Test-Driven Development of Embedded Control Systems: Application in an Automotive Collision Prevention System

Thomas Dohmke

UNIVERSITY
of
GLASGOW

Submitted for the degree of Doctor of Philosophy in the
Department of Mechanical Engineering, Faculty of Engineering,
University of Glasgow, February, 2008

Abstract

With test-driven development (TDD) new code is not written until an automated test has failed, and duplications of functions, tests, or simply code fragments are always removed. TDD can lead to a better design and a higher quality of the developed system, but to date it has mainly been applied to the development of traditional software systems such as payroll applications. This thesis describes the novel application of TDD to the development of embedded control systems using an automotive safety system for preventing collisions as an example.

The basic prerequisite for test-driven development is the availability of an automated testing framework as tests are executed very often. Such testing frameworks have been developed for nearly all programming languages, but not for the graphical, signal driven language Simulink. Simulink is commonly used in the automotive industry and can be considered as state-of-the-art for the design and development of embedded control systems in the automotive, aerospace and other industries. The thesis therefore introduces a novel automated testing framework for Simulink. This framework forms the basis for the test-driven development process by integrating the analysis, design and testing of embedded control systems into this process.

The thesis then shows the application of TDD to a collision prevention system. The system architecture is derived from the requirements of the system and four software components are identified, which represent problems of particular areas for the realisation of control systems, i.e. logical combinations, experimental problems, mathematical algorithms, and control theory. For each of these problems, a concept to systematically derive test cases from the requirements is presented. Moreover two conventional approaches to design the controller are introduced and compared in terms of their stability and performance.

The effectiveness of the collision prevention system is assessed in trials on a driving simulator. These trials show that the system leads to a significant reduction of the accident rate for rear-end collisions. In addition, experiments with prototype vehicles on test tracks and field tests are presented to verify the system's functional requirements within a system testing approach. Finally, the new test-driven development process for embedded control systems is evaluated in comparison to traditional development processes.

Contents

1 Introduction **1**

1.1 Automotive Safety Systems . . . 2
1.1.1 History . . . 2
1.1.2 Design of Automotive Safety Systems . . . 3
1.2 Control System Design . . . 5
1.2.1 History . . . 5
1.2.2 The Basic Feedback Loop . . . 6
1.2.3 Process of Control System Design . . . 7
1.3 Software Development Processes . . . 9
1.3.1 History . . . 9
1.3.2 Test-Driven Development Process . . . 12
1.3.3 Design Patterns . . . 13
1.4 Thesis Contributions . . . 15
1.5 Thesis Outline . . . 18

2 Methods and Tools **20**

2.1 Model-Based Development . . . 20
2.1.1 Definition . . . 20
2.1.2 Overview of Simulink . . . 24
2.2 Model-Based Testing Framework . . . 26
2.2.1 Testing Frameworks for Simulink . . . 26
2.2.2 The xUnit Family . . . 28
2.2.3 Assertions . . . 29
2.2.4 Creating and Executing Tests . . . 32
2.2.5 Fixtures . . . 36
2.2.6 Exceptions . . . 38
2.2.7 Further Patterns for slUnit . . . 39
2.2.8 Summary . . . 40
2.3 Using Tests for Verification . . . 41
2.3.1 Testing Techniques . . . 42
2.3.2 Test Design Techniques . . . 45
2.3.3 Test Evaluation Techniques . . . 50

2.4 Using Tests for Design . . . 53
2.5 Testing Different Layers of an Automotive Safety System . . . 56

3 Development of a Collision Prevention System 59
3.1 Collision Prevention Systems . . . 60
3.1.1 Overview . . . 60
3.1.2 Definition . . . 60
3.2 Requirements . . . 63
3.3 Architecture . . . 65
3.3.1 State-Space Approach . . . 66
3.3.2 Layer Approach . . . 70
3.3.3 Comparison of the State-Space Approach and the Layer Approach 72
3.4 Realisation . . . 74
3.4.1 Activation . . . 74
3.4.2 Driver Assessment . . . 82
3.4.3 Situation Assessment . . . 88
3.5 Summary . . . 95

4 Development of a Longitudinal Vehicle Controller 96
4.1 Controllers for Longitudinal Control . . . 96
4.2 Conventional Approach . . . 100
4.2.1 Synthesis with Pole Placement and LQG Optimisation . . . 100
4.2.2 Stability and Robustness . . . 101
4.2.3 Performance . . . 102
4.2.4 Choice of the Parameters . . . 105
4.3 Test-Driven Development Approach . . . 109
4.3.1 Concept . . . 109
4.3.2 Basic Implementation . . . 110
4.3.3 Extended Implementation . . . 115
4.4 Comparison . . . 123
4.4.1 Stability and Robustness . . . 123
4.4.2 Performance . . . 125
4.4.3 Conclusions . . . 127
4.5 Summary . . . 130

5 Evaluation of the Collision Prevention System 132
5.1 Trials on a Driving Simulator . . . 132
5.1.1 Methods . . . 132

5.1.2 Results 134
5.2 Experiments with Prototype Vehicles 134
5.2.1 Introduction 134
5.2.2 System Testing on Test Tracks 135
5.2.3 Acceptance Testing in Real Life 137
5.3 Summary 140

6 Analysis of the Test-Driven Development Process 142
6.1 Comparison to Design Patterns 142
6.1.1 TDD Tuning Process 143
6.1.2 TDD Design Process 146
6.2 Comparison to the Traditional V-Model Approach 150

7 Conclusions and Outlook 154
7.1 Conclusions 154
7.2 Outlook and Further Work 156

List of Figures

1.1 Milestones of automotive safety systems 3
1.2 Sensors for environment detection 4
1.3 The basic feedback loop 7
1.4 The process of control system design 8
1.5 The Waterfall model 10
1.6 The Spiral model 10
1.7 The Multiple V-model 11
1.8 Comparing the cycle times of the different development models 12
1.9 The development cycle of test-driven development 13

2.1 Different models for the same activation subsystem 22
2.2 Basic Elements of Simulink 25
2.3 Core classes of the xUnit architecture 28
2.4 Behaviour of the Assert and Assert State Change block 32
2.5 Example for a user-defined assertion 32
2.6 Tree of Test Suite and Test Case Objects 34
2.7 Architecture of a slUnit model 35
2.8 Example for a slUnit system 36
2.9 The four-phase test with xUnit and slUnit 41
2.10 Different test setups for embedded systems 43
2.11 Relationship between different testing techniques 45
2.12 Example classification tree for a collision avoidance system 47
2.13 Example control-flow model with labelled transitions 49
2.14 Difference between regression and back-to-back testing 52
2.15 Acceptance and unit tests in test-driven development 55
2.16 Layers for an automotive safety system 58

3.1 Definition of a rear-end collision 62
3.2 Red light with a triangular shape in the instrument cluster 64
3.3 Block diagram of the state-space representation 66
3.4 State-space approach for the Collision Prevention System 69
3.5 Layer 2: Architecture of the control units 70

3.6 Layer 3: Architecture of the software components 71
3.7 Layer 3: Architecture of the CPS subcomponents 72
3.8 Combination of the layer and the state-space approach 73
3.9 Concept for deriving tests from the classification tree 75
3.10 Classification tree for the component Activation 76
3.11 Test setup with a foam object 77
3.12 Test sequence with the classification-tree method 79
3.13 Test sequence with slUnit 80
3.14 The realisation of the component Activation 81
3.15 Concept for using tests for experimental problems 83
3.16 Example trajectories for braking manoeuvres 85
3.17 Classification of braking manoeuvres 86
3.18 Defining the threshold by a synthetic trajectory 87
3.19 Implementation of the component Driver Assessment 88
3.20 Concept for deriving tests for mathematical problems 90
3.21 Definition of the reaction time 90
3.22 Motions of System Vehicle and Object: Reaction time exists 91
3.23 Motions of System Vehicle and Object: Reaction time not exists . . . 92
3.24 Implementation of the component Situation Assessment 95

4.1 Range-vs-Range-Rate diagram 97
4.2 Virtual Bumper approach 98
4.3 Concept of fuzzy control for a longitudinal vehicle controller 99
4.4 Relationship between the peak value and a collision 103
4.5 Influence of λ 107
4.6 Concept for deriving tests for control systems 110
4.7 Simplified control loop with only one state 111
4.8 Manipulating the Object's acceleration to prevent the collision 117
4.9 Implementation of the component Controller 120
4.10 Plots for different values of K_{ref} 122
4.11 Simulation for the initial configuration of Equation (4.50) 126
4.12 Simulation for the initial configuration of Equation (4.51) 127
4.13 Simulation for the initial configuration of Equation (4.52) 128
4.14 Simulation for different values of Δv and Δs 129

5.1 Results from the Driving Simulator 134
5.2 Verification and reproduction of measurement data 136

5.3 Comparison between simulation results and measurement data 137
5.4 Two examples for real situations 139

6.1 Origins of a new test 146
6.2 The Model-View-Controller pattern 147
6.3 Test-Driven Development as a Design Process 149
6.4 The Test-Driven Process 152
6.5 Usage of different process models 153

List of Tables

2.1 Modelling tools for production purposes 24
2.2 The architectures of xUnit and slUnit 41

4.1 Gain and phase margins of the different controllers 124
4.2 Comparison of performance criteria for the three examples 129

Acknowledgements

It feels like a long time that has passed since I started my PhD project five years ago. A time, in which I witnessed and actively participated in the development of a new car. A time, in which I visited countries all over the world and stayed abroad for an average of 14 weeks each year. A time, in which I met a wonderful woman and, on a beautiful and unforgettable day, married her. This page is dedicated to the people, who have been part of my live during this time and helped me, not to loose focus on my thesis.

First and foremost I would like to acknowledge my supervisors Dr Henrik Gollee and Prof Ken Hunt for their advice, patience and guidance over the last years. Their detailed knowledge about control theory was very helpful and kept me continuously rethinking my ideas and the concepts of my thesis. Sometimes it was hard to bridge the distance between Scotland and Germany by means of modern communication, but especially Henrik's questions and the resulting discussions have been of tremendous benefit to me. I couldn't have asked for better supervision and I'm very grateful.

Furthermore I wish to acknowledge Dr Volker Schmid, who has been my project leader at DaimlerChrysler from the beginnings in 2003 until I left the company at the end of 2006. Together, we had a great spirit to drive the project forward and the feeling that nobody can stop us. I will never forget some key moments: when the system was running in the car for the first time in 2003, when we did a test-drive (or was it a roadtrip?) through the whole of the USA in 2004, and of course, when we presented the final system to journalists in 2005.

Finally, I am very appreciative of my wife Sarah, who was a big support in every moment I can remember: when I came home from office and immediately sat down in front of my laptop every day during the last one and half years; when I worked on my thesis on most weekends, without doing my part of taking care of the household; when I was frustrated about control theory, somehow demotivated, or distracted by browsing through the internet and discovering interesting things which had nothing to do with my thesis. Sarah, I owe you a lot of hugs and I will always love you.

Chapter 1

Introduction

Functional software is one of the most essential parts of automotive electronics, in a sector of automotive development which will realise products with the most added value of a new car in a few years. A modern car has already built-in more than 80 electronic control units. This includes telematic systems like a navigation system, an mp3 player or a mobile phone, safety systems like an anti-lock brake system or an airbag, and driver assistance systems such as automatic headlight control.

With the increased complexity of electronic systems quality problems are also rising. In May 2004 Mercedes-Benz was forced to start the biggest product recall in its history. About 680,000 cars of the E- and SL-Class had to be checked, because in approximately 2 out of 1000 cases an error in the brake-by-wire system SBC might lead to a reduction of the brake force. Only one year later, DaimlerChrysler announced another even bigger recall of about 1,300,000 cars because of problems with the same system. Besides the loss of reputation and the inconvenience to the customer such a recall is quite expensive for the company.

Especially new safety systems, which are the focus of many manufacturers to achieve a technological leadership in safety due to its high rating by customers, have to deal with a development time frame that is shortened with every new car, and a rise in functionality at the same time. Furthermore these systems must never be activated if the situation is not critical, but always if it is, which is difficult to realise for control systems as the design of the plant model often leads to a simplification, e.g. through omitting non-linear influences.

In the field of software development several new methods were presented in the last years to improve the engineering process as well as the product itself. In 2001 Kent Beck published an article about a new style of development called test-driven development (TDD). With TDD the code is tested before it is written. Despite its test-centric approach, test-driven development is considered to be a design technique, not a testing technique. To date TDD has mainly been applied to the development of traditional software systems, but not to the design of control systems. This thesis introduces a new test-driven approach to develop embedded control systems with the focus on automotive safety.

1.1 Automotive Safety Systems

This section gives an historical overview of automotive safety systems and introduces the fundamental design principles of such systems.

1.1.1 History

It was in 1886 when Carl Benz invented and built the automobile [1]. He was the first of a number of today famous men like Gottlieb Daimler, Rudolf Diesel, Henry Ford, Wilhelm Maybach, Nikolaus Otto and Armand Peugeot who amongst others laid the way for the modern car [2, 3]. From the first cars at the end of the 19th century the total number of vehicles produced has grown to 38.6 million per year in 1980 and 66.5 millions in 2005 [4]. With this still increasing number of vehicles a lot of problems arose, which can be mainly divided into two categories: pollution of the environment and traffic safety [5].

In 2005, 43,443 persons were killed on the streets of the USA due to traffic accidents, compared to 51,091 in 1980. Taking into account that the vehicle miles travelled have increased by nearly 100% during this time (from 1,527 billion to 2,990 billion miles), the fatality rate per 100 million vehicle miles travelled has decreased from 3.35 to 1.45 [6]. In Germany the absolute number of persons killed decreased more significantly from 15,050 in 1980 to 5,361 in 2005 (that is 1.25 per 100 million vehicle miles travelled) [7]. The cause for this decrease is primarily the progress made through the so called safety systems [8].

Automotive engineers are trying to improve the safety of the automobile nearly as long as the automobile exists. In 1932 Chrysler was the first manufacturer who introduced power-assisted brakes to help to prevent accidents [9]. Those systems are called active safety systems. Their counterpart, the passive safety systems, try to

mitigate the damage of an accident when the accident is unavoidable, and in doing so their effects unfold not until the collision is happening. An example are the crumple zones, which came to market with the legendary "tailfin" Mercedes-Benz W111 in 1959 [9]. The era of electronic safety systems started with the anti-lock brake system (ABS), presented by Mercedes-Benz and Bosch in 1978 [10]. Since then, the number of new safety systems has grown steadily, as shown in Figure 1.1. Today, car makers as well as customers consider those kind of systems as some of the most important features of a new car [11, 12, 13, 14].

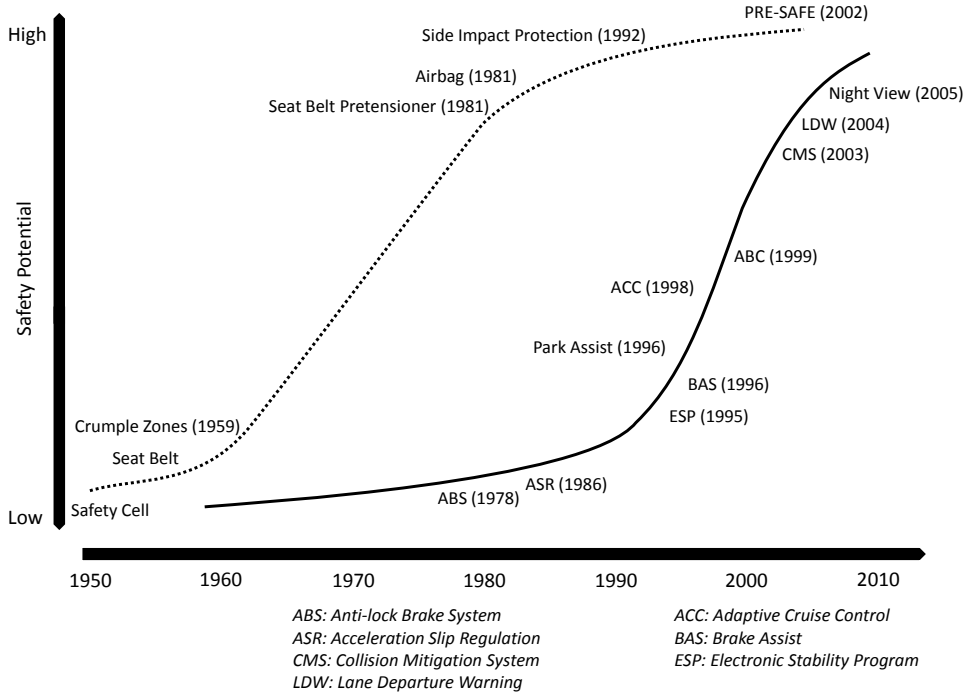


Figure 1.1: Milestones of automotive safety systems [15]

1.1.2 Design of Automotive Safety Systems

The design of a modern automotive safety systems consists basically of four parts: sensors, controller, actuators and the human-machine-interface (HMI). This subsection gives an overview of the role of each component.

Sensor: Depending on the system, one or more sensors measure the states of the vehicle and of the driver as well as observe the environment. The states of the vehicle usually include physical values such as the speed, e.g. measured by wheel impulse counters, the acceleration, e.g. measured by an accelerometer, or

the steering wheel angle; but also values such as the state of the turn indicator signal, the headlights, the wiper or the selected gear. The environment can be detected using ultrasonic, LIDAR (Light Detection and Ranging), RADAR (Radiowave Detection and Ranging), or vision-based sensors, see Figure 1.2. The sensors' data can be handled not only stand-alone, but also combined using sensor fusion algorithms. Finally, the state of the driver can be estimated with both groups of sensors, either directly by monitoring him/her with infrared or vision-based sensors, or implicitly by analysing the vehicle data, e.g. the steering wheel angle to detect fatigue [16].

Figure 1.2: Sensors for environment detection (from left to right): Ultrasonic sensor, infrared camera, and RADAR sensor (Photos: Robert Bosch GmbH)

Controller: The controller processes the data of the sensors to decide whether the system should be activated or deactivated, and to calculate the control signals for the actuators.

Actuator: The basic actuators of an active automotive safety system are the brakes, the engine (including the gearbox) and the steering, which are the same actuators the driver is using for controlling the vehicle. In contrast, passive automotive safety systems typically trigger a nonrecurring action, for example the ignition of a gas generator propellant to inflate an airbag. More advanced systems for combined active and passive safety also control the chassis (suspension and dampers)[1] and different servo motors, e.g. for the position of the headlights or for reversible seatbelt tensioners [17].

