, that framework itself is not applicable for the virtual development of sensor-based algorithms for sensor- and actuator-based autonomous systems.

9.2.3 The Iowa Driving Simulator

A similar system like the hexapod driving simulator at DLR is the Iowa Driving Simulator (*IDS*). The IDS is a scenario-based driving simulator for ground-based vehicles developed at the Center for Computer Aided Design at University of Iowa [37, 148]. Its focus is the simulation of urban and suburban environments as well as highway scenarios. Furthermore, for supporting military research, battlefield simulation is provided. The main focus of the IDS are different, complex, and potentially dangerous traffic situations and their impact on drivers. Thus, the IDS can be classified as DiL- and HiL-simulation as well.

The IDS consists of a three layer architecture. The first layer provides visual, auditive, and haptic feedback for the current traffic situation for the driver using a hydraulic motion platform. Using this platform, different passenger cabins can be mounted. The surroundings are visualized using a four channel projection system covering a 190° field of view in front of the vehicle, and a 60° field of view in the rear of the vehicle. Realistic sounds as well as haptic feedback in the steering wheel and the braking system are generated to suggest a realistic appearance to the driver.

The second layer computes realistic driving behavior of the own vehicle using non-linear differential equations for a composite rigid body model. Thus, a realistic motion of the own car can be achieved. The third layer updates the environment based on the own vehicle's motions. Besides stationary elements like roads including curbs and traffic signs, the user can specify up to 40 dynamic objects like other vehicles or bicyclists which follow the specified traffic rules autonomously.

Technically, IDS uses several independent databases providing a specific subset of the entire simulation, which are connected using a real-time capable communication. All databases reflect the layer they provide data for: One database contains only visual objects for representation, another database provides precise information about the road network. The last database provides information about the scenario itself.

To define behaviors for dynamic elements, IDS uses Hierarchical Concurrent State Machines (*HCSM*) for providing modular behavioral elements to be combined for realizing a complex behavior like passing an object. HCSMs enable parallel execution of independent processes like observing distances and steering the vehicle for a passing maneuver. Furthermore, for creating realistic situations, several virtual elements like directors, beacons, and triggers, which can be associated to surroundings' elements to invoke special behaviors or to control other objects like a traffic light are available [36].

Contrary to the approach outlined in this thesis, the main focus of IDS is to support the research of the driver itself by ensuring repeatable traffic situations. Therefore, a sophisti-

cated visualization environment combined with actuators realizing a haptic perception for the test persons was developed. For supporting the development of autonomous ground vehicles, the IDS is rather inapplicable due to missing interfaces for providing traffic-dependent sensor raw data. Furthermore, a combination with an unattended continuous integration build system seems to be inappropriate. Moreover, the use as a dedicated interactive simulation tool which supports the developer's daily work is impossible on the other hand.

9.2.4 IPG Automotive GmbH: CarMaker

The software suite *IPG CarMaker* from IPG Automotive GmbH is a simulation environment based on MATLAB/Simulink for supporting the development of control systems for automotive applications ranging from control loops using simple vehicle models up to the limit ranges of driving dynamics. Due to the possibility to use real automotive components integrated in the simulation during development besides the aforementioned software controllers, this suite can be classified as HiL- and SiL-system [86].

The simulation suite IPG CarMaker consists of two components, namely "Virtual Vehicle Environment" and the simulation control application. The former component consists of the parts IPGCar describing the virtual car, IPGRoad describing a three-dimensional model of the road, IPGDriver which realizes different driver profiles, and IPGTraffic for simulating surroundings' dynamic elements. The latter component can be used to setup and control the simulation process itself by setting parameters for all models. Furthermore, the execution of simulation runs can be automated and exported as video files for further analysis.

Comparable to the IDS, IPGCar uses a three-dimensional, non-linear, composite rigid body system to compute the data for driving dynamics of a vehicle allowing the mapping of a real vehicle into the simulation. IPGDriver can be used to model the driver itself producing input values for control algorithms. Hereby, different situation-dependent driving profiles realizing defensive or aggressive driving can be realized.

The component IPGTraffic provides different elements for designing traffic situations including stationary elements like traffic signs or parked vehicles, or dynamic elements like cars or pedestrians. Using a graphical tool, these elements can be composed for a scenario setting desired parameters like velocities, timings, or event-based maneuvers. Comparable to the aforementioned trigger concept for IDS, IPGTraffic realizes event-based maneuvers for dynamic elements to ensure reproducibility.

IPG CarMaker seems to support the development of embedded control algorithms and

also driver assistance systems well by providing a tight coupling to MATLAB/Simulink, however, contrary to the approach presented in this thesis, the combined development of high-level algorithms for deriving driving decisions using complex and event-based data structures, and low-level algorithms implementing feature detection on sensor's raw data is less possible. Furthermore, for generating sensor's raw data, every surroundings' element must be modeled in MATLAB/Simulink or proprietary available in the IPG CarMaker environment. Furthermore, properties like reflectivity or its bounding shape for any detectable object in the scene must specified explicitly. Thus, freely positionable stationary or dynamic elements created by popular 3D modeling tools to setup complex situations seems to be less supported only [85].

9.2.5 Microsoft Corporation: Robotics Developer Studio

The *Robotics Developer Studio* provided by Microsoft is a development environment to support the development of experimental robotics platforms. The product features a Visual Programming Language (*VPL*), Visual Simulation Environment (*VSE*) based on NVidia PhysX to provide realistic motion, and realizes the developed components in the sense of software services. Furthermore, a scenario editor is provided to ease the creation of robotics environments [87].

Despite the contribution to the 2007 DARPA Urban Challenge from the Princeton University which was realized using the Microsoft Robotics Developer Studio, their simulation component was self-implemented without using the VSE [92]. Thus, the Microsoft Robotics Developer Studio seems to be inapplicable for virtualizing the development process for developing automotive software systems for sensor- and actuator-based autonomous systems in general and it seems to be rather applicable for selected areas.

9.2.6 PELOPS

The software suite "Programm zur Entwicklung längsdynamischer, mikroskopischer Prozesse in systemrelevanter Umgebung", program for developing longitudinally dynamic, microscopic processes in system-relevant environment (*PELOPS*) provided by Forschungsgesellschaft Kraftfahrwesen mbH Aachen is a simulation environment for traffic flows comparable to the framework mentioned in Section 9.2.2. Beyond, the program can be directly fed by actuating variables computed by control algorithms developed in MATLAB/Simulink or using real hardware components. Thus, the system can be classified as SiL- and HiL-simulation [35].

The vehicle's surroundings consist of different roads modeled using mathematical models, traffic signs, and other vehicles. Furthermore, radar-like raw data can be generated using the current vehicle's surroundings. In contrast to the approach presented in this thesis, three-dimensional models for surroundings' stationary or dynamic elements cannot be used. Furthermore, generating sensor's raw data for laser-based range detectors is not possible.

9.2.7 Player/Stage/Gazebo

The *Player/Stage/Gazebo* project is an open source project aiming to support the development of various different robotics platforms comparable to the Microsoft Robotics Developer studio. The component player is the network interface to robot devices, while Stage is a two-dimensional simulation component supporting populations of several independent robots. Gazebo extends the two-dimensional simulation provided by stage into the third dimension for outdoor applications. The main goal of this project is to simplify the development of control algorithms for robots perceiving their surroundings using different kinds of sensors. Currently, the component stage provides information about the surroundings simulating sonar sensors and laser-based range detectors [67].

This framework is rather suitable because its focus is on experimental robotics platforms offering actuators which are very different from those provided by a vehicle because many experimental robots can turn around their z-axis while not moving at all. Contrary to the approach presented in this thesis, modeling of the vehicle's surroundings is unsupported.

9.2.8 TESIS Gesellschaft für Technische Simulation und Software: DYNAware

Comparable to the aforementioned suite provided by IPG, TESIS provides the simulation suite called *DYNAware* [152]. Their components *DYNA4* and *veDYNA* are meant to support both the HiL- and SiL-development processes for embedded control algorithms and to provide a driving dynamics simulation. Like Player/Stage/Gazebo, veDYNA uses sophisticated models realized in MATLAB/Simulink to compute continuously the model's state. The vehicle model itself is realized using a composite rigid body model.

For providing environmental data, either a standard single-lane road or a double-lane road can be used. The lane itself can consist of a maximum of 1,000 elements. Furthermore, 16 dynamic and 64 stationary elements can be added to a traffic situation. To detect sur-

roundings' elements, up to eight sensors can be defined providing distances and relative velocities to stationary and dynamic elements.

To evaluate simulation results, DYNAanimation can be used to render a video file. For using realistic models in the animation, objects modeled with the Virtual Reality Modeling Language (*VRML*) can be used. However, these models are applied in the post-processing stage for visualization purposes only and are not used in the on-line simulation.

Contrary to the approach presented here, DYNAware provides only rudimentary support for modeling the vehicle's surroundings especially due the limited number of stationary and dynamic elements. Furthermore, only a limited sensor model providing distances and relative velocities can be applied without generating data provided by a camera sensor. Thus, this suite is rather inapplicable for the development of combined low-level and high-level algorithms in sensor- and actuator-based autonomous systems.

9.2.9 IAV GmbH: Test Environment for Synthetic Environment Data

A test environment for generating synthetic environmental data is provided by IAV GmbH [135]. This system aims at generating synthetic data for any kind of active sensors like radar- or laser-based range detectors by specifying the surroundings and the sensor's field of view in a two-dimensional manner. Furthermore, the sensor specific noise can be provided to lower the sensor's raw data's quality.

In contrast to the approach described here, only open-loop test runs can be provided like traditional measuring test drives using the real vehicle including all mounted sensors avoiding potentially dangerous traffic situations using this approach. However, on-line data generation in closed-loop test runs necessary for evaluating an algorithm's behavior depending on its interactions with the surroundings is impossible.

9.2.10 TNO PreScan

The software TNO PRE-crash SCenario ANalyzer (*PreScan*) provided by TNO Automotive is meant to support the development of sensor-based driver assistance systems. Therefore, this suite is also based on MATLAB/Simulink and can be integrated in HiL environments.

Using a graphical scenario editor, stationary elements like roads, trees, and buildings, and dynamic elements like cars, trucks, or pedestrians for the vehicle's surroundings can be freely positioned in a scene. Additionally, behaviors can be associated with dynamic

objects to define routes to be driven. Moreover, sensors providing information about the perceived surroundings in either sensor's raw data for radar-based range detectors or cameras, distances for laser-based range detectors using the pre-defined positions and orientations of the scene's elements, or in abstracted high-level data can be associated with the own vehicle.

Besides the virtual environment supporting the development of pre-collision systems, a mobile robotics platform was developed to realize Vehicle-Hardware-in-the-Loop (*VeHiL*) simulations for evaluating simulation's results in the reality. Therefore, the real vehicle is fixed in a test stand in which the vehicle can safely accelerate and brake. Thus, the vehicle defines the logical origin for the simulation. In front of the vehicle, several mobile robotics platforms simulating surroundings' vehicles are moved regarding to the measured vehicle's motions to evaluate the simulation's results of an algorithms as well as to test real sensor hardware [68].

The main focus of PreScan and VeHiL is to support the development of pre-collision systems. Due to the integration in the MATLAB/Simulink environment as well as the missing support of generating sensor's raw data for laser-based range sensors including not explicitly modeled elements of the surroundings, this suite is only limitedly applicable for the development of sensor- and actuator-based autonomous systems.

9.2.11 VIRES Simulationstechnologie GmbH: Virtual Test Drive

The software *Virtual Test Drive* provided by VIRES Simulationstechnologie GmbH [166] aims to support SiL-, HiL-, ViL-, and DiL-simulations by providing application-dependent input data from a test drive within a virtual scenario. These scenarios base on OpenDRIVE [44] for the description of road networks and environments [168]. The system can be combined with a mock-up cockpit from a vehicle to integrate a human driver. Furthermore, the system provides a script language which allows repeatable situations within a given scenario.

The software was successfully applied for pre-adjusting an algorithm for lane detection [169]. Therefore, a scenario was defined for producing vision data consisting of a sequence of frames which were used as input data for the lane detecting algorithm. Furthermore, the model for the surroundings provided by the specified scenario was used to get the perfect data from the current situation which was stored for the off-line post-processing stage. Afterwards, an evaluation for the detected features from the lane detecting algorithm was carried out off-line which used the stored perfect data for comparison.

As shown by the aforementioned setup, Virtual Test Drive is applicable for the development of combined low-level and high-level algorithms in general. Sensor models for generating input data for sensors-based algorithms base on the ideal model of the surroundings as specified in a scenario; their input data is generated by the specified a range and viewing angle.

The approach presented in this thesis also includes an algorithm for generating synthetic raw data for a laser scanner based on an arbitrary complex 3D scene. Furthermore, an on-line evaluation of an SUD is possible to support the automation of acceptance tests.

9.2.12 Evaluation of Approaches for Software Development and System Testing

In Figure 9.2, a qualitative comparison of the approaches mentioned before is shown. In this table the last three columns reflect one of the aforementioned requirements: *Supporting a virtual development process*, *Supporting the integrated development of low-level and high-level algorithms*, and *Integration of the development and testing environment*. Every approach was qualitatively evaluated using the scale *high*, *medium*, and *low* which is denoted by the corresponding amount of black squares.

A classification as *high* describes that the regarded approach fulfills to a large extent the criterion; *medium* reflects that only a portion of a considered requirement is fulfilled, and *low* describes that the approach does not fulfill or only limitedly fulfills a criteria. Further information about the approaches is available within the respective section.