[1]The suspension and dampers not only influence the driving comfort and the driving dynamics, but can also be used to reduce the pitching of a vehicle during a braking manoeuvre, or to help the driver to perform an evasive steering manoeuvre.

Human-Machine-Interface (HMI): The communication between the driver and the system is done through the Human-Machine-Interface. It includes operating elements such as buttons or menu options for switching the system on or off as well as for adjusting parameters, and output elements to display the activation or error state, to warn the driver, or as the system's function itself, e.g. in the Night View system which displays the image of the camera directly in the cluster instrument. Furthermore also the throttle, the brake pedal and the steering wheel can be considered as input elements.

These components are usually distributed over different parts and control units of the car, which are connected by a network system like LIN, CAN or FlexRay [18]. Therefore these systems belong to the group of distributed embedded real time systems [15].

An example is the Adaptive Cruise Control System (ACC). It consists of a RADAR sensor mounted behind the grille to measure the distance and relative speed of the vehicle ahead, an Electronic Control Unit (ECU) which implements the controller and whose results are sent as brake commands to the ECU of the Electronic Stability Program (ESP) and as acceleration commands to the ECU of the engine (including also brake commands using the drag torque of the engine). The states of the vehicle, in particular the speed and the acceleration, are usually received from the ESP, which calculates them using the wheel impulse counters and accelerometers. Furthermore the ACC sends information to the cluster instrument to display the activation state, the desired cruise speed, the distance to the vehicle in front as well as its speed.

1.2 Control System Design

This section gives an historical overview of control system design and introduces the basic concept of a feedback loop. Furthermore, the basic design process for control systems is outlined.

1.2.1 History

When looking at the history of control, a number of events occurred which can all be seen as its starting point. For instance, in 1788 James Watt designed one of the first centrifugal governors as a part of the steam engine. A centrifugal governor controls the engine's speed by using the centrifugal force to regulate the amount of steam admitted. In other words, it uses the principle of proportional control to maintain a constant speed independent of the load or fuel supply conditions. About 50 years

later, Airy published one of the first theoretical works on control by mentioning instability of closed-loop systems and introducing differential equations for their analysis [19]. In 1868 Maxwell wrote his famous paper "On governors", in which he analysed different devices (including Watt's governor) by regarding them as a feedback control system [20]. He also showed that a system is stable if the roots of the characteristic equation have negative real parts. In 1892 Lyapunov discussed the stability of nonlinear differential equations [21].

Until the 1920's all mathematical analysis was carried out in the time domain. Frequency domain methods were first applied in control by Black in 1927, when he invented the negative feedback amplifier to reduce the distortion in repeater amplifiers[2] [22]. In the following years Nyquist with his stability criterion [23] and Bode with the magnitude and phase frequency plots [24] developed two major theoretical tools in classical control. In 1922 Minorsky first used a PID controller [25], whose tuning was (and still is) often governed by empirical techniques such as the formulae of John Ziegler and Nathaniel Nichols [26]. In 1948 Evans presented his root locus technique, which determines the closed-loop pole locations in the s-plane [27].

The era of modern control began in 1960, when Kalman and Bertram introduced the concept of internal system states and brought control theory back to the time domain [28]. With the invention of digital computers in the middle of the 20th century, sampled-data systems were developed by Ragazzini and Zadeh in 1952 [29] and Åström analysed digital control in 1970 [30].

Finally a new control theory, which takes the best features of classical and modern techniques, evolved in the 1970's and 1980's. In 1974 Rosenbrock and in 1977 MacFarlane and Postlethwaite extended the classical frequency domain techniques by the inverse Nyquist array [31] and the generalized Nyquist stability criteria [32]. Then Doyle and Stein showed in 1981 the importance of the singular value plots versus frequency [33], which allowed the use of many classical frequency-domain techniques for modern design.

1.2.2 The Basic Feedback Loop

In control theory, a dynamic system is often described as a set of mathematical formulae and a block diagram [34]. A block diagram representing the basic feedback loop is shown in Figure 1.3. It has three components: the plant consisting of the actuator and the process, the sensor and the controller. The plant represents the object to be controlled, whose input u is called the control variable, and whose output

[2]This was necessary to amplify the voice in long telephone lines, but not the noise.

y is called the plant variable modified by the disturbance z. The plant variable is measured by the sensor, which adds a measurement disturbance d. The controller generates the input of the process from the reference variable w and the sensor output r. In this thesis, the naming convention in Figure 1.3 is used.

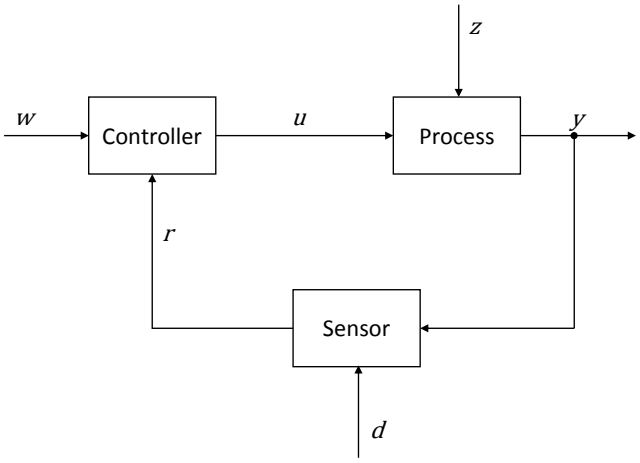


Figure 1.3: The basic feedback loop

It has to be mentioned that the variables are named with different letters in different literature. In figure 1.3 the variables are named after [35]. Other books like [36] or [37] uses for example r for the reference variable.

If the feedback is cut off between the sensor and the controller, it is called an open loop system, whereas the basic feedback loop is often called a closed loop system. Closed-loop control has the advantage that it can implicitly compensate for disturbances to the system and account for changes in the plant. For example, in a model of a vehicle, external disturbances would be caused by drag or the slope of the road. The aim of the controller design is then to make the acceleration of the vehicle follow a desired acceleration while external disturbances are rejected.

1.2.3 Process of Control System Design

The process of control system design is divided into two basic activities: analysis and synthesis. The major purpose of the analysis is the identification of the system in defining the equations that govern the plant's dynamics. The model is either built from known physical, usually differential, equations, e.g. for a mass-spring-damper system, or identified by using system identification [38]. With this, the system is stimulated with different input signals while measuring the output signals and trying to determine a mathematical relation between the two, e.g. using least-

squares optimisation [38]. In general, the obtained model can not entirely describe the real physical system as nominal parameters are never known with absolute precision. Furthermore for complex systems with non-linear behaviour or dead time it might be necessary to choose a simplified representation of the system to allow the creation of a controller at all [35].

The derivation of the controller, e.g. a feedback compensator, from the mathematical model is called synthesis. Different strategies are known, for example the above mentioned PID controller with its tuning techniques, direct pole-placement in calculating the feedback matrix with desired positions of the poles [35], or optimal control, which tries to minimize a performance index [39].

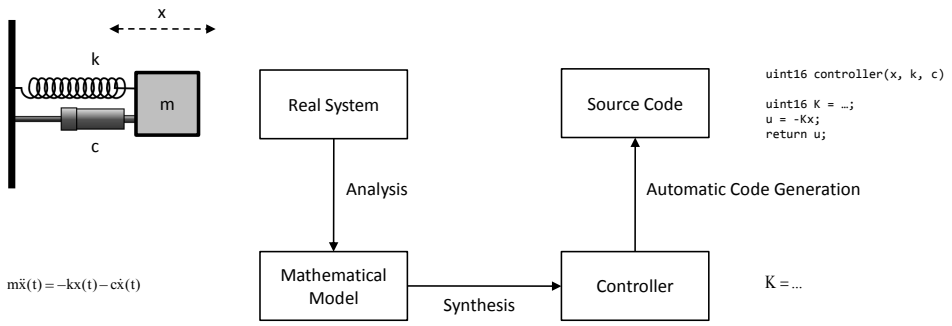


Figure 1.4: The process of control system design

A modern technique for control system design is model-based development [40, 41, 42]. In model-based development, the system is designed as a graphical model including all parts of the control loop [43, 44]. Such a model can be created with the tool Simulink [45], which furthermore allows the simulation of the model's states as well as the animation of the system, for example to show the trajectory of a controlled vehicle. Most important in model-based development is that the model represents the central artefact of the development process, which is systematically refined throughout the different process steps.

The controller represents the final product of this process and is usually implemented as software. This can be supported by the automatic generation of the software's source code from the graphical model, see Figure 1.4. The source code is then translated to binary code by a compiler and executed at the system's ECU. The ECU can be considered as a special-purpose computer system designed to perform a number of dedicated functions. It is embedded into a more complex device, e.g. a vehicle, which consists of further electronic and mechanical parts, e.g. an actuator as described in Section 1.1.2. Such systems are called *embedded control systems*.

As a consequence, the process of control system design can use two kinds of models. The analysis of the system is done with mathematical models, the realisation from a prototype for an initial idea up to the final product by graphical models.

1.3 Software Development Processes

Similar to the previous introductions to automotive safety systems and control system design, this section gives an historical overview in the first subsection. This overview describes different software development processes. Then a second subsection outlines the characteristics of the test-driven development process. Finally, the concept of design patterns, which describe common development practises in the form of a problem and its solution, is introduced.

1.3.1 History

A software development process describes the activities and sequences for the development of a software product. In the past, numerous models for such a process were developed. One of the oldest and best-known models is the Waterfall model, proposed by Winston W. Royce in an article published 1970 [46], which focusses on a distinct separation of the development phases to allow a lightweight way of planning and controlling the project, see Figure 1.5. Furthermore it offers a high efficiency for projects with stable requirements and limited complexity. However the waterfall model is only applicable to small projects, particularly because of its inflexibility against changes and the late recognition of defects. Ironically, Royce himself regarded this process as unsuitable for large projects and proposed several methods in the same article, which are comparable to an iterative approach.

In 1988 Barry Boehm introduced the Spiral model which divides the development process into different phases circling around the beginning of the project like a spiral, cf. Figure 1.6 [47]. Furthermore the coordinate system is cut into four quadrants with activities that define the objectives and constraints (north-western quadrant), evaluate the alternatives and identify as well as resolve risks (north-eastern quadrant), develop and verify the software (south-eastern quadrant) and finally plan the next cycle (south-western quadrant). The major advantage of such a model is the adaption of the project plan onto the progress of the project as well as onto the development of new technologies and the changing needs of the market. Disadvantageous is however the generalization of the actual development in one quadrant without specifying the relationship between the activities, e.g. implementation and test.

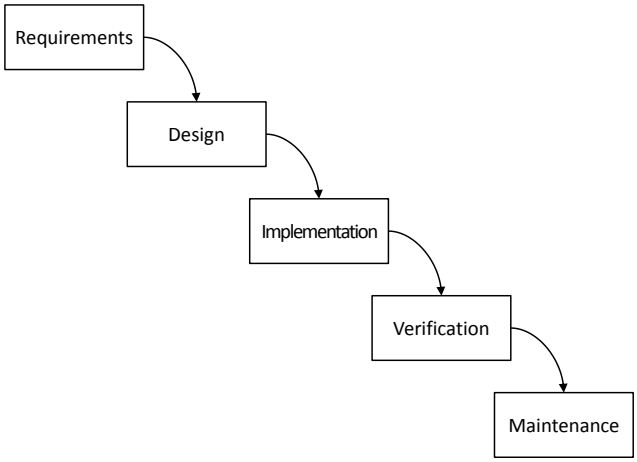


Figure 1.5: The Waterfall model

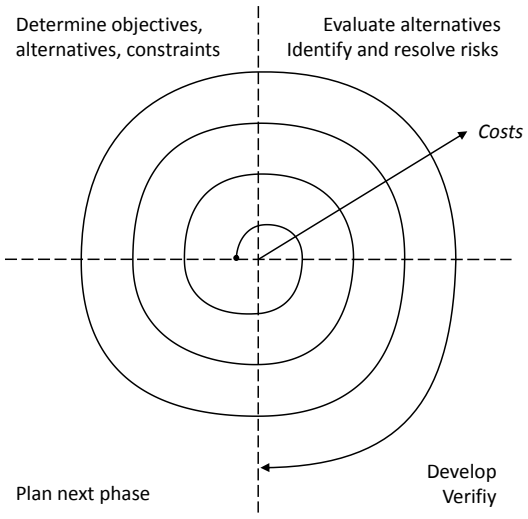


Figure 1.6: The Spiral model

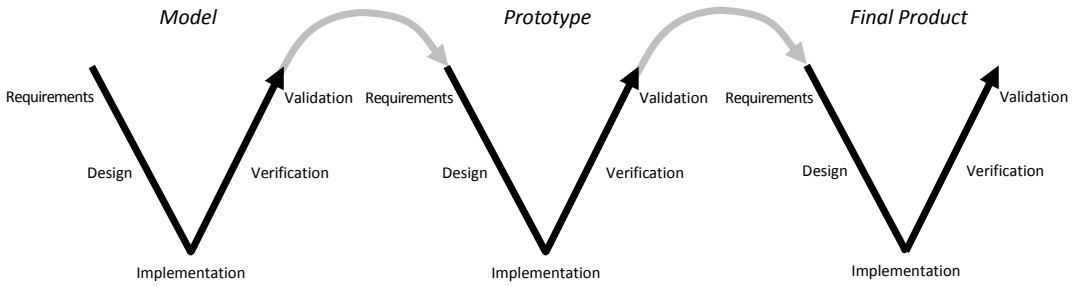


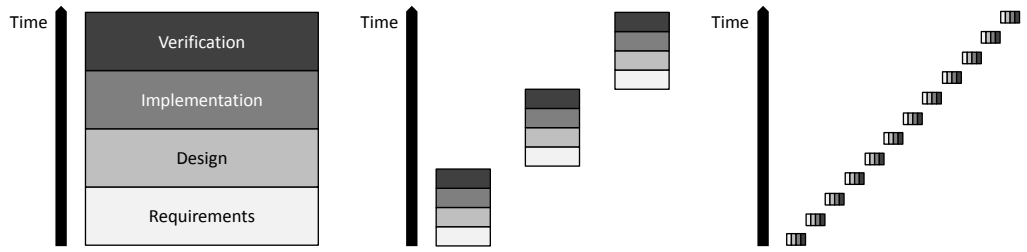
Figure 1.7: The Multiple V-model

The V-model separates the development process into phases, where the activities for design and implementation are positioned opposite to their respective activities for verification and validation [48]. Therefore the V-model belongs to the group of symmetric process models. The verification is done bottom-up, which means that, prior to the test of the complete system, all components have to be tested, beginning with the smallest entity, the software unit, which is usually defined as a function or method of the respective programming language. In consequence, the final product can be validated not until the final phase of the process which makes it complicated to react to failures in the early phases, primarily related to the design. The Multiple V-model is an attempt to solve this issue through dividing the process into multiple smaller V-models, which should allow the same flexibility as the Spiral model [49, 50]. An example realisation with three V-models is shown in Figure 1.7.

The software unit is also an important key element of Extreme Programming (XP), invented by Kent Beck and others in 1996 during the Chrysler C3 project[3] and first published in 1999 [51, 52, 53]. XP defines four basic programming practises - coding, testing, listening and designing - which form the major focus of the process. It differs from traditional methodologies primarily in placing a higher value on adaptability than on predictability, which is also expressed through its slogan "Embrace Change". The process is centred around short iterations (2-3 weeks) which focus on simple solutions to design for the needs of the current release. Figure 1.8 shows a comparison of the cycle times of the different development processes.

With extreme programming, testing plays an important role as tests are executed very often and a test is written for every new feature - the so called acceptance test - and for every software unit - the so called unit test. Typically the code is tested before it is written, which is referred to as test-first programming. In 2001 Beck presented

[3] C3 was the abbreviation for the Chrysler Comprehensive Compensation system whose aim was to realise the payroll processing for all employees of the Chrysler Corporation.



(a) Waterfall model (b) Multiple V-model (c) Extreme Programming

Figure 1.8: Comparing the cycle times of the different development models

this idea as a separate process, the test-driven development process [54]. Although the name includes the word test, test-driven development is considered to be a design technique, not a testing technique [55]. The reason is that a better design can be realised by the programmer, who just writes code to solve a problem - the failing test - and not to create a "masterpiece of software". This leads to a cleaner code, because defects in functions are found sooner as there is already a client for the function - the test [56, 57, 58]. Research studies showed that software created with TDD consists of more and smaller units which are less complex and highly tested [59, 60]. Furthermore when a function is changed later in the development process, the tests assure that no functionality is broken [61]. TDD can increase the productivity [62, 63], but might also affect it in a negative way due to the effort of more frequent testing [64, 65].

1.3.2 Test-Driven Development Process

In test-driven development there are two rules: New code is written only if an automated test has failed, and duplication is always removed. Therefore the following development cycle is defined [66]:

1. Write a test.
2. Run the test, which means
   - make it compile, and
   - run it to see that it fails.
3. Make it run, which means
   - make a change which solves the test.
4. Remove duplication.

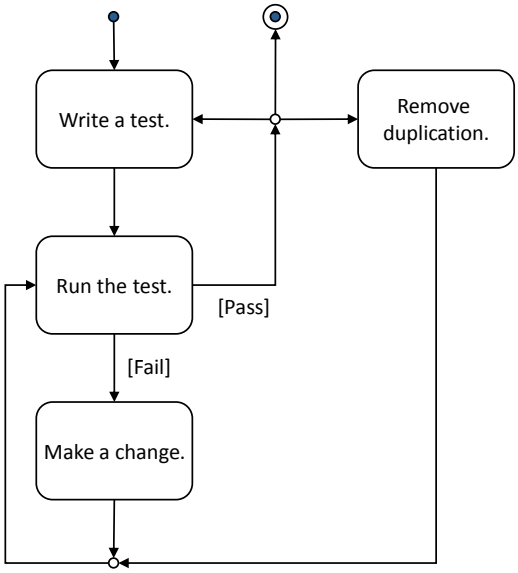


Figure 1.9: The development cycle of test-driven development

This cycle is represented as a state diagram in Figure 1.9.

In order to run automated tests, Beck also describes the development of a testing framework called xUnit for the programming language Python [66]. With xUnit, the programming language and the language to define a test are the same. Thus running an automated test is as simple as running the actual program. During test execution a progress bar shows how many tests have already been completed. At the end, a red bar means that one or more tests have failed, while a green bar shows that all tests have passed. For a failed test additional information is shown, for example the line number of an error. More recently a number of similar testing frameworks for most programming languages have been development: JUnit for Java, CppUnit for C++ or NUnit for the .NET platform [67]. Today the family of those frameworks is called xUnit [68].

1.3.3 Design Patterns

The origins of design patterns reach back to the architect Christopher Alexander who presented solutions for various architectural problems as "a pattern language" in 1977 [69]. A pattern in this sense is defined as "a three-part rule, which expresses a relation between a certain context, a problem and a solution" [70, p. 247]. It presents a solution for a well-defined problem which occurs under the circumstances of the described context. Thus patterns serve as a kind of design guideline or reference

description.

10 years later, Ward Cunningham and Kent Beck adapted the concept of patterns to the design of object-oriented software [71]. Then in 1994, the probably most influential work on design patterns for software was published by Erich Gamma et al. [72]. Their book, "Design Patterns - Elements of Reusable Object-Oriented Software", uses patterns to name, identify and extract the key aspects of common architectural and implementation structures, and with this, to create a reusable software design.

The characteristics of these software design patterns are similar to Alexander's structure: name, context, problem, and solution. The description of a pattern is not bound to a textual description. Instead the *Pattern Schemata* can also be represented by a model such as a block diagram. Common to all patterns is that they are created by analysing existing (successful) solutions and identifying (recurring) designs within these solutions. This process is called *Pattern Mining*. In addition, patterns can be used as a part of the system's documentation to describe a design decision by the three-part rule (context, problem, solution).

A number of patterns which are interrelated within a certain domain and grouped into a collection are denoted by a pattern language [69]. Nowadays, such languages exists for programming languages, software systems and software processes [73], and also for processes which are not bound to the software domain.[4] Several pattern languages have been introduced for the domain of control engineering and embedded systems [74]. These include patterns for time-triggered systems [75, 76], distributed systems [77, 78], controlling autonomous vehicles [79], real-time applications [80, 81], and generic real-time modelling [82]. The work on specific application domains has also been described with patterns, e.g. the design of an avionics control system [83]. In fact, even most textbooks on control theory include design patterns but do not use the typical pattern schemata. An example for this is the Feedback pattern (adapted from [74]):

> *Pattern Feedback*
>
> ***Context***
>
> *The system has a measurable output and a controllable input. A system model may or may not be available. A desired reference signal exists.*

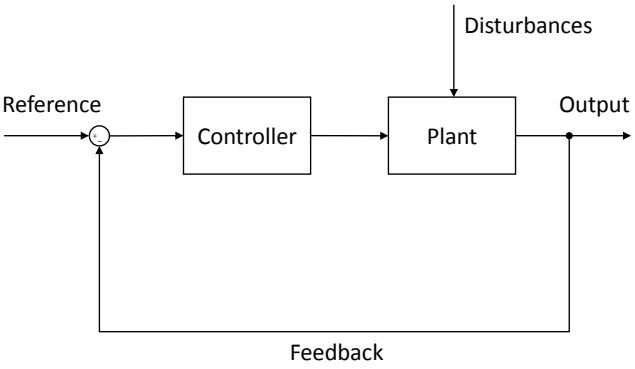
[4] A good example for the wide scope of the pattern concept is the book "Fearless Change: Patterns for Introducing New Ideas", M.L. Manns and L. Rising, Addison-Wesley, 2005; it describes techniques for introducing new ideas with the help of a pattern language.

Problem

The system output does not behave as desired. The response may be too slow, too oscillatory, unstable, or non-linear, for example.

Solution

The plant input is determined from the difference between the reference signal and the plant output by a feedback controller as shown by the following block diagram.



1.4 Thesis Contributions

Automotive safety systems are already one of the most important ingredients for a new car and will gain even more relevance in the next years. Their development is driven by the vision of accident-free driving, thus to reduce the number of collisions and fatalities, as well as by the need for features to differentiate a new car from its predecessor and from low-cost vehicles. With this increasing functionality the systems must handle more and more complex traffic situations while processing data from different sensors and controlling multiple actuators of the car. At the same time a system should never be activated if the situation is not critical, but virtually always if it is.[5] In consequence it is necessary to consider as many conditions as possible during the design of the system while preventing a development process which is too complex or does not fit into the development cycle of a new car [Issue 1].

Such safety systems consist of the components sensor, controller, actuator and human-machine interface. The design of the controller is typically based on mathematical design approaches in the frequency and/or in the time domain. At first a

[5]Typically, it is acceptable that a system is not activated at certain (borderline critical) situations, e.g. the airbag may not be inflated at low impact speeds.

model for the plant – the vehicle, the driver and its environment – is created using system identification techniques. The resulting plant is simplified to realise a model which is suitable for controller design. Such a simplification might lead to an insufficient behaviour of the final system in situations which are not considered by the model [Issue 2].

The next step is the synthesis of the controller, for example using the pole-placement method. In this case, the position of the poles is restricted by the stability criterion, which requires that all poles have negative real parts, and by the limits of the input value of the actuator. Moreover it is defined by a desired rise time, settling time, maximum overshoot, etc. These values must be determined by the developer [Issue 3].

In addition, model-based development allows the creation and evaluation of the controller not only with mathematical calculations, but also with graphical tools like Simulink. This allows for example the numerical simulation of non-linear characteristics of the plant, which are difficult to solve with mathematical analysis tools. Furthermore the model can be transformed into software by using automatic code generation. This software and the ECU, at which it is executed, represent the controller's implementation as an embedded control system. A crucial aspect of such embedded control systems is that they do not only implement the controller itself, but also realise components with more or less complex decision logic, e.g. for the system's activation. The influences of this decision logic are typically not part of the modelling of the control loop, but have to be considered during the development process [Issue 4].

The quality of an embedded control system is not only described by the performance of the controller, but also through other characteristics such as the stability and the robustness over the lifespan, and the number of defects, which are in most cases not zero. Therefore testing becomes more and more important for the developer. In contrast to traditional testing which is done after the design of the software, test-driven development introduces the concept of testing first. No code is written before a test has failed. To date, test-driven development has only been applied to common programming languages such as Java and to common software systems like payroll applications, but not to graphical signal driven languages like Simulink and to the design of control systems implemented with it [Issue 5].

In addition, it has to be considered that the controller is implemented as software, i.e. the results of the mathematical design approaches are transformed into a different representation.

Therefore the goal of this thesis is to use testing as a design method for control

systems to address the issues defined above. A new design process will be presented which is based on test-driven development and considers the characteristics of control system design. This process is then applied to an automotive collision avoidance system which helps the driver to prevent collisions through an acoustical warning before an imminent collision and through increasing the brake force up to the maximum if necessary. Finally this system is evaluated based on three criteria. First, the system is reviewed for aspects of control system design, e.g. stability or robustness. Second, measurement data from a pre-series vehicle are analysed for different use cases and situations. Third, the new methods are compared to alternative design methods and to patterns for test-driven development.

Some of the work reported in this thesis has been published in the following paper:

[84] **Test-Driven Development of a PID Controller**, T. Dohmke and H. Gollee, IEEE Software, 2007

The paper describes the novel application of test-driven development to control system design using a simplified vehicle system as an example. Furthermore an automated testing framework is implemented with the graphical programming language Simulink.

Additionally, more detailed information about the Collision Prevention System can be found in the following patents:

[85] **Method for Identifying Critical Collision Situations from the Rear**, B. Danner, T. Dohmke, J. Hillenbrand, V. Schmid, A. Spieker, 2006

[86] **Method for Identifying Rear End Collision-Critical Situations in Lines of Traffic**, B. Danner, T. Dohmke, J. Hillenbrand, V. Schmid, A. Spieker, 2006

[87] **Method and Vehicle Assistance System for Preventing Collisions or Reducing the Severity of a Vehicle Collision**, B. Danner, T. Dohmke, J. Hillenbrand, V. Schmid, A. Spieker, 2006

[88] **Method for Operating a Collision Avoidance System of a Vehicle and Associated Collision Avoidance System**, B. Danner, T. Dohmke, J. Hillenbrand, V. Schmid, A. Spieker, 2006

[89] **Method for Operating a Braking Assistance System in a Vehicle**, B. Danner, T. Dohmke, J. Hillenbrand, V. Schmid, A. Spieker, 2006

[90] **Method for Avoiding a Collision or for Reducing the Consequences of a Collision and Device for Carrying Out Said Method**, B. Danner, T. Dohmke, J. Hillenbrand, V. Schmid, A. Spieker, 2006

[91] **Method for Adapting Intervention Parameters of an Assistance System of a Vehicle**, B. Danner, T. Dohmke, J. Hillenbrand, V. Schmid, A. Spieker, 2006

[92] **Method for Operating a Collision Avoidance System or Collision Sequence Reducing System of a Vehicle, and Collision Avoidance System or Collision Sequence Reducing System**, B. Danner, T. Dohmke, J. Hillenbrand, V. Schmid, A. Spieker, 2006

[93] **Method for Operating a System for Avoiding Collisions or for Reducing the Consequences of a Collision for a Vehicle and a Corresponding System for Avoiding Collisions or for Reducing the Consequences of a Collision**, B. Danner, T. Dohmke, J. Hillenbrand, V. Schmid, A. Spieker, 2006

[94] **Method for Operating an Assist System for a Vehicle and Park Assist System**, T. Dohmke, V. Schmid, 2006

The patents can be divided into the following groups: (i) the assessment of driving situations to identify potential collisions [85, 86, 90], (ii) the assessment of the driver's behaviour to allow the activation of a collision prevention system [89], (iii) the definition of threshold values and the adaption of parameters for a collision prevention system [88, 91], and (iv) the realisation of a collision prevention system which outputs a warning to the driver and carries out an automatic braking manoeuvre [87, 92, 93, 94].

1.5 Thesis Outline

This chapter outlined the motivation for the development of a test-driven design process for embedded control systems and formulated the goals of the thesis.

Chapter 2 introduces the methods and tools for this process. First, the development of a novel automated testing framework for the graphical programming language Simulink is explained along with the structure and patterns of the xUnit family. Then the new development process is defined based on the multiple V-model and test-driven development. Furthermore, a layer concept of the automobile is presented to show how TDD can be applied during different development phases.

Chapter 3 describes the application of the test-driven design process to an automotive safety system using a collision prevention system as an example. First the complete system is specified using the different layers from the second chapter. Then

the test-driven development process is explained for the activation of the system based on three components. The realisation of a fourth component, which implements the controller of the system, is done by a conventional approach as well as a test-driven approach. These two approaches are described in Chapter 4. Furthermore, the system is analysed using control criteria such as stability and performance. Chapter 5 then presents the results of the experimental evaluation of the developed system based on measurements from test-drives with prototype vehicles and statistics from trials with a driving simulator.

Chapter 6 discusses the new test-driven development process in terms of design patterns for software development, testing and test-driven development. Two specific patterns are derived for the test-driven development of embedded control systems. Using these patterns, the process is evaluated in comparison to traditional development processes. In addition, the application to different kinds of embedded components is explained.

Finally, in Chapter 7 the thesis addresses the conclusions which can be drawn from the work presented. An outlook to future work on test-driven development for embedded control systems is given.

Chapter 2

Methods and Tools

The purpose of this chapter is to describe the methods and tools of the test-driven design process for embedded control systems. First, model-based development and its notations are described, including an overview of the modelling tool Simulink. Furthermore the xUnit family is introduced with its structure and patterns. These patterns are used in the development of a novel automated testing framework for Simulink. Simulink and the new framework are the primary tools for the new development process, which is based on the multiple V-model and test-driven development (TDD). The last part of the chapter presents a layer concept of the automobile to show how TDD can be applied during different development phases.

2.1 Model-Based Development

2.1.1 Definition

The origins of model-based development are reaching back to the so-called Computer Aided Software Engineering (CASE), which describes the use of software-based tools for the development and maintenance of software itself [95, 96]. CASE can significantly increase the productivity of the development process and the quality of the final product [97, 98]. Therefore CASE provides tools for automatic code generation, revision control and change management. The system is usually represented as a model which describes the architecture as well as the function. These models can be created with three paradigms:

Functional Notation / Data-Flow Models

The functional notation is based on mathematical functions, whose purpose is to transform input data to output data. Such functions have no side-effects and no states. Moreover a value can be assigned to a variable, but not changed. Therefore the execution order of a statement is negligible, for example the following statement

```
y = f(x) + f(x)
```

is identical to

```
y = 2 * f(x)
```

because the function `f` always returns the same output value for the same input value of `x`. Recursion is used instead of iteration as variables are invariable and functions have no state. In consequence, an algorithm is described through a sequence of functions, often including differential equations.

Usually data-flow models, e.g. block diagrams, are used to describe systems by the functional notation [99, 100]. The models represent the basic elements in terms of vectored lines (signals) and blocks (functions), either as built-in functions for basic algorithms like addition or as user-defined functions composed from a sequence of built-in functions, cf. Figure 2.1(b). Typically there are also blocks which have a state, e.g. blocks to integrate or differentiate a signal. This can be understood either as initial value of a function, e.g. $f(0) = f_0$, or as the realisation of functions which depend on the time, e.g. $f(t_i) = x + f(t_{i-1})$.

Imperative Notation / Control-Flow Models

The imperative notation is based on the operation of computers. It describes the stepwise execution of a process, including statements, conditions and loops. In contrast to the functional notation, functions can have side-effects and states, e.g. due to the assignment of a global variable. Thereby the aggregation of the states of all functions is considered as the program's state.

Besides the well-know programming languages like C or Pascal this notation can be reflected through control-flow models. The three major forms of such models are Nassi-Shneidermann diagrams [101], flowcharts [102, 103] and statecharts [104]. Both Nassi-Shneidermann diagrams and flowcharts are state-less. While the first use a kind of tabular scheme with basic cells for process steps, conditional decisions and loops, the latter are described with a diagram of geometrical forms and vectored lines. With this, the geometrical forms specify the same actions as the cells, e.g. a rhombus

represents a conditional decision. In contrast, statecharts are build with transitions and, as the name implies, states. It is possible to compose states to superstates and concurrent states, i.e. the model can have more than one active state at the same time. A transition can be annotated with a condition which activates the transition and therefore allows the creation of branches, and an action which is executed during the transition.

Figure 2.1 shows a comparison of a data-flow model to a flowchart and a statechart model for the activation subsystem of an automotive safety system as an example.

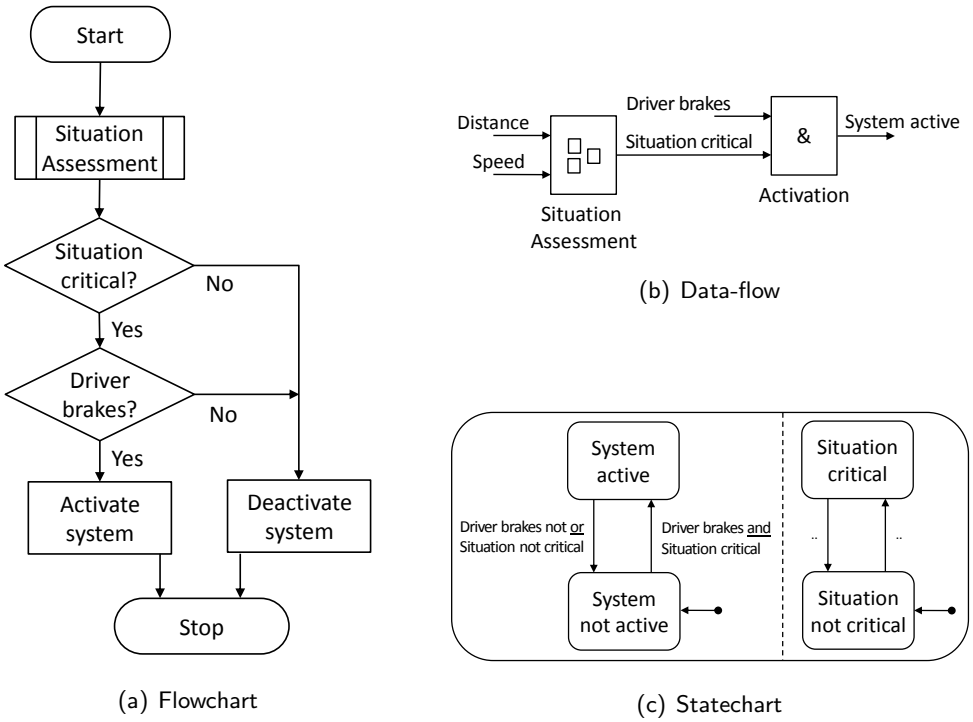


Figure 2.1: Different models for the same activation subsystem of an automotive safety system

Object-Oriented Notation and Models

The object-oriented paradigm classifies data based on its attributes and applicable operations [105, 106]. The notation is represented through so-called classes, which encapsulate similar attributes and related operations (methods). An instance of a class is named an object. As a real world example, the construction plan of a car can be considered as a class; an object is then a car built from this plan. Furthermore classes can be derived from other classes inheriting both attributes and methods (generalization). In doing so, so-called polymorphy allows to use the same interface

for accessing different objects of the same base class, but considering the respective implementation. Continuing the example of the car, a second construction plan for a sport variant would inherit all features of the first construction plan, but use a different engine. Then pressing the throttle will accelerate both kinds of cars, but the value of the acceleration will depend on the engine. Moreover it is possible to define a general relationship between classes (association) as well as part-whole relationship (aggregation, composition).

In the last years the Unified Modelling Language (UML) has become a standard to model object-oriented systems [107, 108]. It offers two basic types of diagrams: (i) structure diagrams, e.g. class diagrams, component diagrams or deployment diagrams, and (ii) behaviour diagrams, e.g. activity diagrams, sequence diagrams or state diagrams. The latter use the same notation as the statecharts of the imperative paradigm. In consequence, all object-oriented programming languages, such as C++, Java or Smalltalk, include also control-flow or procedural parts.

•

With CASE, notations like data-flow or control-flow models were mainly used to describe a prototype of the system, while the implementation was done with common programming languages [109]. In contrast, model-based development focusses on the model as the central and primary artefact throughout the whole development process [44, 43, 110].[1] The architecture, the components and the system itself are described using different models, which are partly or fully abstracted from the implementation platform. Usually it is possible to check for the consistency of these models, to execute a simulation and to generate automatically source code from them. The source code can be translated to binary code by a compiler and then executed at the system's implementation platform. An overview of tools which realise this model-based process not only for prototyping, but also for production purposes is given in Table 2.1 [114].