Approach	**Supporting a virtual development process**	**Supporting the integrated development of low-level and high-level algorithms**	**Integration of the development and testing environment**
Driving Simulator at DLR	■	■■	■
Framework for Simulation of Surrounding Vehicles in Driving Simulators	■	■	■
IDS	■■	■■	■
CarMaker	■■■	■■	■■
Robotics Developer Studio	■■	■■	■
PELOPS	■■■	■■	■■
DYNAware	■■	■	■
Test Environment for Synthetic Environment Data	■■	■	■
PreScan	■■	■■	■■
Virtual Test Drive	■■■	■■	■■

Figure 9.2: Qualitative evaluation of the approaches for software development and system testing: *Driving simulator at DLR*, *Framework for Simulation of Surrounding Vehicles in Driving Simulators*, *IDS*, *CarMaker*, *Robotics Developer Studio*, *PELOPS*, *DYNAware*, *Test Environment for Synthetic Environment Data*, *PreScan*, and *Virtual Test Drive*. The approach *Player/Stage/Gazebo* is not regarded due to its specific application scope as mentioned in Section 9.2.7.

10 Conclusion And Future Work

For evaluating the quality of systems which rely on sensors and actuators to process incoming data for interacting with their context, an appropriate tooling and methods are necessary. However, due to the dependency on the actual sensors' setup already at early stages during the development of the system, which can only be reduced by interfaces or architectural design decisions in a limited manner, an approach for the software engineering is required which supports not only the software development but which also assists activities for evaluating the quality.

In this thesis, a methodology for the software engineering is outlined which supports the creation, evaluation, and automation of acceptance tests for the entire data processing chain of sensor- and actuator-based autonomous systems. The methodology relies on a formal specification of the system's context which is deduced from the customer's requirements. For carrying out an evaluation of the system's behavior in its intended context, the customer's acceptance criteria are used to derive various metrics which are continuously applied during the run-time of the system for evaluation purposes. To enable an automation of the these evaluations, a virtualization of the system and its context is necessary to break the dependency on a real hardware environment for the entire data processing chain. The overall methodology and its application for the V-model is described in Chapter 3.

As mentioned before, the methodology relies on a formal and consistent specification of the system's context. Therefore, on the example of autonomous ground vehicles the surroundings are analyzed to identify stationary and dynamic elements and their relations and behavior. To rely on a consistent representation, mathematical relations are identified which are used to derive a DSL for the stationary and dynamic surroundings. Using this language, consistent and repeatable situations for the SUD can be specified in so-called scenarios. The mathematical relations and the domain analysis of the surroundings of autonomous ground vehicles which are used to derive the DSL for the specification of the system's context are outlined in Chapter 4.

To use these scenarios as artifacts in the software development process as well as to provide the appropriate tooling which supports the aforementioned methodology, a software framework was designed and implemented with the main focus on distributed and commu-

nicating real-time applications. The software framework is designed in an object-oriented manner to be highly portable and is realized using pure ANSI-C++; it was successfully tested on Microsoft Windows XP, Windows Vista, Windows 7, Ubuntu 8.10, openSUSE 11.2, Debian 5.0, NetBSD 5.0.1, and FreeBSD 7.2 and may be used with nearly any POSIX-compatible operating system. Moreover, the framework $\mathcal{H}$esperia contains several tools to support the developer's regular tasks during the development of a sensor- and actuator-based system. For example, a non-reactive data-capturing tool, a complementary playback component, a non-reactive visualization environment, and an application for tracking the life-cycle of all running applications which also deploys consistent, application-dependent, and centrally maintained configuration data. The main design decisions and concepts of the framework $\mathcal{H}$esperia are outlined in Chapter 5.

As outlined in the methodology, the formally specified system's context combined with the evaluation metrics which are derived from the customer's acceptance criteria are used to realize evaluation runs for the SUD. Therefore, a deterministic scheduling environment was designed to evaluate a sensor- and actuator-based autonomous system by decoupling the currently running system from the real system's time; moreover, the scheduling environment also supervises and controls the entire communication. Additionally, for decoupling the software engineering's dependency on the real hardware environment, different algorithms were designed to provide the necessary input data at all layers of the data processing chain during the system simulations. The scheduling environment for system simulations as well as algorithms for virtualizing hardware sensors like monocular color cameras, stereo vision systems, or single layer laser scanners are described in Chapter 6.

For interactively and non-reactively supervising a set of running applications, a monitoring environment was designed as part of the framework $\mathcal{H}$esperia. This application can be used to visualize, inspect, suspend, and replay even step-wisely the entire communication. The application can be easily extended by plug-ins and bases on the concept of a so-called device-independent visualization for 2D- and 3D-representations which allows the visualization of user-contributed data structures without modifying the monitoring environment at all. For evaluating the SUD as outlined in the aforementioned methodology, system simulations which can be unattendedly executed and evaluated by continuous integration systems can be realized with the framework $\mathcal{H}$esperia. Therefore, a concept similar to unit testing was developed; but contrary to unit tests which are mainly used for testing discrete algorithms, the formally specified system's context combined with the aforementioned system simulations is used to provide continuous input data to evaluate continuously the running SUD. Both concepts, the interactive monitoring as well as the unattended evaluations are described in Chapter 7.

The concepts developed in this thesis were finally applied to an autonomous vehicle at University of California, Berkeley. The goal was to develop the required software system to navigate the vehicle safely on a given digital map purely virtually before deploying the resulting artifacts on the vehicle itself. Therefore, a digital map based on consecutive, highly precise GPS-points was created reflecting a simple course on the test site Richmond Field Station. Using an enhanced draw-bar control algorithm both for steering and for accelerating and decelerating the vehicle, the software system was developed interactively first. Therefore, the customer's requirements for this algorithm led to the modeling of the system's context of the SUD which included the stationary context of the Richmond Field Station. This context was used to calculate the required position data from a virtualized IMU system for which a model was derived from the real IMU system.

Following, the software environment was deployed on the vehicle to perform real vehicle tests and to validate the results from the interactive simulations. Afterwards, these previously carried out interactive system evaluations were implemented using the unattended system tests as mentioned before to create an executable specification of a vehicle test using the system's context and the required accuracy. These system tests were automated to be executed and validated automatically whenever any changes to the source code were made. Thus, any errors which might negatively influence the software's quality can be identified easily by evaluating the automatically generated reports for any unattended test runs to locate the modifications which yield to the unexpected behavior of the system. These reports also include the entire communication which is automatically captured during the unattended system evaluations. The case study is presented in Chapter 8.

Thus, not only unit tests for partly ensuring a software's quality can be used for sensor- and actuator-based CPS. Instead, entire sub-systems or systems may be tested virtually to validate the system's quality on the topmost level of the V-model to cover the complete data processing chain for getting a report about the system's quality right before delivery or if the source code was finally optimized for example. Moreover, even interferences with other user-contributed applications introduced by modifications to one application can be identified easily with unattended system simulations; on the example of autonomous ground vehicle, an optimization to a control algorithm which yields in a more sharp steering in curves might influence negatively a possible lane detection algorithm. This interference could be identified automatically if appropriate tests for unattended system simulations were defined. However, the methodology outlined in this thesis is intended to complete and not to substitute real system tests because the reported quality of a system depends directly on the a priori met assumptions about sensors, the system's surroundings, and the like.

To continue the outlined work in this thesis, the separated usage of the two grammars–the MontiCore for Java and the Spirit for C++–could be integrated to derive one from the other to remove the redundancy; another possibility would be to provide a C++ variant of MontiCore to support the DSL-driven development for embedded systems in a native manner. Furthermore, due to the template-based realization in header files of the Spirit grammar, the compile-time takes a long time. Here, a file-based encapsulation of the required Spirit input data would reduce this required compilation time.

Moreover, as already indicated for the system simulations, noise and latency models for the data transmission to artificially reorder, delay, or drop any sent data can be added to the system simulation. Thus, bandwidth limitations depending on the communication or system load can be simulated for a running system simulation. But artificial noise or quality reducing algorithms cannot only be applied at this lower level. Noise models for all sensors which are provided by the framework $\mathcal{H}$esperia may be derived and specified to reduce the quality of the simulated sensors' raw data for example.

Additionally, the run-time control for the system simulation could be extended to allow variable time steps together with the current fixed time step implementation. Furthermore, a graphical user interface could be integrated into the monitoring environment to interactively supervise and interrupt a currently running unattended system simulation. Moreover, the current implementation of the system simulation could be extended to support distributed simulations which are running on several independent computing nodes; thus, even simulations which contain complex elements or which are executed very often by a continuous integration system due to frequent modifications to the software repository could be realized to scale better with an increasing demand of unattended system simulations. Therefore, the virtualized system clock must be distributed to all remotely running system simulations and all communication must also be routed to all these instances.

For supporting the developers' work, an analysis of several test runs of the SUD for the same situation could be realized which combines several recorded data files. Therefore, an appropriate visualization using a transparent overlay technique for example which visualizes all test runs at once would assist the developers to inspect the SUD's algorithm's improvements over time. This technique could also be used to visualize differences between two versions from the software's version history.

Furthermore, the DSL outlined in this thesis for describing the stationary surroundings of an autonomous ground vehicle could be fused with the existing language provided by OpenDRIVE. Alternatively, an Extensible Style-sheet Language (*XSL*) transformation could be applied to instances of OpenDRIVE to transform them for using the data within $\mathcal{H}$esperia.

Besides these technical aspects, both the methodology and the framework could be integrated with AUTOSAR to provide an integrated environment for testing any modeled functionality purely virtually before deploying it onto ECUs. Therefore, any function which is provided as a software component for AUTOSAR resulting as an artifact of an existing tool chain can be wrapped to be used with $\mathcal{H}$esperia. Therefore, the function is embedded into a wrapping instance of `ConferenceClientModule` which provides the required input- and output-interfaces to the system's context. Thus, the function can also be evaluated with the unattended system simulations as outlined in the methodology.

To complete the formal and consistent specification of the SUD's surroundings, a DSL for describing the required test cases for different layers of the V-model or parts of the system might be created. Moreover, this test case-DSL could be used to derive appropriate test cases automatically for interactive or unattended system tests.

Another aspect is the automatic training of a priori unknown input data for intelligent algorithms for example. Instead of defining lots of real test drives to collect the required input data, a set of virtual test situations may be defined with only slight differences in the initial parameters to provide the required input data. This would significantly reduce the necessary time for gathering this training data compared to manual setup or even more compared to real test drives.

Bibliography

[1] IEEE Standards Interpretations for IEEE Standard Portable Operating System Interface for Computer Environments (IEEE Std 1003.1-1988). *IEEE Std 1003.1-1988/INT, 1992 Edition*, July 1992.

[2] M. Aso and T. Suzuki. Automated Steering Control for the Intelligent Multimode Transit System. In *Proceedings of the IEEE Intelligent Vehicles Symposium*, pages 590–595, 2000.

[3] AUTOSAR GbR. Technical Overview. Technical report, AUTOSAR GbR, 2008.

[4] Aviation Week and Space Technology. Researchers Channel AI Activities toward Real-World Applications. *Aviation Week and Space Technology*, 17:40–52, February 1986.

[5] A. Bacha, C. Bauman, R. Faruque, M. Fleming, C. Terwelp, C. Reinholtz, D. Hong, A. Wicks, T. Alberi, D. Anderson, S. Cacciola, P. Currier, A. Dalton, J. Farmer, J. Hurdus, S. Kimmel, P. King, A. Taylor, D. V. Covern, and M. Webster. Odin: Team VictorTango's Entry in the DARPA Urban Challenge. *Journal of Field Robotics*, 25(9):467–492, September 2008.

[6] E. Bakker, H. Pacejka, and L. Lidner. A New Tire Model with an Application in Vehicle Dynamics Studies. *SAE Paper*, 890087, 1989.

[7] A. Bartels, C. Berger, H. Krahn, and B. Rumpe. Qualitätsgesicherte Fahrentscheidungsunterstützung für automatisches Fahren auf Schnellstrassen und Autobahnen. In Gesamtzentrum für Verkehr Braunschweig e.V., editor, *AAET 2009 – Automatisierungssysteme, Assistenzsysteme und eingebettete Systeme für Transportmittel*, volume 10, pages 341–353, February 2009.

[8] R. H. Bartels, J. C. Beatty, and B. A. Barsky. *An Introduction to Splines for Use in Computer Graphics & Geometric Modeling*. Morgan Kaufmann, 1995.

[9] C. Basarke, C. Berger, K. Berger, K. Cornelsen, M. Doering, J. Effertz, T. Form, T. Gülke, F. Graefe, P. Hecker, K. Homeier, F. Klose, C. Lipski, M. Magnor, J. Morgenroth, T. Nothdurft, S. Ohl, F. W. Rauskolb, B. Rumpe, W. Schumacher, J.-M.

Wille, and L. Wolf. Team CarOLO – Technical Paper. Informatik-Bericht 2008-07, Technische Universität Braunschweig, October 2008.

[10] C. Basarke, C. Berger, K. Homeier, and B. Rumpe. Design and Quality Assurance of Intelligent Vehicle Functions in the "Virtual Vehicle". *Proceedings of 11. Automobiltechnische Konferenz – Virtual Vehicle Creation, Stuttgart*, 9, June 2007.

[11] C. Basarke, C. Berger, and B. Rumpe. Software & Systems Engineering Process and Tools for the Development of Autonomous Driving Intelligence. *Journal of Aerospace Computing, Information, and Communication*, 4(12):1158–1174, December 2007.

[12] C. Basarke, C. Berger, and B. Rumpe. Using Intelligent Simulations for Quality Assurance in the 2007 DARPA Urban Challenge. IEEE International Conference on Robotics and Automation - Workshop: The 2007 DARPA Urban Challenge: From Algorithms to Autonomous Vehicles, May 2008.

[13] bdec. bdec. `http://www.hl.id.au/projects/bdec/`, February 2010.

[14] K. Beevers and J. Peng. A Graph Search Within the BGL Framework. Technical report, Rensselaer Polytechnic Institute, 2003.