To date, the tools Simulink and Stateflow from the company The Mathworks are commonly used in the automotive industry and can be considered as state-of-the-art for the design and development of embedded control systems for automotive safety systems [115, 116]. Therefore these tools were the basis of this work to introduce a test-driven development process for model-based development and to develop the example vehicle system.

[1] Furthermore several sub-techniques emerged, e.g. model-based testing [111, 112] and model-based verification [113].

Vendor	Name	Notations
ETAS	ASCET	block diagrams, statecharts
The MathWorks	Simulink & Stateflow	block diagrams, statecharts
Rational	Rose RealTime	UML
Telelogic[a]	Rhapsody	UML
Telelogic	Tau	UML

Table 2.1: Modelling tools for production purposes

[a]The former vendor I-Logix was acquired by Telelogic in March 2006.

2.1.2 Overview of Simulink

The tool Simulink, whose first version came to market in 1990 [117], is based on the numerical computing environment MATLAB, which offers a mature set of functions to solve mathematical problems using vectors and matrices [118, 45]. Simulink implements a graphical programming language with the basic elements *signal* and *block*, cf. Figure 2.2. A block transforms input values into output values using a mathematical function and is drawn as a rectangle with incoming and outcoming ports. Furthermore, each block can be assigned a name using a label. A signal represents a scalar value and is displayed as a vectored line, optionally also with a label. It is always connected with a minimum of two blocks, one as the source of the signal, one or more as the sinks. In consequence, Simulink models can be considered as data-flow diagrams.

The type system of Simulink consists of all known basic data types, i.e. Boolean, fixed point and floating point values. The data type is assigned to a signal at the output port of its block. For example, the result of an addition of two input values of the type double is again a value of the type double, while the comparison of the same values with a conditional operator creates a Boolean value. Moreover it is possible to combine different signals to a vector or to a bus. The difference between a vector and a bus is that only a bus can contain signals with different data types. In addition, the bus is displayed as a triple line and the signals in it retain their labels.

Two further elements of Simulink are the *subsystem* and the *model*. A subsystem encapsulate blocks and signals into a (logical) group, where two special types of blocks represent the interface of the subsystem: inports for the input interface, and outports for the output interface. Both can connect to all three kinds of signals: scalar values,

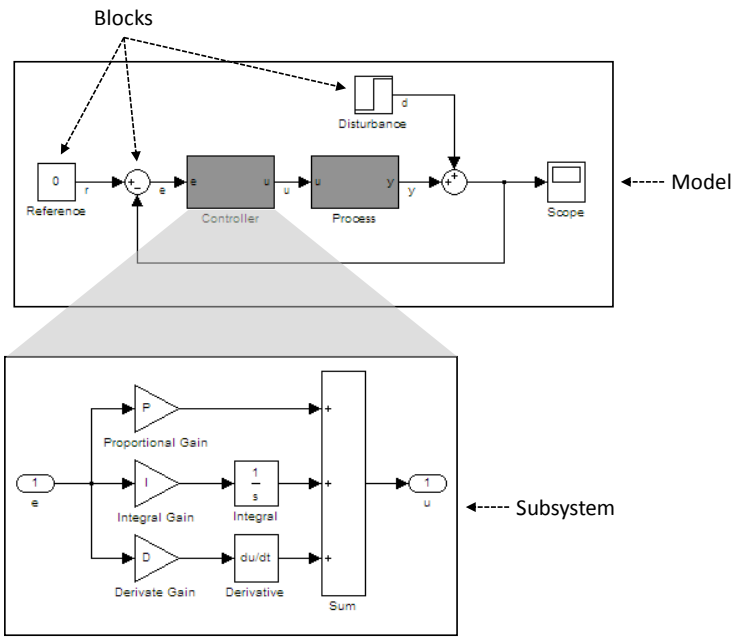


Figure 2.2: Basic Elements of Simulink

vectors and buses. A subsystem can not only be labelled with a name, but also marked with a masked type, i.e. an attribute. While the name of a block has to be unique within the subsystem's level, the mask type can be re-used and therefore allows a classification of similar blocks. Furthermore, a subsystem can be masked with an icon and a graphical user interface, so that it behaves like a block.

The model is the top-level subsystem and is represented by a single file. Moreover, models can be marked as libraries. Such a library is the Simulink library. It includes the basic blocks for the creation, conversion and routing of signals, for mathematical and logical operations or for the output of signals as plots as well as additional blocks in the form of subsystems, e.g. blocks for various forms of controllers or to represent dynamic models in a transfer function or state space form. Therefore the model's file can contain the top-level system with the implementation of all subsystems, but also references to subsystems which are part of a library or another model (model referencing).

In addition, elements to model control-flow structures are part of the Simulink library. These provide the developer with special blocks for *for*, *if-else*, *switch* or *while* constructs, and with tools to generate statechart or flowcharts diagrams using the Stateflow block. Hence it is possible to design and implement complex systems using data-flow as well as control-flow models with Simulink.

Finally, Simulink can be extended by add-on products, so-called toolboxes. For the automatic generation of source code from a Simulink model several toolboxes are available, the most important are Real-Time Workshop from The MathWorks and TargetLink from dSpace. With both it is possible to generate stand-alone ANSI C code directly from a Simulink model. To date, the high quality of the code generators as well as the efficiency and flexibility of the generated code led to a wide adoption of the model-based process for the production of embedded systems [119, 120].

2.2 Model-Based Testing Framework

With the concept of extreme programming and test-driven development and its technique of automated testing a number of testing frameworks for most programming languages have been developed in recent years. The concept and architecture of all these frameworks are reaching back to SUnit, a unit test framework for the Smalltalk language published by Kent Beck in 1994 [121]. Due to its simplicity, a consequence, amongst other things, of the philosophy that tests are written in the same language as the actual program, SUnit was ported to Java in 1998, creating JUnit [122]. Today a whole family of these frameworks exists, called the xUnit family, e.g. CppUnit for C++, PyUnit for Python or NUnit for the .NET framework [67]. Furthermore a framework for the MATLAB programming language has been realised during this thesis project, called mlUnit [123].

For Simulink several testing frameworks were developed in the last years. In this section, these frameworks are first evaluated for the compliance with the test-driven development cycle, cf. Chapter 1.3.2. Then the architecture of the xUnit family is described. As a result, the concept and realisation of a new testing framework, slUnit, is explained by analysing the six main patterns of xUnit and deriving a realisation for Simulink. Finally further patterns discuss different practises for the usage of slUnit.

2.2.1 Testing Frameworks for Simulink

The different testing frameworks for Simulink can be divided into two groups:

Manual Test Design: The frameworks of this group allow the manual design of tests using MATLAB variables or scripts, test vectors, the classification-tree method (see 2.3.2) or separate tabular files created for example with Microsoft Excel. The test execution is divided into two steps: First a test bed is automatically generated from the model, then a single test, a set of tests or all tests can be executed. Furthermore different techniques are provided for the analysis of the

results, for example regression with other tests or comparison with expected output values. All steps are supported by a graphical user interface.

Members of this group are dSpace MTest [124, 125] and MathWorks SystemTest [126].

Automatic Test Design: With these frameworks, the tests are created automatically from the test objective. Therefore the developer does not define test cases, but designs patterns for the test generator, which automatically generates test scripts from these patterns and the Simulink model. The test scripts link the test patterns with the Simulink model, i.e. they describe the test vectors for the stimuli of different input signals. The first step of the test execution is again the automatic creation of the test bed. Then the tests are run for a number of iterations. After each iteration, different criteria, for example statement or branch coverage, are evaluated and subsequent tests modified in such a way that previously uncovered elements are included. The frameworks also provide a graphical user interface to support the testing process.

Members of this group are Reactis Tester for Simulink [127, 128] and T-VEC Tester for Simulink [129, 130].

It is not possible to implement test-driven development with the group of frameworks for automatic test design. In this case, tests would be created by analysing a model which does not yet exist. Moreover the test design from patterns destroys the simplicity of the development cycle as patterns are more general than a single test and can not be used for an incremental development cycle.

The group of frameworks for manual test design do in general allow a test-driven development process. In contrast to xUnit, the tests are not specified in the same language as the test objective, but with an external tool such as MATLAB, CTE or Microsoft Excel. This is only feasible if the tests are simple and input data does not depend on output values. To use a plant model inside a test case for the testing of a closed-loop system, the plant model has to become part of the test objective (as seen by the testing framework). Then the test stimuli are defined as input values of this system, modifying (initial) parameters of the plant model which stimulates the original test objective (as seen by the developer). Such a concept requires that the model and the test framework are implemented in the same language.

All four frameworks, i.e. MTest, SystemTest, Reactis Tester and T-VEC, are closed source and commercial products. Therefore a new testing framework was developed within the scope of the PhD project with three main objectives:

1. Based on the xUnit family.
2. Using Simulink for the design and execution of tests as well as the development of the test objective.
3. Available as open source although it is based on the commercial tool Simulink.

The new testing framework is called slUnit. The remainder of this section will describe the architecture of xUnit, analyse its patterns and derive a realisation that considers the specific characteristics of Simulink.

2.2.2 The xUnit Family

The basic design of the xUnit family is based on an object-oriented architecture with a set of key classes, which are shown in Figure 2.3 [67]. In its centre stands the class *Test* which allows, in combination with inheritance, to easily create a test, to manage hierarchies of tests and to prevent duplication (*Test Case*, *Test Suite*, and *Test Fixture*). A test is executed using an instance of the class *Test Runner* and the results are collected in an instance of *Test Result*.

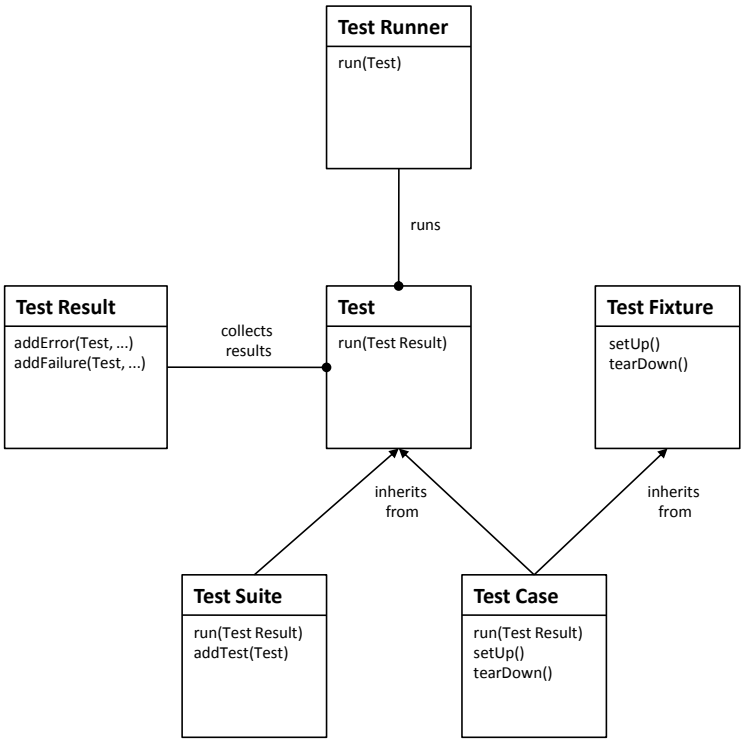


Figure 2.3: Core classes of the xUnit architecture [67]

The major question for the design of a testing framework for a model-based tool like Simulink is how to translate the class-based hierarchy of xUnit into the graphical notation, which uses models, blocks and signals.[2] Besides having the same architecture, all testing frameworks allow certain patterns for testing ([66], chapter "xUnit Patterns", pp. 157ff). These are:

Assertion: How to check that the tests worked correctly?

Test Method: How to represent a single test case?

All Tests: How to run all tests together?

Fixture: How to create common objects needed by several tests?

External Fixture: How to release external resources in the fixture?

Exception Test: How to test for expected exceptions?

In the following sections the concept of the new unit testing framework, called slUnit [132], is described. As the xUnit architecture is not one-to-one portable to the elements of Simulink, the focus will be on the above patterns and the testing techniques of xUnit [68]. Another fundamental difference is the time-dependent behaviour of the model-based approach, which needs to be taken into account by the new framework. With xUnit, the developer is able to test the return values of a function call with static arguments while the signals and states of a Simulink system always depend on the point in time at which the simulation step is calculated.

2.2.3 Assertions

The *Assertion* is the basic element for automatic testing as it is evaluating whether a result or output value meets the expectation or not and thereby checks, whether a test has passed or failed.

Generic Concept (xUnit)

Most xUnit frameworks provide a basic set of built-in assertion methods, divided into the following groups:

Single Outcome Assertions always behave in the same way, usually to let a test fail which is not finished yet, or within the `try`-branch of a `try/catch`-mechanism

[2] One adaptation of the xUnit architecture to message sequence charts as a model-based approach has been developed by Yuefeng Zhang [131].

to detect a missing expected exception (see section 2.2.6). The only parameter is an error message – common to all assertion methods –, which is collected in the test result and displayed by the test runner. Example:

```
fail('Unfinished test.');
```

Stated Outcome Assertions accept a parameter which states whether the assertion will fail or not. The most widely used variant handles a Boolean input, e.g.:

```
assert(y > foo(x));
```

`y` is the expected value, `foo(x)` a function which is the test objective. Here, the assertion fails if the output of the test objective is smaller than or equal to `y`, i.e. if the Boolean expression is false.

Equality Assertions compare its two parameters, the expected value and the actual value, and fail if the outcome of the comparison is not true. The relational operator is usually specified through the name of the assertion, e.g.

```
assert_equals(y, foo(x));
```

checks whether the values of `y` and `foo(x)` are equal. The idea behind this is to display a more descriptive error message than with single or stated outcome assertions. Assuming that the value `y` was set to 2 and the function `foo(x)` returns 3, the error message will be automatically set to

```
Expected <2>, but was <3>.
```

Furthermore an optional third parameter allows to specify a tolerance, e.g.:

```
assert_equals(1.41, sqrt(2), 0.01);
```

will pass if the result of sqrt is greater than or equal to 1.40 and smaller than or equal to 1.42.

More complex assertions can be built using methods or functions of the corresponding programming language. The aim of these user-defined assertions is to prevent duplication in reusing code and to avoid conditional test logic.

If an assertion fails the test is stopped at the line of the assertion. However the execution of the next tests is continued as tests are independent of each other in

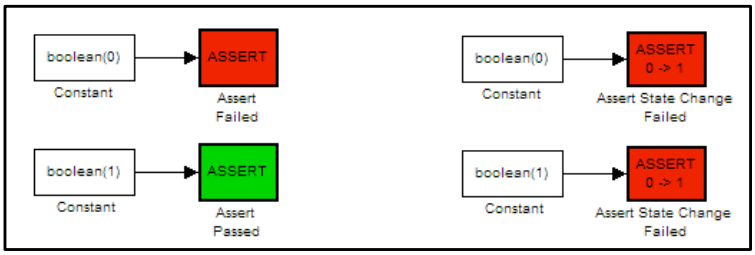
xUnit. After all assertions associated with a specified test have passed, the test passes which is typically indicated by a green bar. If one assertion fails, the test fails which is usually indicated by a red bar. With the red bar a list of failures including the expected result, the line of the failure or the optional message is shown, so that the developer is able to find the position and the reason for the problem.

Realisation (slUnit)

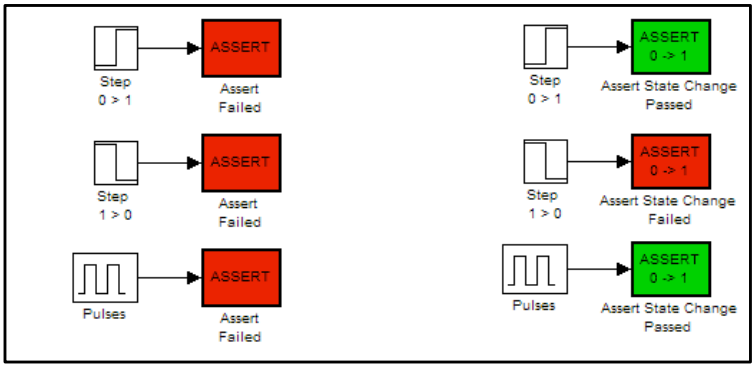
With slUnit, the Assertion is implemented as the Assert block. It has one Boolean input similar to the method `assert`, describing whether the assertion should pass – the Boolean input is true – or fail – the Boolean input is false. The state of the assertion is displayed as the background colour of the block – green for success, red for failure. This allows on the one hand the evaluation of the test without the need for an additional graphical user interface or an output on the command line, on the other hand it is easy to locate the failed assertions. In contrast to the xUnit assertion, the Assert block has to consider the time-dependent behaviour of its input signal. The Assert block is failing if the input signal becomes false and will not pass for the remainder of the test, but it will not stop the test. Therefore the assertions are independent from each other, i.e. more than one assertion can fail during a test in slUnit. This is typical for Simulink-based tests as they mostly analyse the behaviour of different signals. If consecutively executed Assertions are required they can be built through additional Simulink components such as enabled Subsystems or state machines.

An extension of the Assert block is the Assert State Change block, which evaluates whether the input changes from a value x to a value y during the simulation. It fails if this change does not happen. The difference to the Assert block is that the Assert State Change block can not fail until the end of the simulation, because until then it is not proved that the state change will not happen. Figure 2.4 shows the difference of both blocks for a number of static and dynamic inputs. In general, the Assert block only gets passed if it is stimulated by a constant Boolean input with the value *true*. In contrast, the requirements of the Assert State Change block are never fulfilled by a constant input, but by dynamic inputs which describe the specified number of state changes.

A similar block to the method `assert_equals` is not provided as the comparison of two time-dependent signals is in most cases much more complicated than $a = b$. Instead, the Propagate block provides the propagation of the background colour of the Assert blocks to the belonging subsystem. This allows the creation of user-defined



(a) Static Inputs



(b) Dynamic Inputs

Figure 2.4: Behaviour of the Assert and Assert State Change block for different inputs

assertions based on the two basic assertion blocks and all common Simulink blocks. Figure 2.5 shows an example assertion which checks the boundaries of its input signal. Finally it is also possible to automatically open Scope blocks or Figure windows for debugging purposes.