[15] R. Behringer. *Visuelle Erkennung und Interpretation des Fahrspurverlaufes durch Rechnersehen für ein autonomes Straßenfahrzeug*. PhD thesis, Universität der Bundeswehr München, 1996.

[16] Benz & Co. Patenschrift 37,435 – Fahrzeug mit Gasmotorenbetrieb. Kaiserliches Patentamt des Deutschen Reiches, 1886.

[17] C. Berger and B. Rumpe. Hesperia: Framework zur Szenario-gestützten Modellierung und Entwicklung Sensor-basierter Systeme. In S. Fischer, E. Maehle, and R. Reischuk, editors, *Proceedings INFORMATIK 2009*, volume 154, pages 328,2668–2680. GI-Edition Lecture Notes in Informatics (LNI), September 2009.

[18] C. Berger and B. Rumpe. Nutzung von projektiven Texturen auf einer GPU zur Distanzmessung für automotive Sensorsimulationen. In Gesamtzentrum für Verkehr Braunschweig e.V., editor, *AAET 2009 – Automatisierungssysteme, Assistenzsysteme und eingebettete Systeme für Transportmittel*, volume 10, pages 319–329, February 2009.

[19] C. Berger and B. Rumpe. Supporting Agile Change Management by Scenario-Based Regression Testing. *IEEE Transactions on Intelligent Transportation Systems*, 11(2):504–509, June 2010.

[20] K. Berger, C. Lipski, C. Linz, T. Stich, and M. Magnor. The area processing unit

of Caroline - Finding the way through DARPA's Urban Challenge. *2nd Workshop Robot Vision*, pages 260–274, Feb. 2008.

[21] J. Biermeyer, H. Gonzales, N. Naikal, T. Templeton, S. Sastry, C. Berger, and B. Rumpe. Rapid Integration and Automatic Calibration for new Sensors using the Berkeley Aachen Robotics Toolkit BART. In Gesamtzentrum für Verkehr Braunschweig e.V., editor, *AAET 2010 – Automatisierungssysteme, Assistenzsysteme und eingebettete Systeme für Transportmittel*, volume 11, pages 71–88, February 2010.

[22] J. Blanchette and M. Summerfield. *C++ GUI Programming with Qt 4*. Prentice Hall Press, 2nd edition, 2008.

[23] Boost. Boost::Spirit. `http://spirit.sourceforge.net`, April 2009.

[24] H. Bossel. *Modeling and Simulation*. A K Peters, 1994.

[25] G. Bradski and A. Kaehler. *Learning OpenCV*. O'Reilly, 2008.

[26] H.-H. Braess and G. Reichart. PROMETHEUS: Vision des "intelligenten Automobils" auf der "intelligenten Strasse"- Versuch einer kritischen Würdigung. *Automobiltechnische Zeitschrift*, 4:200–205, 1997.

[27] H.-H. Braess and G. Reichart. PROMETHEUS: Vision des "intelligenten Automobils" auf der "intelligenten Strasse"- Versuch einer kritischen Würdigung. *Automobiltechnische Zeitschrift*, 6:330–343, 1997.

[28] A. Broggi, M. Bertozzi, and A. Fascioli. ARGO and the MilleMiglia in Automatico Tour. *IEEE on Intelligent Systems and their Applications*, 14(1):55–64, January/February 1999.

[29] A. Broggi, M. Bertozzi, A. Fascioli, C. G. L. Bianco, and A. Piazzi. The ARGO Autonomous Vehicle's Vision and Control Systems. *International Journal of Intelligent Control and Systems*, 3(4):409–441, 1999.

[30] I. N. Bronstein, K. A. Semendjajew, G. Musiol, and H. Mühlig. *Taschenbuch der Mathematik*. Verlag Harri Deutsch, 2008.

[31] M. Broy. Automotive Software and Systems Engineering. In *Proceedings of IEEE International Conference on Formal Methods and Models for Co-Design*, pages 143–149, 2005.

[32] H.-J. Bungartz, M. Griebel, and C. Zenger. *Introduction to Computer Graphics*. Charles River Media, Inc., 2004.

[33] N. Bunkley. G.M. to Show a Vehicle That Drives by Itself. The New York Times, January 2006.

[34] CARMEN. CARMEN Robot Navigation Toolkit. `http://carmen.sourceforge.net`, July 2009.

[35] F. Christen. PELOPS White Paper 1.0. Technical report, Forschungsgesellschaft Kraftfahrwesen mbH Aachen, 2007.

[36] J. Cremer, J. Kearney, and Y. Papelis. HCSM: A Framework for Behavior and Scenario Control in Virtual Environments. *ACM Transactions on Modeling and Computer Simulation*, 5:242–267, 1995.

[37] J. Cremer, J. Kearney, Y. Papelis, and R. Romano. The Software Architecture for Scenario Control in the Iowa Driving Simulator. In *Proceedings of the 4th Computer Generated Forces and Behavioral Representation Conference*, May 1994.

[38] CruiseControl. CruiseControl. `http://cruisecontrol.sourceforge.net`, June 2009.

[39] DARPA. Urban Challenge Technical Evaluation Criteria. Technical report, DARPA, 3701 North Fairfax Drive, Arlington, VA 22203-1714, March 2006.

[40] Deutsches Institut für Normung e.V. DIN 70000: Straßenfahrzeuge; Fahrzeugdynamik und Fahrverhalten; Begriffe, January 1991.

[41] E. Dickmanns, R. Behringer, D. Dickmanns, T. Hildebrandt, M. Maurer, F. Thomanek, and J. Schiehlen. The seeing passenger car 'VaMoRs-P'. *Proceedings of the Intelligent Vehicles Symposium*, pages 68–73, October 1994.

[42] M. W. M. G. Dissanayake, P. Newman, S. Clark, H. F. Durrant-Whyte, and M. Csorba. A Solution to the Simultaneous Localization and Map Building (SLAM) Problem. *IEEE Transactions on Robotics and Automation*, 17(3):229–241, June 2001.

[43] B. P. Douglass. *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Addison-Wesley, 2002.

[44] M. Dupuis and H. Grezlikowski. OpenDRIVE–An Open Standard for the Description of Roads in Driving Simulations. *Proceedings of the Driving Simulation Conference*, pages 25–36, 2006.

[45] J. Effertz. *Robot Vision*, volume 4931/2008, chapter Sensor Architecture and Data Fusion for Robotic Perception in Urban Environments at the 2007 DARPA Urban Challenge, pages 275–290. Springer Berlin / Heidelberg, 2008.

[46] Elektrobit Automotive GmbH. EB Assist ADTF 2.1.1 - Benutzerhandbuch. Technical report, Elektrobit Automotive GmbH, April 2009.

[47] Elektrobit Automotive GmbH. EB Assist ADTF 2.1.1 - Entwicklerhandbuch. Technical report, Elektrobit Automotive GmbH, April 2009.

[48] D. Erickson, B. Beckman, and T. Peng. Promoting Interoperability: The libdrdc Data Standards Library. In *Proceedings on Intelligent Robots and Systems*, pages 1842–1847, September 2008.

[49] G. I. Evenden. Cartographic Projection Procedures for the UNIX Environment – A User's Manual. Technical Report Open-File-Report 90-284, United States Department of the Interior, January 2003.

[50] C. Everitt. Projective Texture Mapping. Technical report, NVidia Corporation, `http://www.nvidia.com/developer`, 2001.

[51] H. Fennel, S. Bunzel, H. Heinecke, J. Bielefeld, S. Fürst, K.-P. Schnelle, W. Grote, N. Maldener, T. Weber, F. Wohlgemuth, J. Ruh, L. Lundh, T. Sandén, P. Heidkämper, R. Rimkus, J. Leflour, A. Gilbert, U. Virnich, S. Voget, K. Nishikawa, , K. Kajio, K. Lange, T. Scharnhorst, and B. Kunkel. Achievements and exploitation of the AUTOSAR development partnership. Technical report, Society of Automotive Engineers, 2006.

[52] R. Fenton. IVHS/AHS: driving into the future. *IEEE Control Systems Magazine*, 14(6):13–20, December 1994.

[53] B. Finkemeyer, T. Kröger, D. Kubus, M. Olschewski, and F. Wahl. MiRPA: Middleware for Robotic and Process Control Applications. 2007.

[54] T. Form, B. Rumpe, C. Berger, K. Cornelsen, M. Doering, J. Effertz, T. Gülke, K. Homeier, A. Movshyn, T. Nothdurft, M. Sachse, and J. M. Wille. Caroline – Ein autonom fahrendes Fahrzeug im Stadtverkehr. In Gesamtzentrum für Verkehr Braunschweig e.V., editor, *AAET 2007 – Automatisierungssysteme, Assistenzsysteme und eingebettete Systeme für Transportmittel*, volume 8, pages 121–140, February 2007.

[55] J. W. Forrester. *Industrial Dynamics*. MIT Press, 1961.

[56] Frank E. Schneider. ELROB - The European Robot Trial. `http://www.elrob.org/imprint.html`, June 2009.

[57] U. Franke, D. M. Gavrila, S. Görzig, F. Lindner, F. Paetzold, and C. Wöhler. Autonomous Driving Goes Downtown. *IEEE Intelligent Systems*, 13(6):40–48, November/December 1998.

[58] U. Franke, S. Görzig, F. Lindner, D. Mehren, and F. Paetzold. Steps towards an intelligent vision system for driver assistance in urban traffic. In *IEEE Conference on Intelligent Transportation System*, pages 601–606, Nov 1997.

[59] U. Franke and S. Heinrich. Fast Obstacle Detection for Urban Traffic Situations. In *IEEE Transactions on Intelligent Transportation Systems*, volume 3, pages 173–181, September 2002.

[60] U. Franke and A. Joos. Real-time Stereo Vision for Urban Traffic Scene Understanding. *Proceedings of the IEEE Intelligent Vehicles Symposium*, pages 273–278, 2000.

[61] H.-G. Frischkorn, H. Negele, and J. Meisenzahl. The Need for Systems Engineering: An Automotive Project Perspective. In *Key Note at the 2nd European Systems Engineering Conference (EuSEC 2000), Munich*, 2000.

[62] H. Fritz. Model-Based Neural Distance Control for Autonomous Road Vehicles. *Proceedings of the IEEE Intelligent Vehicles Symposium*, pages 29–34, Sep 1996.

[63] B. Furht, J. Parker, and D. Grostick. Performance of REAL/IX™-fully preemptive real time UNIX. *ACM SIGOPS Operating Systems Review*, 23(4):45–52, 1989.

[64] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[65] D. M. Gavrila, U. Franke, C. Wöhler, and S. Görzig. Real-Time Vision for Intelligent Vehicles. *IEEE Instrumentation & Measurement Magazine*, 4(2):22–27, June 2001.

[66] General Motors Corporation. GM Statement On Officer And Board Announcements. General Motors Corporation, March 2009.

[67] B. Gerkey, R. T. Vaughan, and A. Howard. The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems. In *Proceedings of the 11th International Conference on Advanced Robotics*, pages 317–323, 2003.

[68] O. Gietelink, J. Ploeg, B. D. Schutter, and M. Verhaegen. Testing Advanced Driver Assistance Systems for Fault Management with the VEHIL Test Facility. In *Proceedings of the 7th International Symposium on Advanced Vehicle Control*, pages 579–584, August 2004.

[69] T. D. Gillespie. *Fundamentals of Vehicle Dynamics*. Society of Automotive Engineers, 1992.

[70] T. G. Goodwin. DARPA Schedules Autonomous Robotic Ground Vehicles Event. DARPA, June 2004.

[71] T. G. Goodwin. A HUGE LEAP FORWARD FOR ROBOTICS R&D. DARPA, October 2005.

[72] Google. Protocol Buffers. `http://code.google.com/intl/de-DE-/apis/protocolbuffers/`, February 2010.

[73] T. Gowers, J. Barrow-Green, and I. Leader, editors. *The Princeton Companion to Mathematics*. Princeton University Press, 2008.

[74] H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, and S. Völkel. MontiCore: a framework for the development of textual domain specific languages. In *ICSE Companion '08: Companion of the 30th international conference on Software engineering*, pages 925–926, New York, NY, USA, 2008. ACM.

[75] D. L. Hall and J. Llinas. An Introduction to Multisensor Data Fusion. *Proceedings of the IEEE*, 85(1):6–23, 1997.

[76] Hamburger Hafen und Logistik AG. Containertransport mit Batterie: Null Abgas für den Hafen. Hamburger Hafen und Logistik AG, June 2009.

[77] H. Heinecke, K.-P. Schnelle, H. Fennel, J. Bortolazzi, L. Lundh, J. Leflour, J.-L. Maté, K. Nishikawa, and T. Scharnhorst. AUTomotive Open System ARchitecture– An Industry-Wide Initiative to Manage the Complexity of Emerging Automotive E/E-Architectures. volume 21, October 2004.

[78] B. Heißing and M. Ersoy, editors. *Fahrwerkhandbuch: Grundlagen, Fahrdynamik, Komponenten, Systeme, Mechatronik, Perspektiven*. Springer, 2007.

[79] K. Henning and E. Preuschoff. Einsatzszenarien für Fahrerassistenzsysteme im Güterverkehr und deren Bewertung. In *VDI-Fortschrittsberichte*, number 531, 2003.

[80] M. Henning and M. Spruiell. Distributed Programming with Ice. Technical report, ZeroC Inc., 2008.

[81] High Tech Automotive Systems. Grand Cooperative Driving Challenge. `http://www.gcdc.nl`, June 2009.

[82] HIS. Specification Requirements Interchange Format (RIF). Technical report, HIS, 2005.

[83] IBM. Rational DOORS. `http://www.telelogic.com/Products/-doors/doors/index.cfm`, June 2009.

[84] Institute of Electrical and Electronics Engineers, Inc. IEEE 1588 – Precision clock synchronization protocol for networked measurement and control systems, September 2004.

[85] IPG. CarMaker. `http://www.ipg.com`, April 2009.

[86] IPG Automotive GmbH. IPG CarMaker. `http://www.ipg.de/carmaker.html`, June 2009.

[87] E. Jewett. Robotics Developer Studio 2008R2. Technical report, Microsoft Corp., 2009.

[88] S. S. Johnson. YACC: Yet Another Compiler-Compiler. Technical report, Bell Laboratories, 1978.

[89] Joint Architecture for Unmanned Systems Working Group. Reference Architecture Specification Version 3.3. `http://www.jauswg.org`, June 2007.

[90] J. Kessenich. The OpenGL®Shading Language. Technical report, 3DLabs Inc., `http://www.opengl.org/documentation/glsl/`, 2006.

[91] O. Kindel and M. Friedrich. *Softwareentwicklung mit AUTOSAR: Grundlagen, Engineering, Management in der Praxis*. dpunkt Verlag, 2009.

[92] A. L. Kornhauser, A. Atreya, B. Cattle, S. Momen, B. Collins, A. Downey, G. Franken, J. Glass, Z. Glass, J. Herbach, A. Saxe, I. Ashwash, C. Baldassano, W. Hu, U. Javed, J. Mayer, D. Benjamin, L. Gorman, and D. Yu. DARPA Urban Challenge–Princeton University–Technical Paper. Technical report, Department of Operations Research and Financial Engineering, Princeton University, 2007.

[93] H. Krahn, B. Rumpe, and S. Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Proceedings of Models 2007*, 2007.

[94] E. A. Lee. Computing Foundations and Practice for Cyber-Physical Systems: A Preliminary Report. Technical Report UCB/EECS-2007-72, University of California, Berkeley, May 2007.

[95] S. J. Lee, D. M. Lee, and J. C. Lee. Development of Communication Framework for Unmanned Ground Vehicle. In *Proceedings on Control, Automation and Systems*, pages 604–607, October 2008.

[96] V. Lepetit and P. Fua. *Monocular Model-Based 3D Tracking of Rigid Objects*. Now Publishers, 2005.

[97] J. R. Levine, T. Mason, and D. Brown. *lex & yacc*. O'Reilly, 1992.

[98] C. Lipski, K. Berger, and M. Magnor. vIsage – A visualization and debugging framework for distributed system applications. In *Proc. of International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, volume 17, pages 1–7, February 2009.

[99] C. Lipski, B. Scholz, K. Berger, C. Linz, T. Stich, and M. Magnor. A Fast and Robust Approach to Lane Marking Detection and Lane Tracking. *Proc. IEEE Southwest Symposium on Image Analysis and Interpretation*, 2008.

[100] A. Makarenko, A. Brooks, and T. Kaupp. Orca: Components for Robotics. In *International Conference on Intelligent Robots and Systems*, pages 163–168, 2006.

[101] Massachusetts Institute of Technology. Lightweight Communications and Marshalling. `http://lcm.googlecode.com`, July 2009.

[102] G. Meszaros. *xUnit Test Patterns*. Addison-Wesley, 2007.

[103] D. L. Mills. Network Time Protocol Version 4 Reference and Implementation Guide. Technical report, University of Delaware, 2006.

[104] M. Mitschke and H. Wallentowitz. *Dynamik der Kraftfahrzeuge*. Springer, 2004.

[105] C. Z. Mooney. *Monte Carlo Simulation*. Sage Publications, 1997.

[106] H. P. Moravec. The Stanford Cart and the CMU Rover. *Proceedings of the IEEE*, 71(7):872–884, July 1983.

[107] M. Mossige, P. Sampath, and R. G. Rao. Evaluation of Linux rt-preempt for embedded industrial devices for Automation and Power Technologies-A Case Study. In *Proceedings of the 9th Real-Time Linux Workshop*, 2007.

[108] R. Mukundan. Quaternions: From Classical Mechanics to Computer Graphics, and Beyond. In *Proceedings of the 7th Asian Technology Conference in Mathematics*, pages 97–105, 2002.

[109] National Imagery and Mapping Agency. Department of Defense World Geodetic System 1984. Technical Report NIMA TR8350.2, National Imagery and Mapping Agency, January 2000.

[110] National Marine Electronics Association. NMEA 0183 Standard for Interfacing Marine Electronic Devices, Version 4.00. Technical report, National Marine Electronics Association, 2002.

[111] U. Noyer, H. Mosebach, S. Karrenberg, H. Philipps, and A. Bartels. Hochpräzise Erfassung und Modellierung von Straßenverläufen für eine digitale Karte zur Unterstützung des autonomen Fahrens im Intelligent-Car-Projekt. In Gesamtzentrum für Verkehr Braunschweig e.V., editor, *AAET 2009 – Automatisierungssysteme, Assistenzsysteme und eingebettete Systeme für Transportmittel*, volume 10, pages 239–253, February 2009.

[112] J. J. Olstam, J. Lundgren, M. Adlers, and P. Matstoms. A Framework for Simulation of Surrounding Vehicles in Driving Simulators. *ACM Transactions on Modeling and Computer Simulations*, 18(3):1–24, 2008.

[113] OpenJAUS.com. OpenJAUS. `http://www.openjaus.com`, June 2009.

[114] OpenSceneGraph. OpenSceneGraph. `http://openscenegraph.sourceforge.net`, July 2009.

[115] B. Page and W. Kreutzer. *The Java Simulation Handbook: Simulating Discrete Event Systems with UML and Java*. Shaker Verlag, 2005.

[116] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, first edition, May 2007.

[117] D. Pomerleau. RALPH: rapidly adapting lateral position handler. In *Proceedings of the Intelligent Vehicles Symposium*, pages 506–511, September 1995.

[118] D. Pomerleau. Visibility estimation from a moving vehicle using the RALPH vision system. In *IEEE Conference on Intelligent Transportation System*, pages 906–911, November 1997.

[119] D. Pomerleau and T. Jochem. Rapidly adapting machine vision for automated vehicle steering. *IEEE Expert*, 11(2):19–27, April 1996.

[120] A. Pretschner, M. Broy, I. H. Krüger, and T. Stauner. Software Engineering for Automotive Systems: A Roadmap. In *Proceedings of 2007 Future of Software Engineering*, pages 55–71, Washington, DC, USA, 2007. IEEE Computer Society.

[121] J. Pritchard. *COM and CORBA Side by Side: Architectures, Strategies, and Implementations*. Addison-Wesley Professional, 1999.

[122] F. W. Rauskolb, K. Berger, C. Lipski, M. Magnor, K. Cornelsen, J. Effertz, T. Form, F. Graefe, S. Ohl, W. Schumacher, J.-M. Wille, P. Hecker, T. Nothdurft, M. Doering, K. Homeier, J. Morgenroth, L. Wolf, C. Basarke, C. Berger, T. Gülke, F. Klose, and B. Rumpe. Caroline: An Autonomously Driving Vehicle for Urban Environments. *Journal of Field Robotics*, 25(9):674–724, September 2008.

[123] E. S. Raymond. *The Art of Unix Programming*. Addison-Wesley, 2003.

[124] K. Reif. *Automobilelektronik: Eine Einführung für Ingenieure*. Friedr. Vieweg & Sohn Verlag, 2007.

[125] D. Riehle, W. Siberski, D. Bäumer, D. Megert, and H. Züllighoven. *Pattern Languages of Program Design 3*, chapter Serializer, pages 293–312. Addison-Wesley, 1998.

[126] P. Riekert and T. E. Schunck. Zur Fahrmechanik des gummibereiften Kraftfahrzeugs. In *Archive of Applied Mechanics*. Springer, 1940.

[127] T. Roosendaal and C. Wartmann. *The official Blender 2.0 guide*. Macmillan Technical Publishing, 2001.

[128] J. K. Rosenblatt. DAMN: a distributed architecture for mobile navigation. *Journal of Experimental & Theoretical Artificial Intelligence*, 9(2):339–360, 1997.

[129] B. Rumpe. *Modellierung mit UML*. Springer Verlag, 2004.

[130] B. Rumpe. *Agile Modellierung mit UML*. Springer Verlag, 2005.

[131] B. Rumpe, C. Berger, and H. Krahn. Softwaretechnische Absicherung intelligenter Systeme im Fahrzeug. In VDI Wissensforum IWB GmbH, editor, *Integrierte Sicherheit und Fahrerassistenzsysteme*, number 1960, pages 473–486. VDI-Gesellschaft Fahrzeug- und Verkehrstechnik, October 2006.

[132] R. Schabenberger. ADTF: Framework for Driver Assistance and Safety Systems. In VDI Wissensforum IWB GmbH, editor, *Integrierte Sicherheit und Fahrerassistenzsysteme*, number 2000, pages 701–710. VDI-Gesellschaft Fahrzeug- und Verkehrstechnik, October 2007.

[133] D. C. Schmidt. Middleware for REAL-TIME and EMBEDDED SYSTEMS. *Communications of the ACM*, 45(6):43–48, 2002.

[134] R. Schmidt, H. Weisser, P. Schulenberg, and H. Goellinger. Autonomous Driving on Vehicle Test Tracks: Overview, Implementation and Results. *Proceedings of the Intelligent Vehicles Symposium*, pages 152–155, 2000.

[135] B. Schonlau and R. Zschoppe. Test und Absicherung von Funktionen mit synthetischen Umfeld- und Fahrzeugeigendaten. In Gesamtzentrum für Verkehr Braunschweig e.V., editor, *AAET 2009 – Automatisierungssysteme, Assistenzsysteme und eingebettete Systeme für Transportmittel*, volume 10, pages 106–121, February 2009.

[136] G. Schulz. *Regelungstechnik: Mehrgr "oßenregelung-Digitale Regelungstechnik-Fuzzy-Regelung*. Oldenbourg Wissenschaftsverlag, 2002.

[137] M. Schulze. CHAUFFEUR – The European Way Towards an Automated Highway System. In *Proceedings of the 4th World Congress on Intelligent Transportation Systems*, 1997.

[138] M. Schulze. PROMOTE–CHAUFFEUR. Final Report. EU Telematics Applications Program (Sector Transport), 1999.

[139] D. Shipley. DARPA Plans Grand Challenge for Robotic Ground Vehicles. DARPA, January 2003.

[140] D. Shipley, J. Celko, and J. Walker. DARPA Announces Urban Challenge Finalists. DARPA, November 2007.

[141] D. Shipley and J. Walker. TARTAN RACING WINS $2 MILLION PRIZE FOR DARPA URBAN CHALLENGE. DARPA, November 2007.

[142] S. E. Shladover, C. A. Desoer, J. K. Hedrick, M. Tomizuka, J. Walrand, W.-B. Zhang, D. H. McMahon, H. Peng, S. Sheikholeslam, and N. McKeown. Automatic Vehicle Control Developments in the PATH Program. *IEEE Transactions on Vehicular Technology*, 40(1):114–130, February 1991.

[143] Sick. Lasermesssysteme LMS200/LMS211/LMS221/LMS291. `http://mysick.com`, April 2009.

[144] J. G. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library*. Addison-Wesley, 2001.

[145] N. Sintara. Entwurf und Realisierung eines plattformübergreifenden, graphischen Verkehrsszenario-Editors. Master's thesis, Technische Universität Braunschweig, December 2008.

[146] I. Sommerville. *Software Engineering*. Addison-Wesley, 2006.

[147] B. Spice and A. Watzman. Carnegie Mellon Tartan Racing Wins $2 Million DARPA Urban Challenge. Carnegie Mellon University, November 2007.

[148] J. W. Stoner, E. J. Haug, K. S. Berbaum, D. F. Evans, J. G. Kuhl, J. C. Lenel, J. F. McAreavy, and F.-F. Tsai. Introduction to the Iowa Driving Simulator and Simulation Research Program. Technical Report Technical Report R-86, Center for Simulation and Design Optimization of Mechanical Systems, University of Iowa, 1990.

[149] G. Süss. Fahrerloser Transport im Krankenhaus. *building & automation*, 1, 2007.

[150] S. Tachi and K. Komoriya. GUIDE DOG ROBOT. Technical report, Mechanical Engineering Laboratory, 1977.

[151] S. Tachi, K. Tanie, K. Komoriya, and M. Abe. Electrocutaneous Communication in a Guide Dog Robot (MELDOG). *IEEE Transactions on Biomedical Engineering*, BME-32(7):461–469, July 1985.

[152] TESIS Gesellschaft für Technische Simulation und Software mbH. TESIS DYNAware. `http://www.tesis.de/en/index.php?page=1004`, July 2009.

[153] The Eclipse Foundation. Eclipse Rich Client Platform. http://www.eclipse.org/rcp, April 2009.

[154] The MathWorks. MATLAB. http://www.mathworks.com, June 2009.

[155] S. Thrun, M. Montemerlo, H. Dahlkamp, D. Stavens, A. Aron, J. Diebel, P. Fong, J. Gale, M. Halpenny, G. Hoffmann, K. Lau, C. Oakley, M. Palatucci, V. Pratt, P. Stang, S. Strohband, C. Dupont, L.-E. Jendrossek, C. Koelen, C. Markey, C. Rummel, J. van Niekerk, E. Jensen, P. Alessandrini, G. Bradski, B. Davies, S. Ettinger, A. Kaehler, A. Nefian, and P. Mahoney. Stanley: The Robot that Won the DARPA Grand Challenge. *Journal of Field Robotics*, 23(9):661–692, September 2006.

[156] Tigris.org. CxxTest. http://cxxtest.tigris.org, April 2009.

[157] TORC Technologies. ByWire XGV. http://www.torctech.com, July 2009.

[158] TORC Technologies. JAUS Toolkit. http://www.jaustoolkit.com, August 2009.

[159] S. Tsugawa, T. Hirose, and T. Yabate. An Automobile with Artificial Intelligence. In *Proceedings of the Sixth International Joint Conference of Artificial Intelligence*, pages 893–895, 1979.