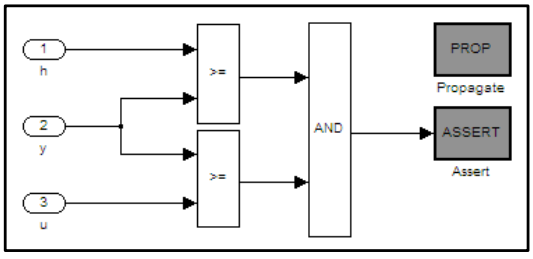


Figure 2.5: Example for a user-defined assertion

2.2.4 Creating and Executing Tests

The pattern *test method* which describes how to represent a single test case, and the pattern *all tests* which explains how to run all tests together, will be discussed

together in the next section. Both build up the basic architecture of xUnit as they specify the way of defining a single test case, a set of test cases, the relationship between them and the execution of tests.

Generic Concept (xUnit)

A single test case is represented as a method whose name begins with "test", called *Test Method*. The designated name was chosen to allow an instance of the class Test Runner to find of these methods. Depending on the programming language some frameworks now use attributes or annotations for this purpose, e.g. with Java 5 and JUnit 4 a test is defined by

```
@Test public void emptyStack() {
   stack = new Stack<String>();
   assertTrue(stack.isEmpty());
}
```

with the annotation `@Test` marking the method as a Test Method [133].

A Test Method can contain various numbers of assertions which are checked consecutively. It fails as soon as an assertion fails. The subsequent assertions are not executed as they might depend on each other, e.g.

```
assert_not_equals(0, foo(x));
assert(0 < (1 / foo(x)));
```

Moreover the method can use all structural features of the respective programming language, e.g. if- and while-clauses, function calls or subroutines, and therefore the assertions might be also part of these structural features.

Test Methods are organized in classes as most xUnit frameworks are implemented in an object-oriented programming language. Typically those Test Methods which are sharing a single fixture (see section 2.2.5) are methods of the same class. The basic class is called *Test Case* and consists of at least one test method. Instances of Test Case, therefore called *Test Case Objects*, are aggregated in the *Test Suite Object*. A Test Suite Object can be also part of another Test Suite Object. Comparing this organisational structure to a tree, Test Suite Objects are the nodes while Test Case Objects are the leaves, see Figure 2.6. Thus for each Test Method one Test Case Object is created. Furthermore it is also possible to have two instances of the same test case class with the same Test Method, e.g. to test the same Test Objective with different input values specified through a parameter of the respective class instance, see test 4 and 5 in Figure 2.6.

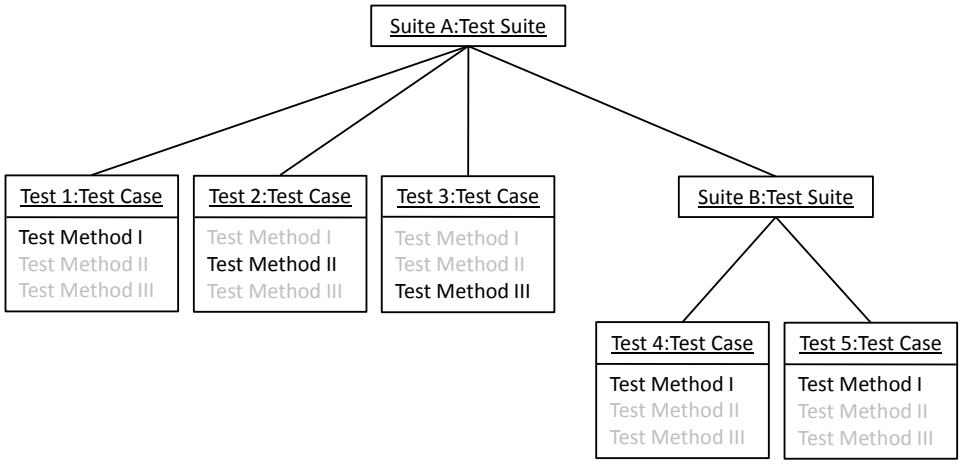


Figure 2.6: Tree of Test Suite and Test Case Objects

The members of the xUnit family provide several forms of command-line or graphical applications to run one or more test cases, which are all realised as an inherited class from Test Runner. These classes are based upon a standard object-oriented interface, which allows the execution of a set of tests in a unified manner regardless to the kind of test runner. The tests to be executed can be defined using the following techniques:

Enumeration: Each Test Method is added manually using the constructor of the test case class, which is called with the name of the test method as a parameter. With this, also a single test method can be executed.

Discovery: The test methods are discovered automatically by the framework. To enable discovery a naming convention or annotation is used depending on the features of the programming language as described above.

Selection: In addition to the discovery, the test runner is invoked together with special test selection criteria specifying a subset of tests. The criteria are usually based on additional attributes or annotations to every test case class or test method.

The test process is documented during the execution through some form of real-time progress indicator, including a count of executed tests, of failures and errors and a coloured bar, which turns red as soon as a failure or error has occurred.

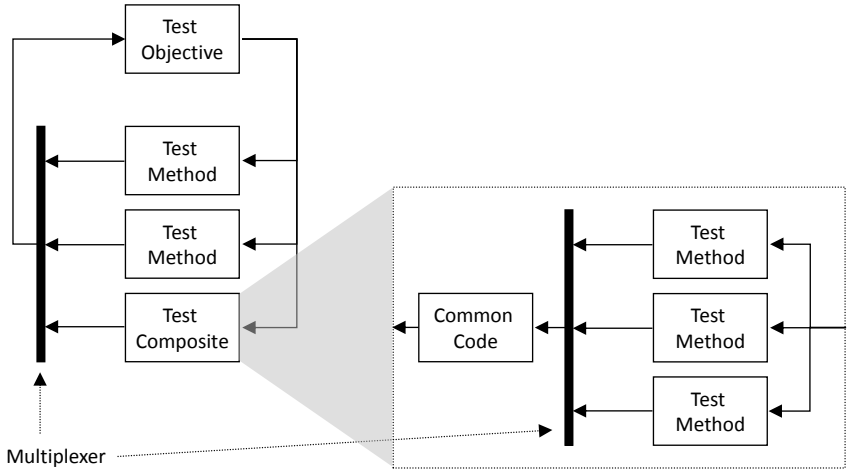


Figure 2.7: Architecture of a slUnit model

Realisation (slUnit)

The design of slUnit translates the class-based hierarchy of xUnit into the structure of Simulink in representing the test objective as well as the test cases as subsystems. The test methods have to be marked with a certain mask type to distinguish them from other subsystems used either for structural meanings or as the test objective. The mask type is an attribute to classify Simulink subsystems and therefore comparable to the attributes or annotations used in other xUnit frameworks. Furthermore a subsystem can also represent a test case class or a test suite. As there is actually no difference between the subsystems for a test case and a test suite, these subsystems are called *Test Composites* in the following. A Test Composite consists of further subsystems, which represent either a Test Method or another Test Composite. The model itself corresponds to a Test Composite with the only difference that it contains additionally the test objective. Figure 2.7 shows the architecture of such a slUnit model and Figure 2.8 shows an example model.

With slUnit the tests are executed as the model is simulated, which means that the signals and states of the model are calculated for a number of simulation steps. Each simulation should execute only one of the defined test methods. Therefore the Assert blocks have to be activated and deactivated automatically at run-time, so that only those Assert blocks are enabled whose test method is executed. This is done by the major element of the slUnit framework, the Multiplexer block. It enables the Assert blocks of the active test method and disables all other Assert blocks. Furthermore it switches the outputs of the active test to the input of the

test objective. In consequence, the Multiplexer block is part of each test composite, either the top-level model or further subsystems, and allows a nested architecture of tests.

The set of tests which should be executed is defined by the same techniques as with xUnit. *Enumeration* is only supported for a single test method, which is run by using the designated "Run" button within the test's subsystem. *Discovery* is realised through a "Run All" button at the top-level of the model. For selecting specific groups of tests, the user can create additional blocks, which execute those tests, whose subsystems also contain a block with the same name. In consequence, this name translates the concept of marking tests with attributes to slUnit.

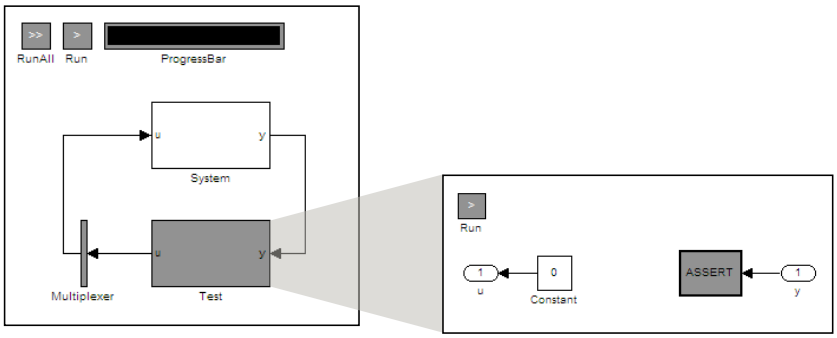


Figure 2.8: Example for a slUnit system

The results of a test are displayed as the background colour of the subsystem and are inherited from all assertions, which means that the colour of the block is set to red if at least one Assert block failed. With an additional simulation command it is also possible to execute all tests of a test composite. The result of this is displayed as the background colour of the subsystem and inherited from the results of the test methods or test composites included within this subsystem.

2.2.5 Fixtures

Fixtures share common code between test methods, usually in combination with the test case class, aggregating all these methods. They prevent duplication and assure that each test is independent of the others even if external resources are used. For this reason the methods for test preparation and clean-up are always called before and after the execution of each test method.

Generic Concept (xUnit)

The xUnit family provides the methods `set_up()` and `tear_down()` to manage the fixture. The following pseudo-code shows two example methods which test for a simple switch with two states: the test method `test_off()` asserts that the switch is off (default), the test method `test_on()` checks that it is on:

```
external ignition;
var switch;

set_up() {
  switch = new generic_switch(ignition);
}

test_off() {
  assert_equals(false, switch.is_on());
}

test_on() {
  switch.on();
  assert_equals(true, switch.is_on());
}

tear_down() {
  switch.off();
}
```

The duplication of the variable `switch` is avoided by using the method `set_up`, which is called before each test method. The switch is furthermore connected to the ignition of a car. It has to be ensured that this external resource is released independent of the test results. This is realised with the method `tear_down`, which is always called after each test method has been executed.

Realisation (slUnit)

Fixtures are realised with slUnit by aggregating the test methods into a test composite, so that it is possible to share common code by extracting it to the same level as the test methods. The outputs of the test methods are then no longer only the inputs of the test objective, but also of the common code, whose outputs themselves

become inputs of the test objective and/or the test methods. The test composite in Figure 2.7 uses this concept. For this model-based common code there is no need to release the Fixture, except for interaction with the MATLAB workspace which can be done using a callback function.

2.2.6 Exceptions

Exception handling is a mechanism available in (most modern) programming languages, for example Java, to handle the occurrence of a condition, called *exception*, which changes the normal control flow of execution and signals an error. Usually exceptions do not occur unexpectedly, but are designed by the developer to enforce the termination of the current method and activate an error handler. Therefore it should be possible to test the implemented exceptions.

Generic Concept (xUnit)

If the developer wants to test for an expected exception, the exception has to be caught and ignored, and the test fails if the exception is not thrown. Continuing the example from the last section, the switch should throw an exception if it is switched on twice:

```
test_already_on() {
  try {
    switch.on();
    switch.on();
    fail();
  }
  catch (SwitchAlreadyOnException e) {
  }
}
```

If the switch is raising an exception within the second call of `switch.on()`, the `catch` branch is activated and the test passes. Otherwise, the control flow goes on to the next statement, `fail()`, which lets the test fail.

Realisation (slUnit)

Simulink has no comparable concept, therefore slUnit does not realise a technique to test for exceptions.

2.2.7 Further Patterns for slUnit

This section describes additional patterns for slUnit, which provide solutions for questions that arise from working with slUnit.

Integration of the Test Objective

In most cases the test objective is part of a bigger system and therefore integrated into a complex Simulink model, here called complete system. Its subsystem is then part of at least two Simulink models – the test bed and the complete system. With TDD, we assume that the developer implements the test objective within the test bed. For this reason it has to be ensured that changes made to the test objective at the test bed are reflected in the complete system. This can be achieved by different solutions:

Library approach: The subsystem, which realises the test objective, is developed inside a Simulink library. Both the test bed and the complete system contain only a link to the library block. Every change is made in the library. The references between library, test bed and complete system are automatically renewed when the model is loaded, updated or simulated.

Copy-To-System approach: The test objective is developed together with the test cases in the test bed. After finishing the development, the subsystem is copied by the developer or by a script to the complete system. Every time additional changes are necessary, these changes have to be done in the test bed and the copying has to be repeated. As there is no possibility to prevent the modification of the test objective when it is integrated into the complete system, e.g. by adding a read-only attribute, the risk of having two different versions of the test objective in the test bed and in the complete system is high.

Model Referencing approach: With version 6 of Simulink a new concept was introduced, which allows a model to include other models as components. Thus the models are not copied, but referenced by their filenames. With this approach, large hierarchies of subsystems can be created by reusing modular components (the referenced models). Furthermore incremental loading and code generation is supported [134]. Employing this model referencing approach, the test objective is created in a separate model and referenced in the test bed as well as in the complete system.

The difference between the first and the third approach is the way library blocks are handled by Simulink in contrast to referenced models. Simulink does not allow

to modify a test objective which is linked to a library within the test bed, unless the developer breaks the link. Such a broken link can be manually restored, but as with all manual processes this is error-prone. In contrast, Simulink always uses transparently the source file of a referenced model. This means that even if the user tries to open and modify the subsystem of a referenced model the original model file is changed. Therefore this approach is the preferred one.

Differentiation between Test Objective and Test

A way to differentiate visually between the test objective and a test is useful to support the navigation through the model and the understanding of the tests. Therefore, the direction of the signal flow is inverted for the modelled tests. The common signal flow of Simulink is from left to right [135], i.e. the inputs of the test objective are on the left side, the outputs on the right side, and the signal lines mainly have a horizontal orientation. Hence, with the inverted signal flow the inputs are arranged on the right side and the outputs on the left side. Furthermore, this clarifies the correlation between the test objective and the tests as the outputs of a test are the inputs of the test objective and vice versa.

2.2.8 Summary

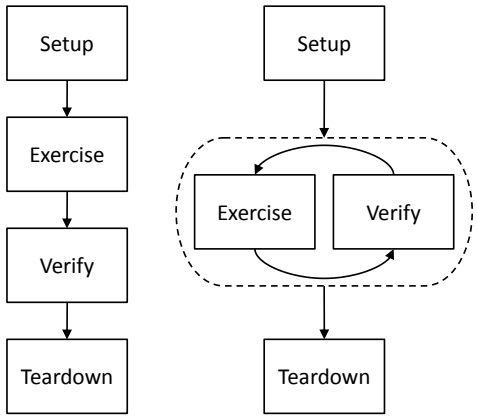
In the previous subsections the concept and the realisation of the testing framework slUnit were presented, which translates the patterns of the xUnit family to the graphical modelling tool Simulink. The resulting architecture is summarised in table 2.2 in comparison to the components of xUnit. slUnit is fully integrated into Simulink and provides two additional basic elements: the Assert block and the Multiplexer block. The test objective as well as the tests are represented as subsystems within a single model – the test bed. While the Assert blocks check whether a test has passed or failed, the Multiplexer blocks switch between the tests and assure their independence.

A test is executed by simulating the model. Thereby xUnit defines four distinct phases for each test, called the four-phase test as shown in Figure 2.9(a) [68]. With slUnit, the phases *setup* and *teardown* play a minor role as they are primarily used for the interaction with the MATLAB workspace, e.g. for loading named parameters. Although the phases *execute* and *verify* are also processed consecutively, this subsequent order is only appropriate for a single simulation step. In general, the process can be described as *in parallel* due to the time-dependent behaviour of Simulink, cf. Figure 2.9(b).

In consequence, it is possible to define and execute tests with slUnit which fulfil the

	xUnit	slUnit
Assertion	Method	Block
Fixture	Class	Subsystem
Test Case	Class	Subsystem
Test Method	Method	Subsystem
Test Suite	Object	Subsystem
Test Runner	GUI	Simulation

Table 2.2: The architectures of xUnit and slUnit



(a) xUnit (b) slUnit

Figure 2.9: The four-phase test with xUnit and slUnit

requirements of xUnit and consider the characteristics of Simulink. slUnit therefore allows a combined approach to model-based and test-driven development.

2.3 Using Tests for Verification

The common goal of testing is the verification of implementation, design and architecture of the system and the validation of the requirements. Therefore the term *Testing* is defined by the IEEE as follows: [136]

> (1) The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component.

> (2) The process of analysing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software items.

Testing is a process of technical investigation, which is intended to reveal quality-related information about the product with respect to its requirements and operational context. This includes, but is not limited to, the process of executing a system with the intent of finding errors [137]:

> Testing is the process of executing a program or system with the intent of finding errors.

Further definitions can be found in [138, 139].

With traditional development processes like the Waterfall model, the Spiral model or the V-model (see Chapter 1.3), the activities of testing are carried out after the implementation of the system. The following sections describe these activities in terms of testing techniques as well as test design and test evaluation techniques.

2.3.1 Testing Techniques

The techniques for testing can be divided into two main groups which are defined by their point of views:

What is tested?

In general, there are three levels within a software being tested: units, groups of units, and the system. The testing of the first level is called *unit testing*, in which the smallest entity, the software unit, is executed to verify that it is coded correctly. Usually such a unit consists of a single function or method, a single class or a single subsystem.

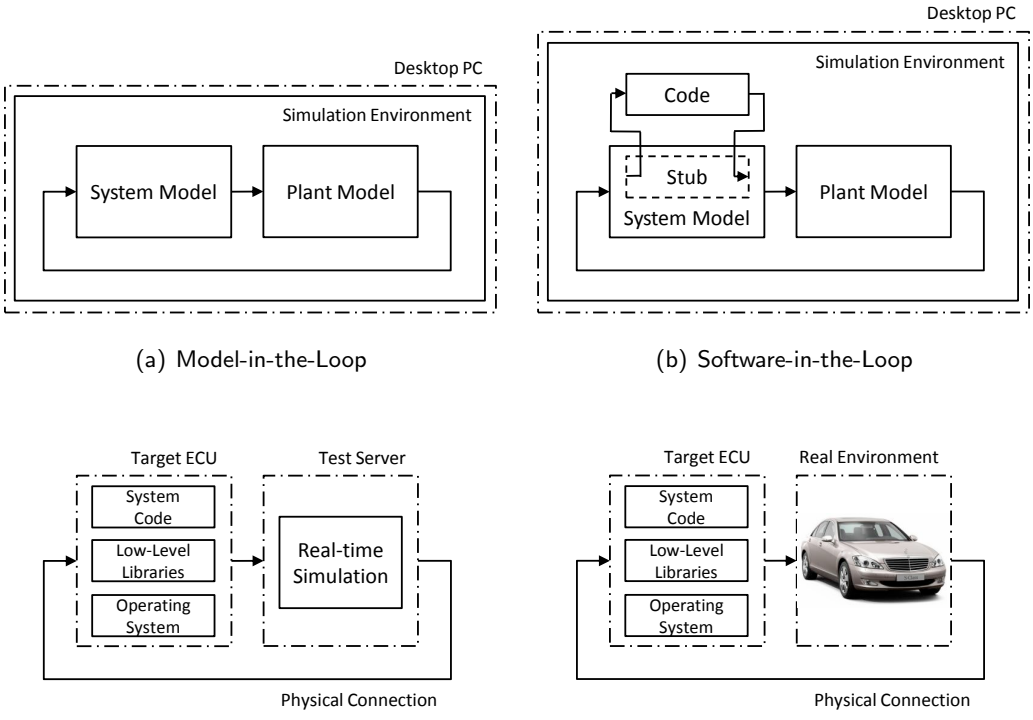
When those small components are combined and tested as a group, this is called *integration testing*. The purpose of integration testing is to confirm the system's design and architecture. Therefore the groups are executed together through their interfaces to check the specified interaction between them. Usually the number of units within a group is progressively increased during this process.

Finally, the software is integrated to form the final product and then tested as a complete system with *system testing*. The purpose of system testing is to evaluate the system's compliance with the specified requirements. A special kind of system

testing is *acceptance testing*, which is usually conducted by the customer[3] at the end of a development cycle or prior to delivery.

How is it tested?

Even early prototypes of traditional software products are usually verified within their final environment, e.g. a word-processing program on a PC. In contrast, the testing of embedded systems requires simulation environments to (a) get defined test conditions by simulating exact environmental situations, and (b) minimise damage by malfunctions. This reduces the costs as defects can be found before building the real system, e.g. an electronic brake system of a car, and without any risks for the system's host or the environment, e.g. crashing the car. Therefore several kinds of test setups are commonly used for the model-based development of such systems [44, 50].



(a) Model-in-the-Loop (b) Software-in-the-Loop

(c) Hardware-in-the-Loop (d) Target

Figure 2.10: Different test setups for embedded systems [44]

[3]The customer does not necessarily have to be an external person, but can also be defined as a role within a software development team.

With *model-in-the-loop* (MiL) tests, the implementation model of the system (the test objective) is simulated in a closed loop together with a plant model which represents the real environment, see Figure 2.10(a). In this setup, the system is not executed in real-time and the simulation is run on a desktop PC[4]. A similar approach are *model tests* (MT). Model tests do not require a closed loop, but stimulate the test objective with specific inputs, e.g. defined with the classification-tree method (see Section 2.3.2).

After the automatic code generation from the implementation model and the following compile process, the target software is tested with *software-in-the-loop* (SiL). Using the test setup of model-in-the-loop, the test objective is replaced by a stub which executes the object code, cf. Figure 2.10(b). Again the simulation is performed on a desktop PC disregarding real-time aspects, but the compiled object code can either run on the same PC or on a special evaluation board with the target CPU. Such an evaluation board is connected to the desktop PC using an I/O interface which communicates with the stub. The goal of this test setup is to verify the results of the source code generation as well as to analyse differences between the modelling language and the generated source code, which result for example from different data types or imprecise mathematical library functions.

The fully integrated system is first executed with *hardware-in-the-loop* (HiL) testing using its target hardware and physical I/O interfaces, cf. Figure 2.10(c). The electronic control unit is therefore physically connected to a test bench. It provides a real-time simulation of the environment and can optionally include or control further hardware components, e.g. hydraulic actuators. Therefore HiL tests aim to show the correct interaction between the modelled system, the operating system, and I/O drivers, as well as the proper implementation of the ECU's interface. Furthermore they are used for the validation of the system, which can not be tested or only at high costs in reality, e.g. a crash simulation for an occupant restraint system.

The last setup is called *target testing*[5]. Here the test bench of the previous setup is replaced by the real environment. This means that the target software and hardware are tested with the real I/O interface, periphery, user interface, network management etc. This might also include further systems, e.g. an object vehicle, see Figure 2.10(d).

The relationship between these setups and the four test scenarios described above is shown in Figure 2.11. While unit testing is only feasible with MiL and SiL testing,

[4] If the test execution can be done (semi-)automatically, e.g. with a number of batch tests, it is also possible to run the simulation on a kind of testing server.

[5] Other publications call this setup also *system testing* [44], this is avoided here to prevent confusion with the same term from the previous section.

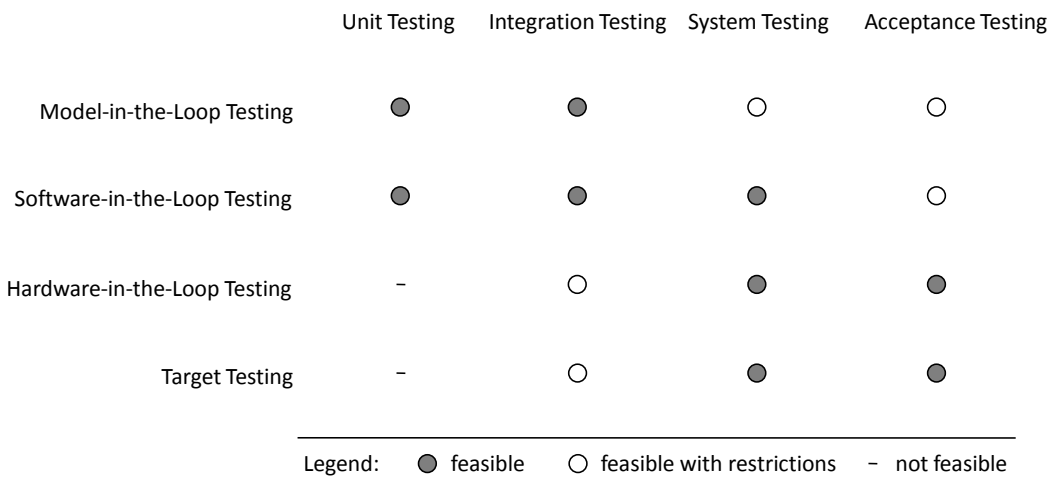


Figure 2.11: Relationship between different testing techniques

integration testing can also be used with HiL and target testing, when the group of jointly tested software units includes at least the operating system. A special use case for such tests is the verification of the system's interface and network management, which is usually done for the first time on an early development stage. For example, during the prototype stages of a new car the correct implementation of the communication between all electronic control units is more important than the real functionality of these systems. In contrast, both system testing and acceptance testing can be combined with all test setups, but some restrictions have to be considered for the tested model and software as both do not implement the whole system.

2.3.2 Test Design Techniques

Since software testing was first introduced, developers have thought about how to create tests systematically and therefore defined so-called *test design techniques*. Test design techniques allow the structured derivation of test cases by means of formal methods, which can be divided into two domains: black-box testing and white-box testing.

Techniques for Black-box Testing

The aim of black-box testing is to verify whether the test objective satisfies its specification or not. Therefore only the interface and the requirements of the test objective are taken into consideration for the definition of a test. The test's result are evaluated

by analysing the output values with regard to the input stimuli and the specification. No internal knowledge about the implementation is used, the developer considers the test objective only from an external perspective – like a black-box [137]. Typical methods for this technique are:

Classification-Tree Method: The *classification-tree method* divides the input domain of a test objective into classification and classes and represents them as a tree [140, 141, 142, 143, 144, 145]. A classification influences the functional behaviour of the system in general, whereas a class is a subset whose members result in the same kind of behaviour (equivalence partitioning). Each classification consists of one or more classes. A test case is generated by combining classes from different classifications.

Figure 2.12 shows an example classification tree for the controller of a collision avoidance system with six example test cases.[6] In addition to the textual descriptions shown, classes can have additional attributes, especially the corresponding signal or state name and a certain value or a range of values, e.g. $[0.6, 0.8]$ for the value of the road friction at wet conditions. Moreover it is possible to define test sequences, which consists of a number of test cases and allow the transition between them, e.g. the transition between wet and dry friction.

Evolutionary Algorithms: *Evolutionary algorithms* are based on the evolution theory of Darwin. Test cases are generated from a "start population", which changes through "recombination" and "mutation" in an iterative testing process [146]. The so-called survival of the fittest is the driving force to find defects of the test objective. With evolutionary algorithms, testing is transformed into an optimization problem, which formulates the test aim as a search for input values that fulfil the respective test aim, e.g. finding defects or verifying requirements [147, 148].

State Transition Testing: Different modelling notations were described in chapter 2.1.1, including statecharts. With *state transition testing*, test cases are derived from the states and transitions of these models, composing a state-event table and a transition tree [50]. While the state-event table consists of all possible and impossible combinations of states and events, the transition tree describes valid paths through this table starting from the initial state. The aim of the technique

[6] The classes and classifications marked with [..] are left out for simplification, the complete tree is more complex.

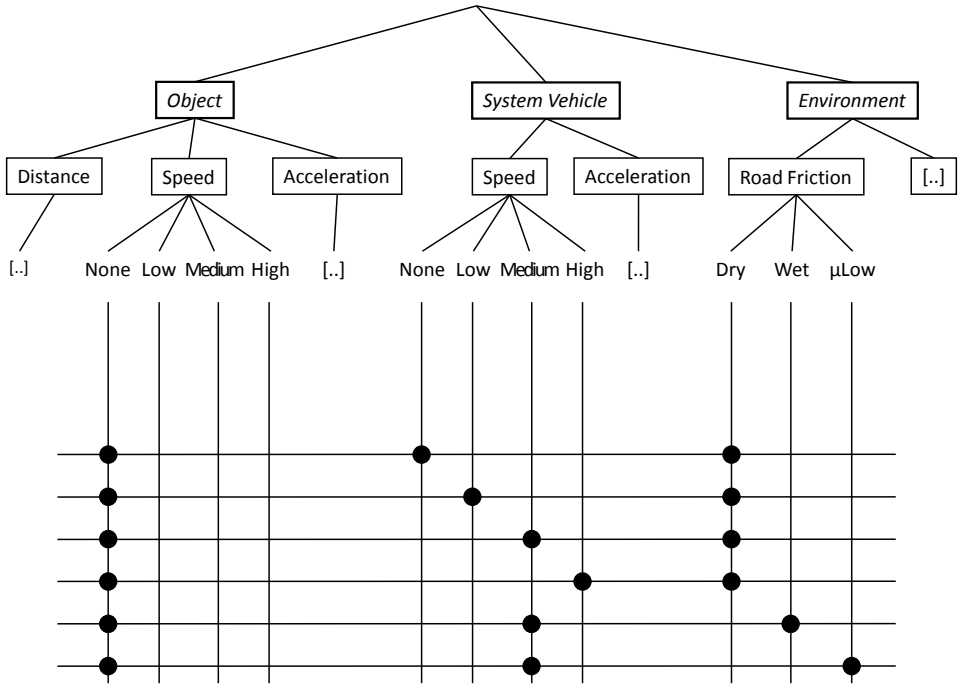


Figure 2.12: Example classification tree for a collision avoidance system

is to cover all of these paths and to test for the impossible combinations. State transition testing is only possible, if the test objective is or can be modelled as a statechart. A similar technique is *control-flow testing*. Here the test cases are derived from a flowchart of the test objective and are created in such a way, that they cover all possible combinations of actions (paths of the diagram) [149, 150].

It should be noted that state transition testing as well as control flow testing can be considered as both black-box and white-box techniques depending on their objective. They are used for black-box testing if the underlying model is the description of the specification of the system, and for white-box testing if it serves as the implementation of the system. Thus with model-based development the boundaries between both techniques can become blurred.

Statistical Usage Testing: The purpose of *statistical usage testing* is the simulation of the real situations of a system to detect defects concerning the operational usage. The probability of the different events is determined (according to the current state and the history of the events or states) to produce a relevant set of test cases [151, 152]. Events, which occur very infrequently (rare), are often relevant for safety aspects of the systems. In contrast to statistical usage

testing, but with similar methods, *rare event testing*, can be used to handle these events [50].

Techniques for White-box Testing

In contrast to black-box testing, white-box testing uses an internal knowledge of the system to design tests. Here, the tester defines input stimuli to exercise paths through the code and determines not (only) the appropriate outputs, but also how the elements of the control flow are exercised by the tests. Therefore, three major coverage criteria define the techniques for white-box testing [153]:

Statement Coverage: With *statement coverage*, each statement of a control flow is executed at least once. It ensures that no dead code exists within the system. Statement coverage is considered as the weakest coverage criterion, because the singular invocation of every statement does not guarantee the coverage of every logical path through the control flow and is in consequence only sufficient for the detection of simple defects [137].

Branch Coverage: The set of tests is defined in such a way that all decisions or jumps are tested at least once. Therefore both the then-branch and the else-branch of an if-statement have to be passed through, including also empty branches. In reality this is often complicated, because subsequent conditions are commonly dependent on prior decisions and the resulting paths. This might lead to complex or even impossible input constellations.

Path Coverage: *Path coverage* requires that all paths of a program are tested. In practise, this is often impossible as even small programs can have a huge or infinite number of paths due to conditional loops. Therefore several other structural testing strategies have been developed to reach a high percentage of path coverage without the need of verifying all paths, e.g. boundary-interior path testing [154].

Figure 2.13 shows an example control-flow model of a simple activation algorithm for a collision avoidance system based on the time left until a collision would occur. The physical meaning of these variables is explained in Section 3.1. The transitions of this model are labelled with capital letters. 100 per cent statement coverage within this example can be realised through the input vectors $(0, 0, 0)$, $(5, 10, 10)$ and $(5, 10, 5)$[7], which cover the paths *ABCL*, *ABDEGJK* and *ABDEGHIEGJK*. For

[7] Note that other combinations of input values also provide this.

a complete branch coverage only the path *ABDEGHIEFL* is missing, therefore another input vector is defined $(5, 10, 15)$. Finally the last possible path *ABDEFL* is addressed by $(5, 15, 15)$ reaching 100 per cent path coverage. Please note that the path can never be executed by a correct implementation of the algorithm. Hence, the occurrence of this path indicates a wrong assignment of the variable *ds*.

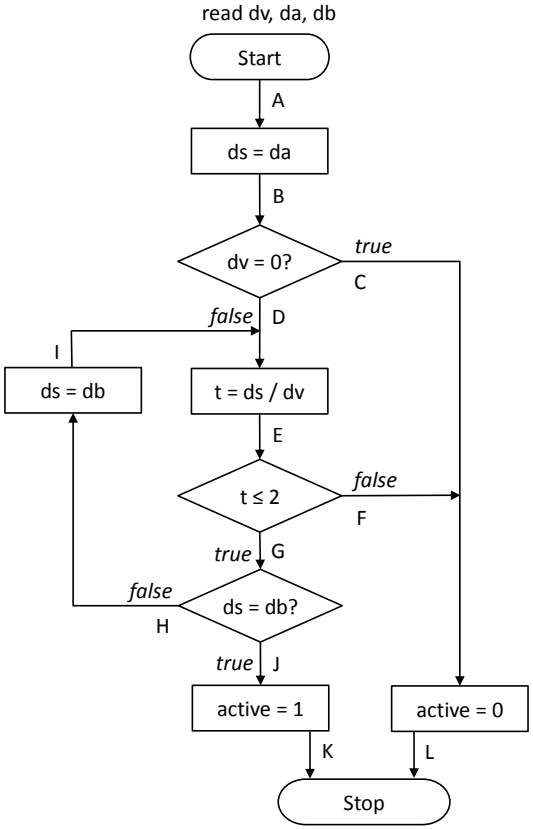


Figure 2.13: Example control-flow model with labelled transitions

The techniques commonly applied to reach a high percentage of these coverage criteria basically consist of the above mentioned state transition testing and control-flow testing, and another technique called *elementary comparison testing*. This technique evaluates the paths of a function by identifying its logical conditions in the source code and translating them into pseudo-code [155]. The pseudo-code is used to specify so called test situations, i.e. to find input values which satisfy the identified conditions and those which do not. Tests are created from these situations trying to cover each result of each condition at least once. In contrast to state transition testing and control flow testing, no model of the test objective is necessary.

Another group of white-box techniques is based on data-flow analysis. They evaluate the interaction between the definitions of a program variable and subsequent references to it. The simplest type is called 2-dr interaction and consists of one definition and one reference reached by this definition [156]. Testing each of these interactions is similar to statement coverage, i.e. branch coverage can not be guaranteed. Hence, the required pairs strategy creates a set of pairs for each 2-dr interaction depending on the occurrence of the definition or the references [157]. If a definition is placed for example within a loop, two iteration counts are used: one for exiting the loop as soon as possible, the other for a distinct number of iterations. Other data-flow based strategies are presented in [158, 159].

2.3.3 Test Evaluation Techniques

In general, the determination of the test's result is carried out by the comparison of the outputs with expected values that arise from the requirements of the test objective. This comparison can be either done manually or automatically. The advantage of the automatic evaluation is that an overall result can be created, generating a statement about the success of the test execution – pass or fail. With xUnit, this is for example displayed as a red or green bar, see Chapter 2.2.3.

Typically, the automatic evaluation is done by two techniques, assertions and regression testing, which are described in the following sections.

Assertions

The concept of the assertion, which is represented by a Boolean expression, was first defined by Robert Floyd in 1967 to express the intended behaviour of a program [160]. Based on this concept, Hoare introduced the following notation in 1969 (today known as the Hoare triple):

$$P\{Q\}R \tag{2.1}$$

It means that if the assertion P is true before the execution of the program Q, then the assertion R is true after its execution [161]. The intention of this notation is the formal verification of the program. In practise, assertions were often used for design-by-contract [162] and for the run-time checking of program states [163, 164, 165]. With the techniques of extreme programming and test-driven development they first became important for the evaluation of test results [54].