[160] B. Ulmer. VITA–An Autonomous Road Vehicle (ARV) for Collision Avoidance in Traffic. *Proceedings of the Intelligent Vehicles Symposium*, pages 36–41, July 1992.

[161] B. Ulmer. VITA II–Active Collision Avoidance in Real Traffic. In *Proceedings of the Intelligent Vehicles Symposium*, pages 1–6, 1994.

[162] M. Underseth. The complexity crisis in embedded software. *Embedded Computing Design*, pages 31–33, April 2007.

[163] University of Central Florida. JAUS++. http://active-ist.sourceforge.net/, June 2009.

[164] C. Urmson, D. Duggins, T. Jochem, D. Pomerleau, and C. Thorpe. From Automated Highways to Urban Challenges. *IEEE International Conference on Vehicular Electronics and Safety*, pages 6–10, September 2008.

[165] T. Veldhuizen. Expression Templates. *C++ Report*, 7(5):26–31, 1995.

[166] VIRES Simulationstechnologie GmbH. VIRES Simulationstechnologie GmbH. http://www.vires.com, January 2010.

[167] Volere. Volere. http://www.volere.co.uk, January 2010.

[168] K. von Neumann-Cosel, M. Dupuis, and C. Weiss. Virtual Test Drive - Provision of a Consistent Tool-Set for [D,H,S,V]-in-the-Loop. In *Proceedings on Driving Simulation Conference*, Februar 2009.

[169] K. von Neumann-Cosel, M. Nentwig, D. Lehmann, J. Speth, and A. Knoll. Preadjustment of a Vision-Based Lane Tracker. In *Proceedings on Driving Simulation Conference*, Februar 2009.

[170] J. Walker and J. S. Jones. DARPA Announces Third Grand Challenge: Urban Challenge Moves to the City. DARPA, May 2006.

[171] H. Wallentowitz and K. Reif, editors. *Handbuch Kraftfahrzeugelektronik: Grundlagen, Komponenten, Systeme, Anwendungen*. Vieweg + Teubner Verlag, 2006.

[172] A. Weiser, A. Bartels, S. Steinmeyer, K. Schultze, M. Musial, and K. Weiß. Intelligent Car – Teilautomatisches Fahren auf der Autobahn. In Gesamtzentrum für Verkehr Braunschweig e.V., editor, *AAET 2009 – Automatisierungssysteme, Assistenzsysteme und eingebettete Systeme für Transportmittel*, volume 10, pages 11–26, February 2009.

[173] H. Weisser, P. Schulenberg, R. Bergholz, and U. Lages. Autonomous Driving on Vehicle Test Tracks: Overview, Motivation and Concept. In *IEEE International Conference on Intelligent Vehicles*, volume 2, pages 439–443, 1998.

[174] J. M. Wille and T. Form. Low Level Control in a Modular System Architecture for Realizing Precise Driving Maneuvers of the Autonomous Vehicle Caroline. In *Proceedings of the 11th International IEEE Conference on Intelligent Transportation Systems*, pages 705–710, October 2008.

[175] J. M. Wille and T. Form. Realizing Complex Autonomous Driving Maneuvers – The Approach Taken by Team CarOLO at the DARPA Urban Challenge. In *Proceedings of the 2008 IEEE International Conference on Vehicular Electronics and Safety*, pages 232–236, Sept. 2008.

[176] Willow Garage. ROS. `http://www.ros.org`, August 2009.

[177] A. Zapp. *Automatische Straßenfahrzeugführung durch Rechnersehen*. PhD thesis, Universität der Bundeswehr München, 1988.

[178] B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press, 2000.

List of Figures

List of Equations

List of Listings

Appendices

A Grammar for Defining the System's Context

A.1 MontiCore Grammar for the Stationary Surroundings

```
grammar Scenario {
  ident AlphaNum "('A'..'Z'|'a'..'z')('A'..'Z'|'a'..'z'|'_'
  '|'-'|'.'|'/'|'0'..'9')*";
  ident Number   "('+'|'-')?( (('1'..'9')('0'..'9')*) | '0' )
  ('.'('0'..'9')*)?";
  ident FQID     "'('(('1'..'9')('0'..'9')*)'.'(('1'..'9')
  ('0'..'9')*)|'0')'.'(('1'..'9')('0'..'9')*)|'0')
  '.'(('1'..'9')('0'..'9')*)')'";

  ScenarioFile =
    ScenarioHeader // File header.
    Ground // Ground layer.
    Layer+ ";"; // Data layers containing roads and other
    stuff.

  ScenarioHeader =
    "Scenario" ScenarioName:AlphaNum // Scenario's name.
    "Version"  Version:AlphaNum // Version of the scenario
    grammar.
    "Date"     Date:AlphaNum // Creation date of the file.
    "OriginCoordinateSystem"
    CoordinateSystem:CoordinateSystem; // String type of
    relative coordinate system.

  CoordinateSystem =
```

```
  WGS84CoordinateSystem
  "Rotation" Rotation:Number; // The scenario's rotation around the coordinate's origin in RAD (3am is 0).

WGS84CoordinateSystem =
  Type:"WGS84" // WGS84 type.
  "Origin" Origin:Vertex2; // Logical (0, 0) is at (OriginX, OriginY) in the chosen coordinates.

Ground =
  "Ground" GroundName:AlphaNum // Begin of the ground layer.
  AerialImage:AerialImage?// Aerial image for the ground.
  HeightImage:HeightImage? // Image for the height data.
  Surrounding? ";"; // Surroundings like houses or trees.

AerialImage =
  "AerialImage" Image:Image;// Data for the aerial image.

HeightImage =
  "HeightImage"  Image:Image// Data for the height image.
  "GroundHeight" GroundHeight:Number // Which color is used for ground level (0 m)?
  "MinHeight"    MinHeight:Number // Which height level in m is meant by color 0?
  "MaxHeight"    MaxHeight:Number; // Which height level in m is meant by color 255?

Image =
  "Image" ImageFile:AlphaNum // File name of the image.
  "OriginX" OriginX:Number // Origin of X in image coordinates (relative to upper/left corner).
  "OriginY" OriginY:Number // Origin of Y in image coordinates (relative to upper/left corner).
  "MPPX"    MeterPerPixelX:Number // Image's resolution per m in X direction.
  "MPPY"    MeterPerPixelY:Number // Image's resolution per m in Y direction.
  "RotZ"    Rotation:Number; // Counterclockwise rotation around Z-axis in RAD (3am is 0).
```

```
Surrounding =
  "Surrounding" // Begin of surroundings.
  Shapes:Shape+; // List of shapes.

Shape =
  "ShapeName" ShapeName:AlphaNum // Name of the shape.
  (Polygon | Cylinder | ComplexModel); // Type of the shape
  .

Polygon =
  ShapeType:"Polygon" // Either polygon (i.e. rectangular
  boxes, complex polygons)...
  "Height" Height:Number // Height of the polygon > 0.
  "Color"  Color:Vertex3 // RGB color.
  Vertex2 // A minimum of two vertices is necessary.
  Vertex2+;

Cylinder =
  ShapeType:"Cylinder" // ...or cylinders...
  Vertex2 // Center of the cylinder.
  "Radius" Radius:Number // Radius of the cylinder > 0.
  "Height" Height:Number // Height of the cylinder > 0.
  "Color"  Color:Vertex3; // RGB color.

ComplexModel =
  ShapeType:"ComplexModel" // ...or complex model to be
  loaded from file.
  "ModelFile" ModelFile:AlphaNum // File name of the model.
  "Position"  Position:Vertex3 // Position of the complex
  model (center of mass).
  "Rotation"  Rotation:Vertex3 // Counterclockwise rotation
   of the complex model in RAD (3am is 0).
  "BoundingBox" // The bounding box defines a rectangular
  outline around the complex model.
  Vertex2 // First: upper/left.
  Vertex2 // Second: upper/right.
  Vertex2 // Third: lower/right.
  Vertex2; // Fourth: lower/left.
```

```
Vertex2 =
  "Vertex2"
  "X" X:Number // X coordinate.
  "Y" Y:Number; // Y coordinate.

Vertex3 =
  "Vertex3"
  "X" X:Number // X coordinate.
  "Y" Y:Number // Y coordinate.
  "Z" Z:Number; // Z coordinate.

IDVertex2 =
  "ID" ID:Number // Integer number greater than 0.
  Vertex2; // Ordinary vertex2.

Layer =
  "Layer"   LayerName:AlphaNum // Begin of layer "LayerName↩
  ". Layers contain roads and are stacked to allow bridges.
  "LayerID" LayerID:Number // Integer identifier of this ↩
  layer greater than 0.
  "Height"  Height:Number // Height of this layer.
  Roads:Road+ // Roads.
  Zones:Zone* ";"; // Zones.

Road =
  "Road" // Begin of a road.
  "RoadID" RoadID:Number // Integer identifier of this road↩
   greater than 0.
  ("RoadName" RoadName:AlphaNum)? // Name of the road.
  Lanes:Lane+ ";"; // Each road contains at least one lane.

Lane =
  "Lane"
  "LaneID"  LaneID:Number // Integer identifier of this ↩
  lane greater than 0.
  LaneModel:LaneModel ";";// Which lane model to be used.

LaneModel =
  LaneAttribute:LaneAttribute // Lane's width as well as ↩
  lane markings.
```

```
  Connectors:Connector* // Connectors describe how lanes ↩
  are interconnected.
  TrafficControls:TrafficControl* // Traffic lights, ↩
  traffic signs...
  (PointModel | FunctionModel); // Either point- or ↩
  functionmodel.

TrafficControl =
  (TrafficLight | TrafficSign) // Either traffic light or ↩
  traffic sign.
  "Name" Name:AlphaNum // Name of the traffic control.
  IDVertex2 // Identifier and position inside the layer.
  Shape; // Shape of the traffic control.

TrafficLight =
  TrafficControlType:"TrafficLight"; // Either traffic ↩
  light...

TrafficSign =
  TrafficControlType:"TrafficSign" // ...or traffic sign.
  "Value" Value:TrafficSignType;

TrafficSignType =
  SignType:"stopline";

LaneAttribute =
  ("LaneWidth"  LaneWidth:Number)? // Lane's width.
  ("LeftLaneMarking"
  LeftLaneMarking:BoundaryStyle)? // Lane's left boundary ↩
  style.
  ("RightLaneMarking"
  RightLaneMarking:BoundaryStyle)?; // Lane's right ↩
  boundary style.

BoundaryStyle =
  Style:"double_yellow" | // Lane markings.
  Style:"solid_yellow" |
  Style:"solid_white" |
  Style:"broken_white" |
  Style:"crosswalk";
```

```
150
    Connector =
      Source:PointID "->" Target:PointID; // Directed ↩
      connection between two lane IDs.

    PointID =
155   PointID:FQID; // Layer-ID . Road-ID . Lane-ID . {Point|↩
      Function}-ID.

    PointModel =
      "PointModel" // The pointmodel consists of many
      IDVertex2+ ";" // points that are identifiable.
160
    FunctionModel =
      "FunctionModel"
      (StraightLine | Clothoid | Arc) ";" // Either straight ↩
      line, clothoid or arc.

165 StraightLine =
      FunctionModel:"StraightLine" // A straight line.
      "Start" Start:IDVertex2 // The start point.
      "End"   End:IDVertex2; // The end point.

170 Clothoid =
      FunctionModel:"Clothoid" // A clothoid.
      "dk"    dk:Number // Curvature change.
      "k"     k:Number // Curvature.
      "Start" Start:IDVertex2 // The start point.
175   "End"   End:IDVertex2 // The end point.
      "RotZ"  Rotation:Number; // Rotation around Z axis.

    Arc =
      FunctionModel:"Arc" // An arc in polar coordinates using ↩
      r and phi (= x).
180   "Radius" r:Number // Radius.
      "["      LeftBoundary:Number // The definition interval.
               RightBoundary:Number "]"
      "Start"  Start:IDVertex2 // The start point.
      "End"    End:IDVertex2 // The end point.
185   "RotZ"   Rotation:Number; // Rotation around Z axis.
```

```
  Zone =
    "Zone" // Begin of a zone.
    "ZoneID"    ZoneID:Number // Integer identifier of this ↩
    zone greater than 0.
    ("ZoneName" ZoneName:AlphaNum)? // Name of the zone.
    Connectors:Connector* // Connectors describe how lanes ↩
    are interconnected.
    Perimeter:Perimeter // Description of the bounding ↩
    polygon.
    Spots:Spot* ";"; // List of special spots in this zone.

  Perimeter =
    "Perimeter" // Begin of a perimeter.
    IDVertex2
    IDVertex2
    IDVertex2+ ";"; // At least three vertices describe a ↩
    perimeter.

  Spot =
    "Spot"
    "SpotID" SpotID:Number // Integer identifier of this zone↩
     greater than 0.
    Vertex2 // Two vertices determine orientation.
    Vertex2 ";";
}
```

Listing A.1: MontiCore grammar for stationary surroundings.