Embedded systems typically interact with the real world through sensors and actuators. The signals processed or created by them are dependent on the time, i.e. the value of a signal changes over time. This change can be either discrete or continuous

for the time as well as the signal magnitude. Therefore the signals are divided into four groups: analogue, time-discrete, value-discrete or digital [166]. For the evaluation of such signals the Boolean expression of the assertion has to be extended by a time-dependent behaviour. Different techniques can be realised based on the Assert block from chapter 2.2.3 [167, 168]:

Boundary Checks: *Boundary checks* evaluate whether the values of a signal remain inside or outside of a specified range. It is possible to define either static or dynamic borders. In addition, one-sided boundaries can be implemented by setting the opposite border to infinity. A special case of boundary checks is the comparison of a signal with a constant expected value (both borders are set to the same value), which can also be extended by symmetric and asymmetric tolerances (the upper border then equals the expected value plus a tolerance, the lower border equals the expected value minus a tolerance).

Gradient Checks: The *gradient check* is actually not a separate technique, because the gradient of a signal can be considered as an independent signal whose evaluation is carried out by boundary checks. Therefore the gradient check is only listed as an example for a group of checks, which combine a signal processing unit with a boundary check.

Type Checks: *Type checks* involve two tasks: The verification of the signal's data type and its range. This is particularly useful when the co-domain is a set of discrete values.

Regression Testing

Regression testing generally specifies the re-running of all or a part of all test cases to identify the impact of modifications of already successfully tested systems [137, 138]. Here test's outputs are evaluated by comparing them with the outputs of a previous test execution. A difference is made between actual regression tests which test different versions of the same representation of the system, e.g. different versions of the model, and the so-called *back-to-back tests* which check the equivalence of different representations of the system, e.g. between the model and the source code [169], cf. Figure 2.14. The outputs of the first test execution have to be initially evaluated, because no reference values exist at this point of the testing cycle. This initial evaluation can be done manually, i.e. by "looking" at the signals and then setting them as the new reference, or automatically with the help of assertions.

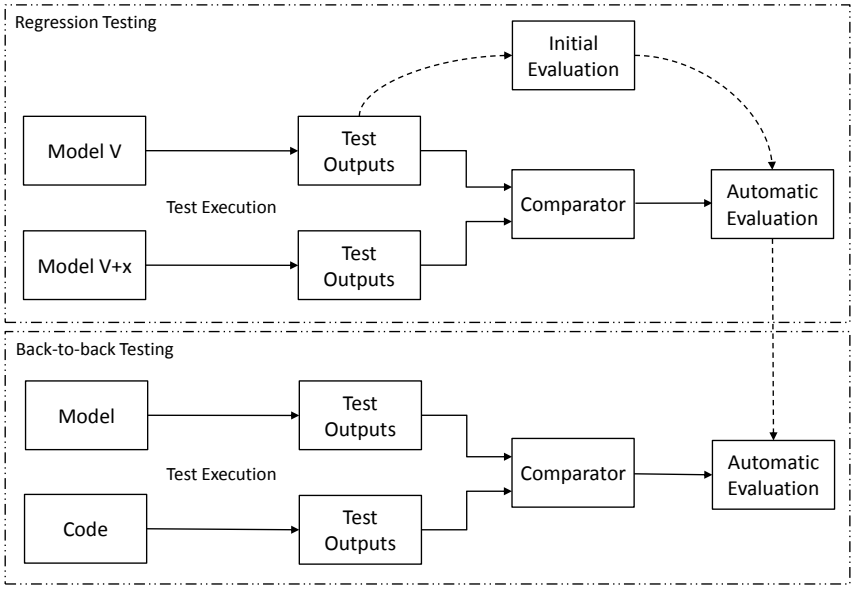


Figure 2.14: Difference between regression and back-to-back testing

The evaluation of the test's outputs, i.e. the system's reaction to specific input stimuli, and the expected values, i.e. the reaction of a different version or representation to the same stimuli, is based on the comparison of two signals. In doing so, it is not important that both signals are equal, but their similarity is checked for different criteria. These criteria can consider phase shifts, different computational accuracies, and the impossibility of an exact reproduction of a physical signal in reality. Signals are classified as similar with several methods, which are selected depending on the context:

Difference Calculations: The difference between the amplitudes of the output and reference values is calculated for each simulation step. It is possible to generate either absolute or relative differences. The signals are classified as similar if the minimum and maximum difference is within a specified tolerance. In addition, the tolerance can be modified depending on the characteristics of the signal, e.g. orthogonal to the signal, depending on the gradient, or through circles around the sample point creating a tolerance tube. Thus a phase shift between two signals can be compensated for. More complex techniques can also pre-process the signals, e.g. the difference matrix method first rearranges the output signal to sufficiently fit the reference signal for identified periods of time and then applies the difference calculation [170].

Statistical Methods: These techniques are based on the computation of statistical values, for example the cross-correlation coefficient, which is calculated with the following formula [34]:

$$c(T) = \frac{1}{T} \int_0^T y(t)y'(t-\tau)dt \tag{2.2}$$

y is the reference signal, y' is the output signal, and $c(T)$ is the cross-correlation coefficient depending on the time T, which denotes the analysed time span of the signals. The maximum value of c marks the most probably time lag between the two signals and gives an indication about the similarity of the signals [171]. Other such values are the signal-to-noise ratio and the total harmonic distortion [50].

The fundamental characteristic of these statistical methods, namely the filtering of disturbance values, is simultaneously its biggest disadvantage as, for example, amplitude peaks are disregarded, although they might be important in the context of the test's evaluation.

2.4 Using Tests for Design

In contrast to the process described above, the motivation for testing with test-driven development is not the verification of the test objective, but its implementation (explicit) and its design (implicit). The tests are therefore created and executed before the implementation. Details have already been outlined in the description of the development cycle of TDD in chapter 1.3.2.

Test-driven development can be used with all testing techniques, but two types are most commonly applied: acceptance testing and unit testing [172, 173].

In the acceptance test the expected behaviour of the developed system is described from the customer's point of view. The requirements or, in general, the use cases are translated to an executable form. The time horizon of acceptance tests is a mid to long-term period and covers at least one iteration of the development process. In fact, the aim of an iteration can be defined as the completion of a set of acceptance tests. For such tests all three software levels and all four testing environments (cf. Chapter 2.3.1) are sufficient, but typically the highest level is chosen to test the system in its real environment. Based on this high-level test, a number of subtests can be created which consider the characteristics and features of the respective level and environment (top-down approach).

The following rules describe the role of acceptance tests within the development cycle of TDD:

> Rule 1 : All requirements of a system have to be covered by acceptance tests.

> Rule 2 : A new requirement is only allowed to be implemented if the respective acceptance test fails.

In contrast, the goal of the unit tests is the realisation of the requirements based on the smallest entity, the software unit. The idea behind this is that every change must follow a failing test, no matter how small the change is. In other words, the rules for unit tests are:

> Rule 3 : The functionality of a unit has to be covered by unit tests.

> Rule 4 : A unit is only allowed to be modified if the respective unit test fails.

Rule 4 requires implicitly that if a defect occurs which is not detected by an unit test, e.g. if it is reported by the customer, then this defect has to be first reproduced by a test before the unit is modified.

The unit tests can be designed using the techniques of black-box testing as well as white-box testing providing the variety of both approaches. Furthermore a classification into two types of unit tests is proposed [84]. The first group analyses and specifies the behaviour of the system ("what it should do"), while the second group helps to implement the controller in a systematic and traceable way ("how it is done"). In addition, a third group is necessary for the verification of the unit and its implementation process [174].

The development of the units is done with a bottom-up approach, i.e. first a single unit is tested and realised, then a number of units can be integrated for further development. However, the testing of two or more units together can also start before the unit is completely implemented, e.g. to develop the interaction between the units.

The interrelationship between acceptance and unit tests is shown in Figure 2.15. The activities "write an acceptance test" and "run the test(s)" can have several meanings in the context of model-based development depending on the test environment:

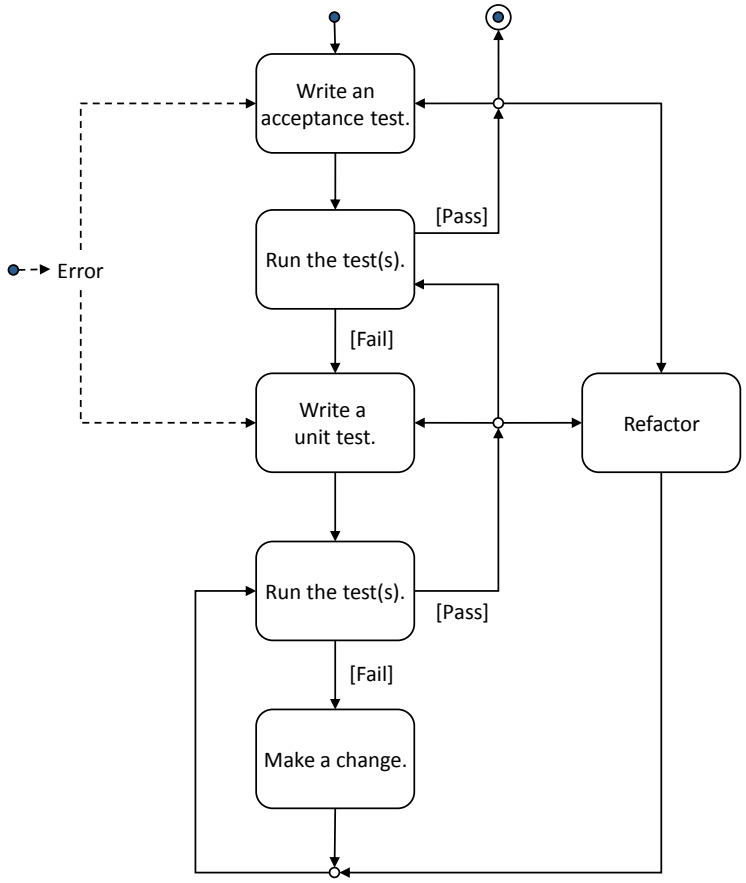


Figure 2.15: Acceptance and unit tests within the development cycle of test-driven development

- With MiL and SiL environments, an acceptance test consists of a plant model, its initial values and the (time-dependent) definition of additional stimuli, e.g. disturbance values. The test is executed by simulating this model together with the test objective.

- The definition of HiL tests is governed by the simulation environment of the test bench and usually achieved by the specification of test vectors or scripts which control this environment. Thus the variables for the models of the environment, e.g. the vehicle's environment with the road's course or other vehicles and the driver, are configured based on events, e.g. the driver will hit the brake pedal when the distance from the vehicle ahead is less than 20 meters. Then the tests are run automatically by the test bench.

- A target test is typically described through the manual execution of the system by its user, e.g. a typical acceptance test for a vehicle system is a (driving) ma-

noeuvre [175, 176]. This execution can be partly or fully automated depending on the kind of the system. Continuing the example, a partial automation can be achieved by steering or braking robots as well as sled tests (crash tests). Other methods include so-called endurance or stress tests, during which the system is either run for a longer period or stimulated with a high workload.

In contrast, the activities of unit testing are always carried out at the model level, i.e. a test is written by building a model and run by simulating both the model and the test objective.

The evaluation of the tests is achieved by the methods described above, see Chapter 2.3.3. Here, the use of assertions is not limited to the automated tests with MiL, SiL and HiL, but can be also used with target testing. The driver compares the system's reaction by means of measured data with the expected results after he has performed a manoeuvre, and marks the test with passed or failed.[8]

Finally the developer is allowed to "make a change" in terms of model-based development if at least one unit test fails. The unit is implemented by creating or modifying a model exactly in such a way that the test is passed.

Another important activity of test-driven development is *refactoring*. Refactoring specifies the improvement of the system's structure while retaining its present behaviour [177, 178, 179]. Thus, this step relies on the results of the tests as they provide an executable form of the system's specification and requirements. The methods of refactoring include the improvement of readability and comprehensibility as well as the iterative refinement and optimization of architecture and design. Moreover, the developer draws conclusions from reviews of the implemented models and from the unit tests, e.g. the duplication of code, to modify the system in terms of modularity, interfaces, maintainability, and testability.

2.5 Testing Different Layers of an Automotive Safety System

The different environments for the testing of embedded systems have already been described in chapter 2.3.1. They are the basis for the following layer approach for test-driven development of an automotive safety system, which is used to illustrate the concepts introduced in this chapter.

The topmost layer represents the system vehicle within its real environment and is therefore considered as target testing, see Figure 2.16. The primary parts of this

[8] Usually some kind of tabular template is used which lists the expected behaviour, offers some space to describe the measured performance and a column for check marks.

environment are the street surface and course, other cars, pedestrians, and surrounding objects such as trees or road signs as well as additional effects like weather or lighting conditions. TDD on this layer means to drive the car without or with a partial implementation of the system's requirements. Obviously, such tests with an automotive safety system such as a collision avoidance system will cause severe damages whenever the test fails (which is a rule of test-driven development). Usually mock objects are used to avoid such problems. The aim of the tests is to analyse the vehicle's behaviour in measuring data even without the complete implementation of the test objective. The measurements are then used for parameter estimation, to stimulate the test objective as an open loop system and for the comparison between the car's reaction with and without the test objective.

Figure has been removed due to Copyright restrictions.

The second layer considers the system vehicle as its point of reference. The real environment is described by the means of sensors and actuators, i.e. its set of attributes is reduced to a set which is specified by the interfaces and signals. This layer corresponds to hardware-in-the-loop testing. Its purpose is the test-driven development of the system under the aspects of the vehicle's architecture. Especially for the build-up of early prototypes it is typically more important to establish the vehicle's communication network than to fulfil all (functional) requirements of the test objective. A common use case is the development of the control units' error handling. Here, a test inserts different errors, e.g. the timeout of a network signal, into the vehicle's communication layer. The reaction of the control units can be automatically evaluated with assertions defining the expected behaviour of their output signals and the onboard diagnosis [180].

The third layer is defined by a single electronic control unit (ECU), the fourth layer by a software unit. With both layers the set of information is further reduced to the set of signals which is available either at the interface of the ECU or at the interface of the software unit. The testing can be applied at the environments SiL and MiL. The goal of test-driven development with these layers is the design and implementation of the system's entities as described in Chapter 2.4. Using the three levels of testing, cf. Chapter 2.3.1, the upper three layers can be considered as system, integration and unit testing in terms of the vehicle. The lower three layers can be considered as system, integration and unit testing in terms of the control unit with respect to its software.

The input stimuli for the test-driven development of all these layers are generated

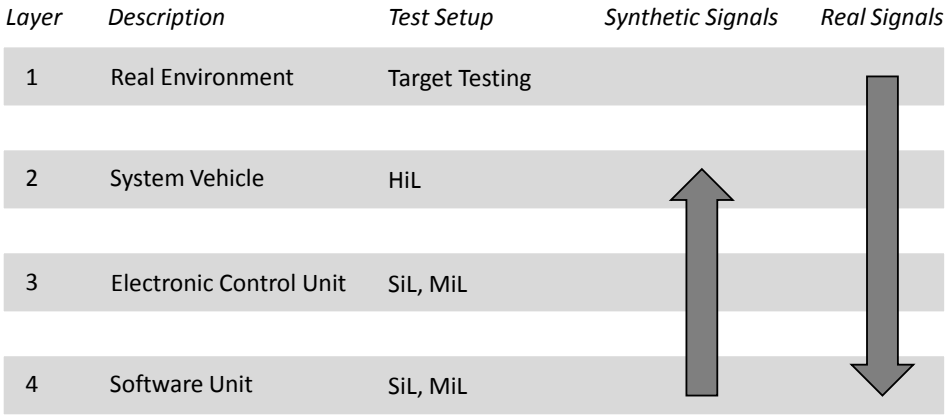


Figure 2.16: Layers for an automotive safety system

with either synthetic values or real physical signals. Signals with synthetic values can be used seamlessly from the bottom layer up to the second layer, but not exactly reproduced in the real environment. The so-called ascending continuity therefore ends at the second layer. In contrast, it is possible to reuse real physical signals in the form of measured data from the top layer at all other layers (descending continuity), see Figure 2.16. Moreover, for both continuities it might be necessary to convert the signals between two layers as for example the system vehicle's velocity at the first layer is represented as wheel impulse counters at the second layer and translated into a speed at the fourth layer. An alternative to the conversion of the signals is the measurement of the signals on all boundaries between the layers.

Chapter 3

Development of a Collision Prevention System

Every year millions of people get injured or die due to road traffic accidents. One of the main types of accidents is the rear-end collision, which accounts for about 30 percent of the overall number of accidents. For instance in the USA, more than 2,000 people lost their life and more than 500,000 people needed to be treated in a hospital due to the consequences of a rear-end collision in 2005 [6]. A rear-end collision happens if a first car collides with the rear of a second car, while both cars are driving in the same direction.

This chapter describes the test-driven development of a system to prevent such rear-end collisions or mitigate their impact. The development process is based on the methods and tools which have been presented in the previous chapter. First different approaches for collision avoidance are discussed. Then we describe the requirements and the architecture of the new system and identify four example components. The realisation of three of these components illustrates the application of test-driven development to different algorithms for embedded control systems. The design and implementation of the fourth component is presented in Chapter 4, which focuses on specific aspects of control theory.

3.1 Collision Prevention Systems

3.1.1 Overview

Active safety systems which focus on preventing rear-end collisions can be divided into three groups: collision warning (CW), collision mitigation (CM) and collision avoidance (CA) systems. Common to all systems are components which try to detect a potential collision between two cars. These components are based on the one hand on sensors which measure the environment as well as the internal states of the vehicle equipped with the system (compare Section 1.1.2). On the other hand they use algorithms for the assessment of the situation, which are usually realised in form of a kinematic analysis in a n-dimensional space [181, 182, 183, 184].

When a potential collision is detected, collision warning systems can activate a visual, acoustical or haptical warning. If the driver does not react to this warning, a collision avoidance system tries to prevent the collision through active methods like steering or braking. Such approaches are commonly based on conservative vector fields [185], on methods of control theory [182] or on geometrical trajectory planning [181].