A.2 MontiCore Grammar for the Dynamic Surroundings

```
grammar Situation {
  ident AlphaNum "('A'..'Z'|'a'..'z')('A'..'Z'|'a'..'z'|'_↩
  '|'-'|'.'|'/'|'0'..'9')*";
  ident Number   "('+'|'-')?( (('1'..'9')('0'..'9')*) | '0' )↩
  ('.'('0'..'9')*)?";
  ident FQID     "'('(('1'..'9')('0'..'9')*)'.'(('1'..'9')↩
  ('0'..'9')*)|'0')'.'(('1'..'9')('0'..'9')*)|'0')↩
  '.'(('1'..'9')('0'..'9')*)')'";
```

```
5
  SituationFile =
    SituationHeader // File header.
    Object+ ";"; // At least one object.

10  SituationHeader =
    "Situation" SituationName:AlphaNum // Name of the ↩
    behavior.
    "Version"   Version:AlphaNum // Version of the behavior ↩
    grammar.
    "Date"      Date:AlphaNum // Creation date of the file.
    "Scenario"  Scenario:AlphaNum; // Associated scenario for↩
     this behavior.
15
  Object =
    "Object"    ObjectName:AlphaNum // Begin of object "↩
    ObjectName".
    "ObjectID"  ObjectID:Number // Integer identifier of this↩
     layer greater or equal than 0.
    Shape:Shape // Shape of this object.
20  "RotZ"      Rotation:Number // Rotation around Z axis ↩
    which defines the front.
    Behavior:Behavior ";"; // The behavior of this object.

  Shape =
    "ShapeName" ShapeName:AlphaNum // Name of the shape.
25  (Rectangle | Polygon | ComplexModel); // Type of the ↩
    shape.

  Rectangle =
    ShapeType:"Rectangle" // Either rectangles...
    "Height" Height:Number // Height of the cylinder > 0.
30  "Color"  Color:Vertex3 // RGB color.
    Front:Vertex2 // Front of the rectangle.
    "Length" Length:Number // Length.
    "Width"  Width:Number; // Width.
      // The construction of a rectangle is defined as:
35    //
      // ---------Length------------
      // |                          |
```

```
    // W                          |
    // i                          |      \
    // d                         X-------+
    // t                          |      /
    // h                          |
    // |                          |
    // ---------Length------------
    //
    // X = Front, + = ROTZ.

Polygon =
  ShapeType:"Polygon" // ...or polygons...
  "Height" Height:Number // Height of the polygon > 0.
  "Color"  Color:Vertex3 // RGB color.
  Front:Vertex2 // Front of the polygon.
  Vertex2+; // A minimum of two vertices is necessary.

ComplexModel =
  ShapeType:"ComplexModel" // ...or complex model to be loaded from file.
  "ModelFile" ModelFile:AlphaNum // File name of the model.
  Front:Vertex2 // Front of the complex model.
  "Position"  Position:Vertex3 // Position of the complex model (center of mass).
  "Rotation"  Rotation:Vertex3 // Counterclockwise rotation of the complex model in RAD (3am is 0).
  "BoundingBox" // The bounding box defines a rectangular outline around the complex model.
  Vertex2 // First: upper/left.
  Vertex2 // Second: upper/right.
  Vertex2 // Third: lower/right.
  Vertex2; // Fourth: lower/left.

Vertex2 =
  "Vertex2"
  "X" X:Number // X coordinate.
  "Y" Y:Number; // Y coordinate.

Vertex3 =
  "Vertex3"
```

```
  "X" X:Number // X coordinate.
  "Y" Y:Number // Y coordinate.
  "Z" Z:Number; // Z coordinate.

Behavior =
  "Behavior"
  (ExternalDriver | PointIDDriver);

ExternalDriver =
  BehaviorType:"ExternalDriver"; // External driver ↩
  behavior is realized by an external system instead of the↩
   simulation.

PointIDDriver =
  BehaviorType:"PointIDDriver" // PointID driver.
  StartType:StartType // When should this object get ↩
  started?
  StopType:StopType // What happens when this object ↩
  reaches the last point?
  Profile:Profile // Driving profile.
  PointIDs:PointID*; // List of points to be reached.

Profile =
  (ConstantVelocity | ConstantAcceleration);  // Either a ↩
  constant velocity or a constant acceleration.

ConstantVelocity =
  "ConstantVelocity"
  "V" V:Number; // Velocity in m/s.

ConstantAcceleration =
  "ConstantAcceleration"
  "A" A:Number; // Acceleration in m/s^2.

StartType =
  "StartType"
  (Immediately | OnMoving | OnEnteringPolygon);

StopType =
  "StopType"
```

```
    (Stop | ReturnToStart | WarpToStart);

  Immediately =
    "Immediately"; // Start immediately.

  OnMoving =
    "OnMoving"
    "ObjectID" ObjectID:Number; // Start when object ID ↩
    starts moving.

  OnEnteringPolygon =
    "OnEnteringPolygon"
    "ObjectID"  ObjectID:Number // Start when object ID ↩
    enters the polygon defined by at least four vertices.
    Vertex2
    Vertex2
    Vertex2
    Vertex2+;

  Stop = "Stop"; // Stop immediately.

  ReturnToStart = "ReturnToStart"; // Find a route to the ↩
  start point and return.

  WarpToStart = "WarpToStart";// "Warp" to the start point.

  PointID =  PointID:FQID; // Layer-ID . Road-ID . Lane-ID . ↩
  {Point|Function}-ID.
}
```

Listing A.2: MontiCore grammar for dynamic surroundings.

B List of Abbreviations

ACC ..... Adaptive Cruise Control, page 16
ADTF ..... Automotive Data and Time Triggered Framework, page 163
AGV ..... Autonomous Ground Vehicles, page 9
ANTLR ..... Another Tool for Language Recognition, page 45
API ..... Application Programming Interface, page 53
ASG ..... Abstract Syntax Graph, page 45
AST ..... Abstract Syntax Tree, page 49
AUTOSAR ..... Automotive Open System Architecture, page 3
BART ..... Berkeley Aachen Robotics Toolkit, page 7
BNF ..... Backus-Naur Form, page 47
C-ELROB ..... Civilian ELROB, page 13
CAN ..... Controller Area Network, page 70
CARMEN ..... Carnegie Mellon Robot Navigation Toolkit, page 168
CEO ..... Chief Executive Officer, page 9
CF ..... Compact Flash, page 137
CHESS ..... Center for Hybrid and Embedded Software Systems, page 7
CIS ..... Continuous Integration System, page 27
COM ..... Component Object Model, page 164
CORBA ..... Common Object Requesting Broker Architecture, page 169
CPS ..... Cyber Physical System, page 15
CPU ..... Central Processing Unit, page 54
CRC ..... Cyclic Redundancy Check, page 65
DARPA ..... Defense Advanced Research Projects Agency, page 165
DESS ..... Differential equation system specification, page 80
DEVS ..... Discrete event system specification, page 80
DHCP ..... Dynamic Host Configuration Protocol, page 68
DiL ..... Driver-in-the-Loop, page 171
DMCP ..... Dynamic Module Configuration Protocol, page 68
DSL ..... Domain Specific Language, page 5
DTSS ..... Discrete time system specification, page 80

EBNF ..... Extended Backus-Naur Form, page 45
ECU ..... Electronic Control Units, page 3
ELROB ..... European Land Robot Trial, page 13
FIFO ..... First-In-First-Out, page 29
FOV ..... Field of View, page 18
GLUT ..... OpenGL Utility Toolkit, page 117
GM ..... General Motors, page 9
GPL ..... General Purpose Languages, page 23
GPRMC ..... GPS Recommended Minimum Specific GPS/Transmit Data, page 71
GPS ..... Global Positioning System, page 167
GPU ..... Graphical Processing Unit, page 54
GUI ..... Graphical User Interface, page 164
HCSM ..... Hierarchical Concurrent State Machines, page 172
ICE ..... Internet Communications Engine, page 167
IDE ..... Integrated Development Environments, page 4
IDL ..... Interface Definition Language, page 169
IDS ..... Iowa Driving Simulator, page 171
IMTS ..... Intelligent Multi-mode Transit System, page 11
IPC ..... Inter-Process Communication, page 169
JAUS ..... Joint Architecture for Unmanned Systems, page 165
LCD ..... Liquid Crystal Display, page 138
LEX ..... lexical analyzer, page 47
LIDAR ..... Light Detection And Ranging, page 10
LIFO ..... Last In First Out, page 61
LIN ..... Local Interconnect Network, page 164
M-ELROB ..... Military ELROB, page 13
MDI ..... Multi Document Interface, page 114
MiL ..... Model-in-the-Loop, page 27
MiRPA ..... Middleware for Robotic and Process Control Applications, page 168
MOST ..... Media Oriented Systems Transport, page 164
NMEA ..... National Marine Electronics Association, page 71
NQE ..... National Qualification Event, page 11
NTP ..... Network Time Protocol, page 134
OBJX ..... Compressed Object Data Format, page 45
OCL ..... Object Constraint Language, page 24

OEM ..... Original Equipment Manufacturers, page 3
ORB ..... Object Request Broker, page 169
OS ..... Operating System, page 54
OSCAR ..... “Optically Steered Car”, page 10
PATH ..... Partners for Advanced Transit and Highways, page 11
PELOPS ..... “Programm zur Entwicklung längsdynamischer, mikroskopischer Prozesse in systemrelevanter Umgebung”, program for developing longitudinally dynamic, microscopic processes in system-relevant environment, page 174
POSIX ..... Portable Operating System Interface for Unix, page 55
PreScan ..... PRE-crash SCenario ANalyzer, page 176
PROMETHEUS ..... Program for European Traffic of Highest Efficiency and Unprecedented Safety, page 10
PTP ..... Precision Time Protocol, page 134
RALPH ..... “Rapidly Adapting Lateral Position Handler”, page 10
RF ..... Radio Frequency, page 9
RIF ..... Requirements Interchange Format, page 2
ROS ..... Robot Operating System, page 169
RSU ..... Road Side Units, page 19
RTE ..... Run-time-Environment, page 165
SAE ..... Society of Automotive Engineers, page 166
SCNX ..... Compressed Scenario Data Format, page 45
SiL ..... Software-in-the-Loop, page 27
SLAM ..... Simultaneously Localization And Mapping, page 40
SLICE ..... Specification Language ICE, page 167
SPAN ..... Synchronized Position & Attitude Navigation, page 137
SPOT ..... Single Point of Truth, page 24
SUD ..... System Under Development, page 4
SUV ..... Sports Utility Vehicle, page 137
TCP ..... Transmission Control Protocol, page 69
UDP ..... User Datagram Protocol, page 60
UML ..... Unified Modeling Language, page 24
UTA ..... “Urban Traffic Assistant”, page 10
VaMoR ..... “Versuchsfahrzeug zur autonomen Mobilität und Rechnersehen”, vehicle for autonomous mobility and computer vision, page 10
VaMP ..... “VaMoRs PKW”, VaMoR’s automobile, page 10
VeHiL ..... Vehicle-Hardware-in-the-Loop, page 177

VFB ..... Virtual Functional Bus, page 165
VITA ..... “Vision Technology Application”, page 10
VPL ..... Visual Programming Language, page 174
VRML ..... Virtual Reality Modeling Language, page 176
VSE ..... Visual Simulation Environment, page 174
W-LAN ..... Wireless-Local Area Network, page 137
WCS ..... World Coordinate System, page 42
WGS84 ..... World Geodetic System 1984, page 36
XSL ..... Extensible Style-sheet Language, page 184
YACC ..... Yet Another Compiler-Compiler, page 47

C Structure

D Curriculum Vitae

Name	Berger
Vorname	Christian
Geburtstag	24. Dezember 1980
Geburtsort	Braunschweig
Staatsangehörigkeit	deutsch

2008 - 2009	Wissenschaftlicher Mitarbeiter am Lehrstuhl für Software Engineering RWTH Aachen
2006 - 2008	Wissenschaftlicher Mitarbeiter am Institut für Software Systems Engineering TU Braunschweig
2006	Abschluss als Diplom-Wirtschaftsinformatiker
2001 - 2006	Studium der Wirtschaftsinformatik an der TU Braunschweig
2000 - 2001	Grundwehrdienst
2000	Abitur
1991 - 2000	Orientierungsstufe und Gymnasium in Braunschweig
1987 - 1991	Grundschule in Königslutter

Related Interesting Work from the SE Group, RWTH Aachen

Agile Model Based Software Engineering

Agility and modeling in the same project? This question was raised in [Rum04]: "Using an executable, yet abstract and multi-view modeling language for modeling, designing and programming still allows to use an agile development process." Modeling will be used in development projects much more, if the benefits become evident early, e.g with executable UML [Rum02] and tests [Rum03]. In [GKRS06], for example, we concentrate on the integration of models and ordinary programming code. In [Rum12] and [Rum11], the UML/P, a variant of the UML especially designed for programming, refactoring and evolution, is defined. The language workbench MontiCore [GKR+06] is used to realize the UML/P [Sch12]. Links to further research, e.g., include a general discussion of how to manage and evolve models [LRSS10], a precise definition for model composition as well as model languages [HKR+09] and refactoring in various modeling and programming languages [PR03]. In [FHR08] we describe a set of general requirements for model quality. Finally [KRV06] discusses the additional roles and activities necessary in a DSL-based software development project.

Generative Software Engineering

The UML/P language family [Rum12, Rum11] is a simplified and semantically sound derivate of the UML designed for product and test code generation. [Sch12] describes a flexible generator for the UML/P based on the MontiCore language workbench [KRV10, GKR+06]. In [KRV06], we discuss additional roles necessary in a model-based software development project. In [GKRS06] we discuss mechanisms to keep generated and handwritten code separated. In [Wei12] we show how this looks like and how to systematically derive a transformation language in concrete syntax. To understand the implications of executability for UML, we discuss needs and advantages of executable modeling with UML in agile projects in [Rum04], how to apply UML for testing in [Rum03] and the advantages and perils of using modeling languages for programming in [Rum02].

Unified Modeling Language (UML)

Many of our contributions build on UML/P described in the two books [Rum11] and [Rum12] are implemented in [Sch12]. Semantic variation points of the UML are discussed in [GR11]. We discuss formal semantics for UML [BHP+98] and describe UML semantics using the "System Model" [BCGR09a], [BCGR09b], [BCR07b] and [BCR07a]. Semantic variation points have, e.g., been applied to define class diagram semantics [CGR08]. A precisely defined semantics for variations is applied, when checking variants of class diagrams [MRR11c] and objects diagrams [MRR11d] or the consistency of both kinds of diagrams [MRR11e]. We also apply these concepts to activity diagrams (ADs) [MRR11b] which allows us to check for semantic differences of activity diagrams [MRR11a]. We also discuss how to ensure and identify model quality [FHR08], how models, views and the system under development correlate to each other [BGH+98] and how to use modeling in agile development projects [Rum04], [Rum02] The question how to adapt and extend the UML in discussed in [PFR02] on product line annotations for UML and to more general discussions and insights on how to use meta-modeling for defining and adapting the UML [EFLR99], [SRVK10].

Domain Specific Languages (DSLs)

Computer science is about languages. Domain Specific Languages (DSLs) are better to use, but need appropriate tooling. The MontiCore language workbench [GKR+06], [KRV10], [Kra10] describes an integrated abstract and concrete syntax format [KRV07b] for easy development. New languages and tools

can be defined in modular forms [KRV08, Völ11] and can, thus, easily be reused. [Wei12] presents a tool that allows to create transformation rules tailored to an underlying DSL. Variability in DSL definitions has been examined in [GR11]. A successful application has been carried out in the Air Traffic Management domain [ZPK+11]. Based on the concepts described above, meta modeling, model analyses and model evolution have been examined in [LRSS10] and [SRVK10]. DSL quality [FHR08], instructions for defining views [GHK+07], guidelines to define DSLs [KKP+09] and Eclipse-based tooling for DSLs [KRV07a] complete the collection.

Modeling Software Architecture & the MontiArc Tool

Distributed interactive systems communicate via messages on a bus, discrete event signals, streams of telephone or video data, method invocation, or data structures passed between software services. We use streams, statemachines and components [BR07] as well as expressive forms of composition and refinement [PR99] for semantics. Furthermore, we built a concrete tooling infrastructure called MontiArc [HRR12] for architecture design and extensions for states [RRW13]. MontiArc was extended to describe variability [HRR+11] using deltas [HRRS11] and evolution on deltas [HRRS12]. [GHK+07] and [GHK+08] close the gap between the requirements and the logical architecture and [GKPR08] extends it to model variants. Co-evolution of architecture is discussed in [MMR10] and a modeling technique to describe dynamic architectures is shown in [HRR98].

Compositionality & Modularity of Models

[HKR+09] motivates the basic mechanisms for modularity and compositionality for modeling. The mechanisms for distributed systems are shown in [BR07] and algebraically underpinned in [HKR+07]. Semantic and methodical aspects of model composition [KRV08] led to the language workbench MontiCore [KRV10] that can even develop modeling tools in a compositional form. A set of DSL design guidelines incorporates reuse through this form of composition [KKP+09]. [Völ11] examines the composition of context conditions respectively the underlying infrastructure of the symbol table. Modular editor generation is discussed in [KRV07a].

Semantics of Modeling Languages

The meaning of semantics and its principles like underspecification, language precision and detailedness is discussed in [HR04]. We defined a semantic domain called “System Model” by using mathematical theory. [RKB95, BHP+98] and [GKR96, KRB96]. An extended version especially suited for the UML is given in [BCGR09b] and in [BCGR09a] its rationale is discussed. [BCR07a, BCR07b] contain detailed versions that are applied on class diagrams in [CGR08]. [MRR11a, MRR11b] encode a part of the semantics to handle semantic differences of activity diagrams and [MRR11e] compares class and object diagrams with regard to their semantics. In [BR07], a simplified mathematical model for distributed systems based on black-box behaviors of components is defined. Meta-modeling semantics is discussed in [EFLR99]. [BGH+97] discusses potential modeling languages for the description of an exemplary object interaction, today called sequence diagram. [BGH+98] discusses the relationships between a system, a view and a complete model in the context of the UML. [GR11] and [CGR09] discuss general requirements for a framework to describe semantic and syntactic variations of a modeling language. We apply these on class and object diagrams in [MRR11e] as well as activity diagrams in [GRR10]. [Rum12] embodies the semantics in a variety of code and test case generation, refactoring and evolution techniques. [LRSS10] discusses evolution and related issues in greater detail.

Evolution & Transformation of Models

Models are the central artifact in model driven development, but as code they are not initially correct and need to be changed, evolved and maintained over time. Model transformation is therefore essential to effectively deal with models. Many concrete model transformation problems are discussed: evolution [LRSS10, MMR10, Rum04], refinement [PR99, KPR97, PR94], refactoring [Rum12, PR03], translating models from one language into another [MRR11c, Rum12] and systematic model transformation language development [Wei12]. [Rum04] describes how comprehensible sets of such transformations support software development, maintenance and [LRSS10] technologies for evolving models within a language and across languages and linking architecture descriptions to their implementation [MMR10]. Automaton refinement is discussed in [PR94, KPR97], refining pipe-and-filter architectures is explained in [PR99]. Refactorings of models are important for model driven engineering as discussed in [PR03, Rum12]. Translation between languages, e.g., from class diagrams into Alloy [MRR11c] allows for comparing class diagrams on a semantic level.

Variability & Software Product Lines (SPL)

Many products exist in various variants, for example cars or mobile phones, where one manufacturer develops several products with many similarities but also many variations. Variants are managed in a Software Product Line (SPL) that captures the commonalities as well as the differences. Feature diagrams describe variability in a top down fashion, e.g., in the automotive domain [GHK+08] using 150% models. Reducing overhead and associated costs is discussed in [GRJA12]. Delta modeling is a bottom up technique starting with a small, but complete base variant. Features are added (that sometimes also modify the core). A set of applicable deltas configures a system variant. We discuss the application of this technique to Delta-MontiArc [HRR+11, HRR+11] and to Delta-Simulink [HKM+13]. Deltas can not only describe spacial variability but also temporal variability which allows for using them for software product line evolution [HRRS12]. [HHK+13] describes an approach to systematically derive delta languages. We also apply variability to modeling languages in order to describe syntactic and semantic variation points, e.g., in UML for frameworks [PFR02]. And we specified a systematic way to define variants of modeling languages [CGR09] and applied this as a semantic language refinement on Statecharts in [GR11].

Cyber-Physical Systems (CPS)

Cyber-Physical Systems (CPS) [KRS12] are complex, distributed systems which control physical entities. Contributions for individual aspects range from requirements [GRJA12], complete product lines [HRRW12], the improvement of engineering for distributed automotive systems [HRR12] and autonomous driving [BR12a] to processes and tools to improve the development as well as the product itself [BBR07]. In the aviation domain, a modeling language for uncertainty and safety events was developed, which is of interest for the European airspace [ZPK+11]. A component and connector architecture description language suitable for the specific challenges in robotics is discussed in [RRW13]. Monitoring for smart and energy efficient buildings is developed as Energy Navigator toolset [KPR12, FPPR12, KLPR12].

State Based Modeling (Automata)

Today, many computer science theories are based on state machines in various forms including Petri nets or temporal logics. Software engineering is particularly interested in using state machines for modeling systems. Our contributions to state based modeling can currently be split into three parts: (1) understanding how to model object-oriented and distributed software using statemachines resp. Statecharts

[GKR96, BCR07b, BCGR09b, BCGR09a], (2) understanding the refinement [PR94, RK96, Rum96] and composition [GR95] of statemachines, and (3) applying statemachines for modeling systems. In [Rum96] constructive transformation rules for refining automata behavior are given and proven correct. This theory is applied to features in [KPR97]. Statemachines are embedded in the composition and behavioral specifications concepts of Focus [BR07]. We apply these techniques, e.g., in MontiArcAutomaton [THR+13] as well as in building management systems [FLP+11].

Robotics

Robotics can be considered a special field within Cyber-Physical Systems which is defined by an inherent heterogeneity of involved domains, relevant platforms, and challenges. The engineering of robotics applications requires composition and interaction of diverse distributed software modules. This usually leads to complex monolithic software solutions hardly reusable, maintainable, and comprehensible, which hampers broad propagation of robotics applications. The MontiArcAutomaton language [RRW12] extends ADL MontiArc and integrates various implemented behavior modeling languages using MontiCore [RRW13] that perfectly fits Robotic architectural modelling. The LightRocks [THR+13] framework allows robotics experts and laymen to model robotic assembly tasks.

Automotive, Autonomic Driving & Intelligent Driver Assistance

Introducing and connecting sophisticated driver assistance, infotainment and communication systems as well as advanced active and passive safety-systems result in complex embedded systems. As these feature-driven subsystems may be arbitrarily combined by the customer, a huge amount of distinct variants needs to be managed, developed and tested. A consistent requirements management that connects requirements with features in all phases of the development for the automotive domain is described in [GRJA12]. The conceptual gap between requirements and the logical architecture of a car is closed in [GHK+07, GHK+08]. [HKM+13] describes a tool for delta modeling for Simulink [HKM+13]. [HRRW12] discusses means to extract a well-defined Software Product Line from a set of copy and paste variants. Quality assurance, especially of safety-related functions, is a highly important task. In the Carolo project [BR12a, BR12b], we developed a rigorous test infrastructure for intelligent, sensor-based functions through fully-automatic simulation [BBR07]. This technique allows a dramatic speedup in development and evolution of autonomous car functionality, and thus, enables us to develop software in an agile way [BR12a]. [MMR10] gives an overview of the current state-of-the-art in development and evolution on a more general level by considering any kind of critical system that relies on architectural descriptions. As tooling infrastructure, the SSElab storage, versioning and management services [HKR12] are essential for many projects.

Energy Management

In the past years, it became more and more evident that saving energy and reducing CO2 emissions is an important challenge. Thus, energy management in buildings as well as in neighbourhoods becomes equally important to efficiently use the generated energy. Within several research projects, we developed methodologies and solutions for integrating heterogeneous systems at different scales. During the design phase, the Energy Navigators Active Functional Specification (AFS) [FPPR12, KPR12] is used for technical specification of building services already. We adapted the well-known concept of statemachines to be able to describe different states of a facility and to validate it against the monitored values [FLP+11]. We show how our data model, the constraint rules and the evaluation approach to compare sensor data can be applied [KLPR12].

Cloud Computing & Enterprise Information Systems

The paradigm of Cloud Computing is arising out of a convergence of existing technologies for web-based application and service architectures with high complexity, criticality and new application domains. It promises to enable new business models, to lower the barrier for web-based innovations and to increase the efficiency and cost-effectiveness of web development. Application classes like Cyber-Physical Systems [KRS12], Big Data, App and Service Ecosystems bring attention to aspects like responsiveness, privacy and open platforms. Regardless of the application domain, developers of such systems are in need for robust methods and efficient, easy-to-use languages and tools. We tackle these challenges by perusing a model-based, generative approach [PR13]. The core of this approach are different modeling languages that describe different aspects of a cloud-based system in a concise and technology-agnostic way. Software architecture and infrastructure models describe the system and its physical distribution on a large scale. We apply cloud technology for the services we develop, e.g., the SSELab [HKR12] and the Energy Navigator [FPPR12, KPR12] but also for our tool demonstrators and our own development platforms. New services, e.g.,c collecting data from temperature, cars etc. are easily developed.

References

[BBR07] Christian Basarke, Christian Berger, and Bernhard Rumpe. Software & Systems Engineering Process and Tools for the Development of Autonomous Driving Intelligence. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 4(12):1158–1174, October 2007.

[BCGR09a] Manfred Broy, Maria Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Considerations and Rationale for a UML System Model. In Kevin Lano, editor, *UML 2 Semantics and Applications*, pages 43–61. John Wiley & Sons, 2009.

[BCGR09b] Manfred Broy, Maria Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Definition of the UML System Model. In Kevin Lano, editor, *UML 2 Semantics and Applications*, pages 63–93. John Wiley & Sons, 2009.

[BCR07a] Manfred Broy, Maria Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 2: The Control Model. Technical Report TUM-I0710, TU Munich, February 2007.

[BCR07b] Manfred Broy, Maria Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 3: The State Machine Model. Technical Report TUM-I0711, TU Munich, February 2007.

[BGH+97] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and Complete Object Interaction Descriptions. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Proceedings OOPSLA'97 Workshop on Object-oriented Behavioral Semantics*, TUM-I9737, TU Munich, 1997.

[BGH+98] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Systems, Views and Models of UML. In M. Schader and A. Korthaus, editors, *Proceedings of the Unified Modeling Language, Technical Aspects and Applications*. Physica Verlag, Heidelberg, 1998.

[BHP+98] Manfred Broy, Franz Huber, Barbara Paech, Bernhard Rumpe, and Katharina Spies. Software and System Modeling Based on a Unified Formal Semantics. In M. Broy and B. Rumpe, editors, *RTSE '97: Proceedings of the International Workshop on Requirements Targeting Software and Systems Engineering*, LNCS 1526, pages 43–68, Bernried, Germany, October 1998. Springer.

[BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, Februar 2007.

[BR12a] Christian Berger and Bernhard Rumpe. Autonomous Driving - 5 Years after the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System. In *Proceedings of the 10th Workshop on Automotive Software Engineering (ASE 2012)*, pages 789–798, Braunschweig, Germany, September 2012.

[BR12b] Christian Berger and Bernhard Rumpe. Engineering Autonomous Driving Software. In C. Rouff and M. Hinchey, editors, *Experience from the DARPA Urban Challenge*. Springer, 2012.

[CGR08] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, CfG Fakultät, TU Braunschweig, 2008.

[CGR09] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within Modeling Language Definitions. In *Model Driven Engineering Languages and Systems. Proceedings of MODELS 2009*, LNCS 5795, pages 670–684, Denver, Colorado, USA, October 2009.

[EFLR99] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Meta-Modelling Semantics of UML. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*. Kluver Academic Publisher, 1999.

[FHR08] Florian Fieber, Michaela Huhn, and Bernhard Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik-Spektrum*, 31(5):408–424, Oktober 2008.

[FLP+11] Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. State-Based Modeling of Buildings and Facilities. In *Proceedings of the 11th International Conference for Enhanced Building Operations (ICEBO' 11)*, New York City, USA, October 2011.

[FPPR12] Norbert Fisch, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. The Energy Navigator - A Web-Platform for Performance Design and Management. In *Proceedings of the 7th International Conference on Energy Efficiency in Commercial Buildings (IEECB)*, Frankfurt a. M., Germany, April 2012.

[GHK+07] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, and Bernhard Rumpe. View-based Modeling of Function Nets. In *Proceedings of the Object-oriented Modelling of Embedded Real-Time Systems (OMER4) Workshop*, Paderborn, Germany, October 2007.

[GHK+08] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In *Proceedings of 4th European Congress ERTS - Embedded Real Time Software*, Toulouse, 2008.

[GKPR08] Hans Grönniger, Holger Krahn, Claas Pinkernell, and Bernhard Rumpe. Modeling Variants of Automotive Systems using Views. In *Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen (MBEFF)*, Informatik Bericht 2008-01, pages 76–89, CFG Fakultät, TU Braunschweig, March 2008.

[GKR96] Radu Grosu, Cornel Klein, and Bernhard Rumpe. Enhancing the SysLab System Model with State. Technical Report TUM-I9631, TUM, Munich, Germany, 1996.

[GKR+06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänspezifischer Sprachen. Technical Report 2006-04, CfG Fakultät, TU Braunschweig, August 2006.

[GKRS06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, and Martin Schindler. Integration von Modellen in einen codebasierten Softwareentwicklungsprozess. In *Proceedings der Modellierung 2006*, Lecture Notes in Informatics LNI P-82, Innsbruck, März 2006. GI-Edition.

[GR95] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical Report TUM-I9533, TUM, Munich, Germany, 1995.

[GR11] Hans Grönniger and Bernhard Rumpe. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems. 16th Monterey Workshop*, LNCS 6662, pages 17–32, Redmond, Microsoft Research, 2011. Springer.

[GRJA12] Tim Gülke, Bernhard Rumpe, Martin Jansen, and Joachim Axmann. High-Level Requirements Management and Complexity Costs in Automotive Development Projects: A Problem Statement. In *Requirements Engineering: Foundation for Software Quality. 18th International Working Conference, Proceedings, REFSQ 2012*, Essen, Germany, March 2012.

[GRR10] Hans Grönniger, Dirk Reiß, and Bernhard Rumpe. Towards a Semantics of Activity Diagrams with Semantic Variation Points. In *Model Driven Engineering Languages and Systems, Proceedings of MODELS*, LNCS 6394, Oslo, Norway, 2010. Springer.

[HHK+13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, and Ina Schaefer. Engineering Delta Modeling Languages. In *Proceedings of the 17th International Software Product Line Conference (SPLC), Tokyo*, pages 22–31. ACM, September 2013.

[HKM+13] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. First-Class Variability Modeling in Matlab / Simulink. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, pages 11–18, New York, NY, USA, 2013. ACM.

[HKR+07] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An Algebraic View on the Semantics of Model Composition. In D. H. Akehurst, R. Vogel, and R. F. Paige, editors, *Proceedings of the Third European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2007), Haifa, Israel*, pages 99–113. Springer, 2007.

[HKR+09] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Scaling-Up Model-Based-Development for Large Heterogeneous Systems with Compositional Modeling. In H. Arabnia and H. Reza, editors, *Proceedings of the 2009 International Conference on Software Engineeering in Research and Practice*, Las Vegas, Nevada, USA, 2009.

[HKR12] Christoph Herrmann, Thomas Kurpick, and Bernhard Rumpe. SSELab: A Plug-In-Based Framework for Web-Based Project Portals. In *Proceedings of the 2nd International Workshop on Developing Tools as Plug-Ins (TOPI) at ICSE 2012*, pages 61–66, Zurich, Switzerland, June 2012. IEEE.

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, Oct 2004.

[HRR98] Franz Huber, Andreas Rausch, and Bernhard Rumpe. Modeling Dynamic Component Interfaces. In Madhu Singh, Bertrand Meyer, Joseph Gil, and Richard Mitchell, editors, *TOOLS 26, Technology of Object-Oriented Languages and Systems*. IEEE Computer Society, 1998.

[HRR+11] Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer, and Frank van der Linden. Hierarchical Variability Modeling for Software Architectures. In *Proceedings of International Software Product Lines Conference (SPLC 2011)*. IEEE Computer Society, August 2011.

[HRR12] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen, February 2012.

[HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta Modeling for Software Architectures. *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VII, fortiss GmbH*, February 2011.

[HRRS12] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Evolving Delta-oriented Software Product Line Architectures. In *Large-Scale Complex IT Systems. Development, Operation and Management, 17th Monterey Workshop 2012*, LNCS 7539, pages 183–208, Oxford, UK, March 2012. Springer.

[HRRW12] Christian Hopp, Holger Rendel, Bernhard Rumpe, and Fabian Wolf. Einführung eines Produktlinienansatzes in die automotive Softwareentwicklung am Beispiel von Steuergerätesoftware. In *Software Engineering 2012: Fachtagung des GI-Fachbereichs Softwaretechnik in Berlin*, Lecture Notes in Informatics LNI 198, pages 181–192, 27. Februar - 2. März 2012.

[KKP+09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM'09), Sprinkle, J., Gray, J., Rossi, M., Tolvanen, J.-P., (eds.)*, Techreport B-108, Helsinki School of Economics, Orlando, Florida, USA, October 2009.

[KLPR12] Thomas Kurpick, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Modeling Cyber-Physical Systems: Model-Driven Specification of Energy Efficient Buildings. In *Proceedings of the Modelling of the Physical World Workshop MOTPW'12, Innsbruck, October 2012*, pages 2:1–2:6. ACM Digital Library, October 2012.

[KPR97] Cornel Klein, Christian Prehofer, and Bernhard Rumpe. Feature Specification and Refinement with State Transition Diagrams. In *Fourth IEEE Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems*. P. Dini, IOS-Press, 1997.

[KPR12] Thomas Kurpick, Claas Pinkernell, and Bernhard Rumpe. Der Energie Navigator. In *Entwicklung und Evolution von Forschungssoftware. Tagungsband, Rolduc, 10.-11.11.2011*, Aachener Informatik-Berichte, Software Engineering Band 14. Shaker Verlag Aachen, 2012.

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering Band 1. Shaker Verlag, Aachen, Germany, 2010.

[KRB96] Cornel Klein, Bernhard Rumpe, and Manfred Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model. In *Proceedings of the first International Workshop on Formal Methods for Open Object-based Distributed Systems*, pages 323–338. Chapmann & Hall, 1996.

[KRS12] Stefan Kowalewski, Bernhard Rumpe, and Andre Stollenwerk. Cyber-Physical Systems - eine Herausforderung für die Automatisierungstechnik? In *Proceedings of Automation 2012, VDI Berichte 2012*, pages 113–116. VDI Verlag, 2012.

[KRV06] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In J. Gray, J.-P. Tolvanen, and J. Sprinkle, editors, *Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling 2006 (DSM'06), Portland, Oregon USA*, Technical Report TR-37, pages 150–158, Jyväskylä University, Finland, 2006.

[KRV07a] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM' 07), Montreal, Quebec, Canada*, Technical Report TR-38, pages 8–10, Jyväskylä University, Finland, 2007.

[KRV07b] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, editors, *Proceedings of the ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MODELS 2007), Nashville, TN, USA, October 2007*, LNCS 4735. Springer, 2007.

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In R. F. Paige and B. Meyer, editors, *Proceedings of the 46th International Conference Objects, Models, Components, Patterns (TOOLS-Europe), Zurich, Switzerland, 2008*, Lecture Notes in Business Information Processing LNBIP 11, pages 297–315. Springer, 2008.

[KRV10] Holger Krahn, Bernhard Rumpe, and Stefen Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010.

[LRSS10] Tihamer Levendovszky, Bernhard Rumpe, Bernhard Schätz, and Jonathan Sprinkle. Model Evolution and Management. In *MBEERTS: Model-Based Engineering of Embedded Real-Time Systems, International Dagstuhl Workshop, Dagstuhl Castle, Germany*, LNCS 6100, pages 241–270. Springer, October 2010.

[MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer*, 43(5):42–48, May 2010.

[MRR11a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *Proc. Euro. Soft. Eng. Conf. and SIGSOFT Symp. on the Foundations of Soft. Eng. (ESEC/FSE'11)*, pages 179–189. ACM, 2011.

[MRR11b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. An Operational Semantics for Activity Diagrams using SMV. Technical Report AIB-2011-07, RWTH Aachen University, Aachen, Germany, July 2011.

[MRR11c] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In *Model Driven Engineering Languages and Systems (MODELS 2011), Wellington, New Zealand*, LNCS 6981, pages 592–607, 2011.

[MRR11d] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Modal Object Diagrams. In *Proc. 25th Euro. Conf. on Object Oriented Programming (ECOOP'11)*, LNCS 6813, pages 281–305. Springer, 2011.

[MRR11e] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Semantically Configurable Consistency Analysis for Class and Object Diagrams. In *Model Driven Engineering Languages and Systems (MODELS 2011), Wellington, New Zealand*, LNCS 6981, pages 153–167. Springer, 2011.

[PFR02] Wolfgang Pree, Marcus Fontoura, and Bernhard Rumpe. Product Line Annotations with UML-F. In G. J. Chastek, editor, *Software Product Lines - Second International Conference, SPLC 2*, LNCS 2379, pages 188–197, San Diego, 2002. Springer.

[PR94] Barbara Paech and Bernhard Rumpe. A new Concept of Refinement used for Behaviour Modelling with Automata. In M. Naftalin, T. Denvir, and M. Bertran, editors, *FME'94: Industrial Benefit of Formal Methods*, LNCS 873. Springer, October 1994.

[PR99] Jan Philipps and Bernhard Rumpe. Refinement of Pipe-and-Filter Architectures. In J. Davies J. M. Wing, J. Woodcock, editor, *FM'99 - Formal Methods, Proceedings of the World Congress on Formal Methods in the Development of Computing System*, LNCS 1708, pages 96–115. Springer, 1999.

[PR03] Jan Philipps and Bernhard Rumpe. Refactoring of Programs and Specifications. In H. Kilov and K. Baclawski, editors, *Practical foundations of business and system specifications*, pages 281–297. Kluwer Academic Publishers, 2003.

[PR13] Antonio Navarro Perez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In I. Ober, A. S. Gokhale, J. H. Hill, J. Bruel, M. Felderer, D. Lugato, and A. Dabholka, editors, *Proc. of the 2nd International Workshop on Model-Driven Engineering for High Performance and Cloud Computing. Co-located with MODELS 2013, Miami*, Sun SITE Central Europe Workshop Proceedings CEUR 1118, pages 15–24. CEUR-WS.org, 2013.

[RK96] Bernhard Rumpe and Cornel Klein. Automata Describing Object Behavior. In H. Kilov and W. Harvey, editors, *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, pages 265–286. Kluwer Academic Publishers, 1996.

[RKB95] Bernhard Rumpe, Cornel Klein, and Manfred Broy. Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme - Syslab-Systemmodell. Technical Report TUM-I9510, Technische Universität München, 1995.

[RRW12] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Requirements Modeling Language for the Component Behavior of Cyber Physical Robotics Systems. In N. Seyff and A. Koziolek, editors, *Modelling and Quality in Requirements Engineering: Essays Dedicated to Martin Glinz on the Occasion of His 60th Birthday*. Monsenstein und Vannerdat, Münster, 2012.

[RRW13] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Workshops and Tutorials Proceedings of the 2013 IEEE International Conference on Robotics and Automation (ICRA), May 6-10, 2013, Karlsruhe, Germany*, pages 10–12, 2013.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, ISBN 3-89675-149-2, 1996.

[Rum02] Bernhard Rumpe. Executable Modeling with UML - A Vision or a Nightmare? In T. Clark and J. Warmer, editors, *Issues & Trends of Information Technology Management in Contemporary Associations, Seattle*, pages 697–701. Idea Group Publishing, Hershey, London, 2002.

[Rum03] Bernhard Rumpe. Model-Based Testing of Object-Oriented Systems. In F. de Boer, M. Bonsangue, S. Graf, W.-P. de Roever, editor, *Formal Methods for Components and Objects*, LNCS 2852, pages 380–402. Springer, November 2003.

[Rum04] Bernhard Rumpe. Agile Modeling with the UML. In M. Wirsing, A. Knapp, and S. Balsamo, editors, *Radical Innovations of Software and Systems Engineering in the Future. 9th International Workshop, RISSEF 2002. Venice, Italy, October 2002*, LNCS 2941. Springer, October 2004.

[Rum11] Bernhard Rumpe. *Modellierung mit UML*. Springer, second edition, September 2011.

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring*. Springer, second edition, Juni 2012.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering Band 11. Shaker Verlag, Aachen, Germany, 2012.

[SRVK10] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. Metamodelling: State of the Art and Research Challenges. In *MBEERTS: Model-Based Engineering of Embedded Real-Time Systems, International Dagstuhl Workshop, Dagstuhl Castle, Germany*, LNCS 6100, pages 57–76, October 2010.

[THR+13] Ulrike Thomas, Gerd Hirzinger, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. A New Skill Based Robot Programming Language Using UML/P Statecharts. In *Proceedings of the 2013 IEEE International Conference on Robotics and Automation (ICRA)*, pages 461–466, Karlsruhe, Germany, May 2013. IEEE.

[Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering Band 9. Shaker Verlag, Aachen, Germany, 2011.

[Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering Band 12. Shaker Verlag, Aachen, Germany, 2012.

[ZPK+11] Massimiliano Zanin, David Perez, Dimitrios S Kolovos, Richard F Paige, Kumardev Chatterjee, Andreas Horst, and Bernhard Rumpe. On Demand Data Analysis and Filtering for Inaccurate Flight Trajectories. In D. Schaefer, editor, *Proceedings of the SESAR Innovation Days*. EUROCONTROL, November 2011.