In contrast to CA systems, the purpose of collision mitigation systems is to reduce the consequences of a collision, rather than to prevent it. This is usually accomplished by restricting the system to a subset of the vehicle's abilities, e.g. a partial braking. Such systems are used because they are easier to implement in the context of verification and validation against faulty activations and product liability. Furthermore the impact of a CA system might be degraded to the impact of a CM system as the information about environment and driver are limited. For example, if the system vehicle approaches a stationary obstacle on the outer side of a curve, the last point to avoid the collision by braking lies earlier in time than the last point by steering. This means that the driver is still able to avoid the collision by steering into the curve, although the system has detected a potential collision in a straight direction. As long as a CA system is not able to detect the driver's intention, it can not autonomously start a full braking manoeuvre until the collision is unavoidable by common driving manoeuvres.

3.1.2 Definition

The aim of the system, which has been developed during this PhD project, is to warn the driver about a potential rear-end collision as well as to actively support him in preventing such a collision. Therefore it can be assigned to the group of collision

warning systems and collision avoidance systems. The system is called Collision Prevention System (CPS). The further description of the system is based on the following items:

Road: The road is only specified through its friction coefficient, μ, which depends on the longitudinal position, s.

$$x_{road} = \mu(s) \tag{3.1}$$

System Vehicle: The System Vehicle is the car in which the Collision Prevention System is installed. It is defined by its acceleration, a_{sys}, its speed, v_{sys}, and the distance travelled in longitudinal direction, s_{sys}, which all depend on the time, t.

$$x_{sys} = \begin{pmatrix} s_{sys}(t) \\ v_{sys}(t) \\ a_{sys}(t) \end{pmatrix} \tag{3.2}$$

The relationship between s, v and a can be described as follows:

$$s = \int v(t)dt = \int\int a(t)dt \tag{3.3}$$

The origin of the earth-fixed coordinate system is laid to the frontmost point of the car, see Figure 3.1.

Driver: The Driver is the person who controls the System Vehicle. In terms of the CPS, his or her behaviour is described by only two variables: the brake pedal travel, s_{pedal}, and the brake pedal speed, v_{pedal}.

$$x_{driver} = \begin{pmatrix} s_{pedal}(t) \\ v_{pedal}(t) \end{pmatrix} \tag{3.4}$$

In addition, we can define a deceleration, a_{drv}, which results from the driver's pressure on the brake pedal. This deceleration depends on different variables, for instance the friction coefficient of the Road or the speed of the System Vehicle. The physical details are neglected here as they are not relevant for the realisation of the Collision Prevention System.

Object: The object or vehicle with which the System Vehicle would collide during a rear-end collision is called the Object. The state vector of the Object is similar to the one of the System Vehicle, the only difference is the origin of the coordinate system which is defined by the Object's rearmost point, cf. Figure 3.1.

$$x_{obj} = \begin{pmatrix} s_{obj}(t) \\ v_{obj}(t) \\ a_{obj}(t) \end{pmatrix} \tag{3.5}$$

Based on these state vectors, we define the relative acceleration, $\Delta a(t)$, the relative speed, $\Delta v(t)$, and the relative distance, $\Delta s(t)$, between the Object and the System Vehicle:

$$\Delta s(t) = s_{obj}(t) - s_{sys}(t) \tag{3.6}$$

$$\Delta v(t) = v_{obj}(t) - v_{sys}(t) \tag{3.7}$$

$$\Delta a(t) = a_{obj}(t) - a_{sys}(t) \tag{3.8}$$

Furthermore the following condition for the initial value of Δs is assumed:

$$\Delta s(t = 0[s]) > 0[m] \tag{3.9}$$

A rear-end collision then takes place, if the relative speed is smaller than zero at the point in time, t_c, at which the relative distance is equal to zero.

$$c_{col} = \begin{cases} true & \text{if } (\Delta v(t_c) < 0) \wedge (\Delta s(t_c) = 0) \\ false & otherwise \end{cases} \tag{3.10}$$

c_{col} is a Boolean value that denotes whether a collision takes places or not. Vice versa the collision is prevented, if at t_c the relative speed is greater than or equal to zero, or in other words, if the mathematical function of the relative distance has a global minimum at this point.

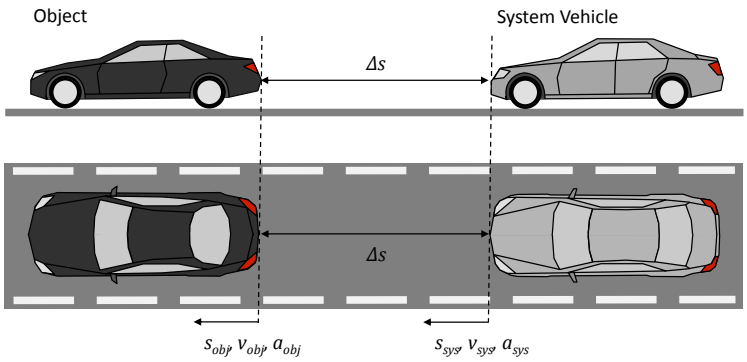


Figure 3.1: Definition of a rear-end collision

3.2 Requirements

The systematic analysis of the system's requirements is one of the primary tasks of software development. It determines the needs and conditions which need to be met by the new system by taking into account a customer's point of view. For an automotive safety system, the source of those requirements is typically not a customer, but the development team itself. The ideas and concepts for such systems are generated by innovation workshops, analysis of market needs or publicly funded research projects. This input is then transformed into formal requirements, typically stored in a database such as DOORS by the company Telelogic [186].

In this section, the main requirements for the new Collision Prevention System are presented. We divide them into three groups: The top-level requirements consists of the two basic aims of the system. Then, the functional requirements are described for each aim. Please note that in reality the list of requirements is much longer and might also include the hardware configuration, non-functional requirements or the specification of external interfaces.

Top-Level Requirements

The following requirements describe the top-level requirements of the Collision Prevention System.

Requirement Warning function

> The system shall warn the driver of situations which might lead to a rear-end collision.

Requirement Brake support function

> The system shall support the driver in preventing a rear-end collision by controlling the brakes of the System Vehicle.

Functional Requirements for the Warning Function

The following requirements describe the condition to activate the Warning function. It is based on the reaction time which is defined as the time left to the driver until a collision is unavoidable. A more precise definition of the reaction time is given in Section 3.4.3. The requirement Warning output specifies how a warning should be indicated to the driver. Figure 3.2 shows a possible realisation of this requirement.

Requirement Warning activation

The system shall issue a warning if the remaining reaction time is less then 2.5[s].

Requirement Warning deactivation

The system shall deactivate the warning if the remaining reaction time is greater then 2.5[s].

Requirement Warning output

If the warning is active, a visual warning shall be shown by a red light with a triangular shape, and an acoustic signal shall be issued by an intermittent beep tone.

Figure 3.2: Red light with a triangular shape in the instrument cluster (Photo: Robert Bosch GmbH).

Functional Requirements for the Brake Support Function

The following requirements describe the conditions at which the system shall be activated in terms of controlling the System Vehicle's brakes. Similar to the warning, the reaction time is used to detect a critical situation. Furthermore, the characteristics of the driver's braking manoeuvre (comfort or emergency braking) are taken into consideration for the activation. The difference between an emergency braking manoeuvre and a comfort braking manoeuvre is explained in Section 3.4.2. Finally, the last requirement specifies how the system shall control the brakes of the System Vehicle.

Requirement System activation

The system shall be activated if the driver is performing an emergency braking manoeuvre, and the remaining reaction time is less then 1.5[s].

Requirement No system activation

The system shall not be activated if the driver is performing a comfort braking manoeuvre, or if the driver is not braking.

Requirement System deactivation (collision criterion)

The system shall be deactivated if the collision was prevented.

Requirement System deactivation (driver criterion)

The system shall be deactivated if the driver releases the brake pedal.

Requirement System operation

If the system is active, it shall control the brakes of the car in such a way that the relative distance between the Object and the System Vehicle reaches a minimum value without a collision.

3.3 Architecture

In general, the design of the architecture of an embedded control system will take place at the beginning of the development process. The decisions about the basic design are made by analysing the system's requirements and environment. This environment is not only defined by the physics of the control loop, but also by the integration container, i.e. the existing hardware or software environment into which the system should be integrated. Therefore the architecture of the Collision Prevention System can be specified either by the means of control theory, i.e. as a closed-loop system using a state-space approach, or with the layer approach, which has been introduced in Section 2.5.

3.3.1 State-Space Approach

Process

With the state-space representation, the plant model of the CPS can be described as a linear system with n inputs and m outputs:

$$\dot{\underline{x}}(t) = A\underline{x}(t) + B\underline{u}(t) + D\underline{d}(t) \tag{3.11}$$

$$\underline{y}(t) = C\underline{x}(t) \tag{3.12}$$

$$\underline{x}(0) = \underline{x}_0. \tag{3.13}$$

$\underline{x} \in \mathbb{R}^p$ is the state vector with $\mathbb{R}$ representing the set of real numbers and p the order of the system, $\dot{\underline{x}}$ denotes the derivative with respect to time. $\underline{u} \in \mathbb{R}^n$ is the input vector and $\underline{y} \in \mathbb{R}^m$ is the output vector. $A \in \mathbb{R}^{p \times p}$ is called the state matrix, $B \in \mathbb{R}^{p \times n}$ the input matrix, $C \in \mathbb{R}^{m \times p}$ the output matrix and $D \in \mathbb{R}^{m \times n}$ the disturbance matrix. Figure 3.3 shows the block diagram of the state-space representation.

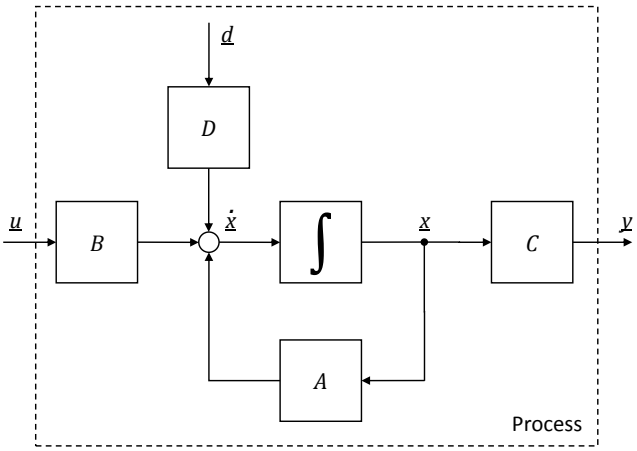


Figure 3.3: Block diagram of the state-space representation

For the control loop of the Collision Prevention System a state vector with three states – the relative distance, the relative speed and the relative acceleration (Equations (3.6), (3.7) and (3.6)) – is considered:

$$\underline{x}(t) = \begin{pmatrix} \Delta s(t) \\ \Delta v(t) \\ \Delta a(t) \end{pmatrix}. \tag{3.14}$$

The control signal is the desired acceleration of the System Vehicle, a_{set}.

$$u(t) = a_{set}(t) \tag{3.15}$$

The brakes of the System Vehicle can be represented as a first order behaviour with a gain of one.

$$\dot{a}_{sys}(t) = \frac{1}{\theta} a_{set}(t) - \frac{1}{\theta} a_{sys}(t) \tag{3.16}$$

θ is the response time of the brakes. Based on measurements on a test track, we assume $\theta = 0.25[s]$ as a suitable value for the vehicle model for which the CPS should be installed, i.e. the System Vehicle.

The disadvantage of Equation (3.16) is that it considers only the System Vehicle, but not the Object. By transposing Equation (3.8) to

$$a_{sys}(t) = a_{obj}(t) - \Delta a(t), \tag{3.17}$$

we can replace a_{sys} in Equation (3.16):

$$\Delta\dot{a}(t) = -\frac{1}{\theta}\Delta a(t) + \frac{1}{\theta} a_{obj}(t) + \dot{a}_{obj}(t) - \frac{1}{\theta} u(t). \tag{3.18}$$

We assume that the acceleration of the Object remains constant, i.e.

$$a_{obj}(t) = const \tag{3.19}$$

$$\Leftrightarrow \dot{a}_{obj}(t) = 0. \tag{3.20}$$

This results in the following state equation

$$\underline{\dot{x}}(t) = \begin{pmatrix} \Delta v(t) \\ \Delta a(t) \\ -\frac{1}{\theta}\Delta a(t) - \frac{1}{\theta} u(t) + \frac{1}{\theta} a_{obj}(t) \end{pmatrix} \tag{3.21}$$

with the initial condition

$$\underline{x}(0) = \begin{pmatrix} \Delta s_0 \\ \Delta v_0 \\ \Delta a_0 \end{pmatrix}. \tag{3.22}$$

The output value of the system is represented by the relative distance between the Object and the System Vehicle:

$$y(t) = \Delta s(t). \tag{3.23}$$

The Object's acceleration is interpreted as an external disturbance:

$$\underline{d}(t) = \begin{pmatrix} 0 \\ 0 \\ a_{obj} \end{pmatrix} \tag{3.24}$$

$$\underline{d}(0) = \underline{d}_0. \tag{3.25}$$

The matrices of the state-space approach for the CPS are then:

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & -\frac{1}{\theta} \end{bmatrix} \tag{3.26}$$

$$B = \begin{bmatrix} 0 \\ 0 \\ -\frac{1}{\theta} \end{bmatrix} \tag{3.27}$$

$$C = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \tag{3.28}$$

$$D = \begin{bmatrix} 0 \\ 0 \\ \frac{1}{\theta} \end{bmatrix} \tag{3.29}$$

For steady state, all elements of $\underline{\dot{x}}(t)$ have to be zero.

$$\underline{\dot{x}}(t) = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & -\frac{1}{\theta} \end{bmatrix} \begin{pmatrix} \Delta s(t) \\ \Delta v(t) \\ \Delta a(t) \end{pmatrix} + \begin{bmatrix} 0 \\ 0 \\ -\frac{1}{\theta} \end{bmatrix} u(t) + \begin{bmatrix} 0 \\ 0 \\ \frac{1}{\theta} \end{bmatrix} a_{obj}(t) \tag{3.30}$$

This results in

$$\Delta v_{ss} = 0 \tag{3.31}$$

$$\Delta a_{ss} = 0 \tag{3.32}$$

$$u_{ss} = a_{obj}, \tag{3.33}$$

and with Equation 3.17, the steady-state value of the System Vehicle's acceleration has to be equal to the Object's acceleration:

$$a_{sys,ss} = a_{obj} = u_{ss}. \tag{3.34}$$

State Feedback

The controller is now realised by a full-state feedback as shown in Figure 3.4,

$$u_{fb}(t) = -Kx(t), \tag{3.35}$$

with the matrix K consisting of one element for each state value:

$$K = \begin{bmatrix} k_{\Delta s} & k_{\Delta v} & k_{\Delta a} \end{bmatrix}. \tag{3.36}$$

As a consequence of equations (3.31) and (3.32), the values of $k_{\Delta v}$ and $k_{\Delta a}$ have no influence to the control variable at steady state, because the values of Δv and Δa are then zero. Instead, the steady-state value of the control signal is equal to the Object's acceleration:

$$u_{fb,ss} = -a_{obj} = -k_{\Delta s}\Delta s. \tag{3.37}$$

In other words, the steady-state value of Δs is only zero if a_{obj} has also a zero value. This contradicts the requirement System operation, which describes that the relative distance between the Object and the System Vehicle should be minimal after the collision was prevented, i.e. $\Delta s_{ss} = 0$. We therefore consider the Object's acceleration as a feedforward control input:

$$u_{ff}(t) = a_{obj}(t). \tag{3.38}$$

Finally, the control variable is defined by the sum of the feedforward control input and the feedback control input:

$$u(t) = u_{ff}(t) + u_{fb}(t). \tag{3.39}$$

To summarise, the state-space representation provides a mathematical model for the process and the controller of the Collision Prevention System. This model not only describes the relationship between the system's input and output variables, but also its internal behaviour.

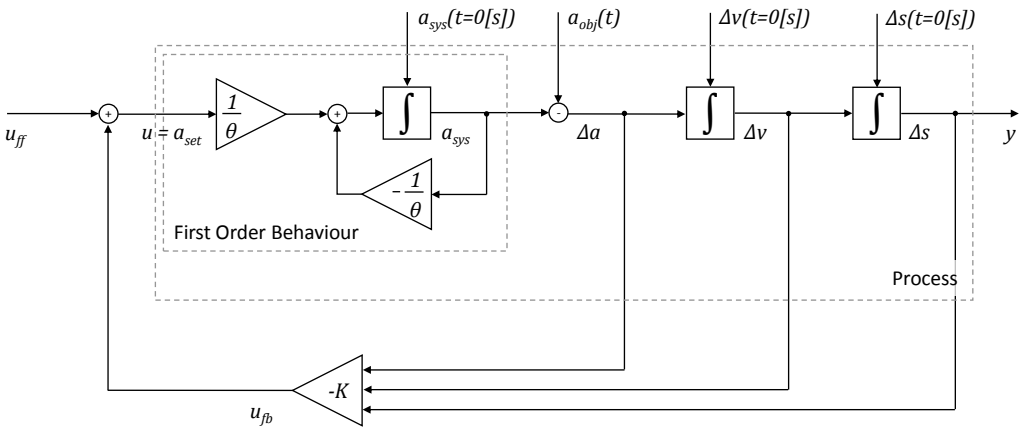


Figure 3.4: State-space approach for the Collision Prevention System

3.3.2 Layer Approach

In Section 2.5, a layer approach for test-driven development of automotive safety systems was introduced. Each layer represents a different level of abstraction of the system's architecture in a top-down approach. In the following, these layers are used to describe the architecture of the Collision Prevention System.

For the CPS, layer 1 is defined by the System Vehicle, the Driver, the Object and the Road as described in the last section, cf. Figure 3.1.

Then layer 2 consists of the following components: RADAR Sensor, Brake Control Unit, Engine Control Unit, Instrument Cluster, and RADAR Control Unit, which are connected by a Controller Area Network (CAN), see Figure 3.5. The RADAR Sensor measures the current relative distance between the Object and the System Vehicle, $\Delta s(t_i)$, the current relative speed, $\Delta v(t_i)$, and the current acceleration of the Object, $a_{obj}(t_i)$. t_i denotes the current processing or simulation step. The Brake Control Unit is connected to a sensor for the brake pedal travel, s_{pedal}, and an accelerometer to measure the System Vehicle's acceleration, $a_{sys}(t)$. Furthermore it computes the speed of the System Vehicle, $v_{sys}(t)$, from four wheel impulse counters. The three values are transmitted via the CAN. The Brake Control Unit also receives the requested acceleration, $a_{set}(t_i)$, from the RADAR Control Unit. As the driver should be able to override the system, the Brake Control Unit calculates the minimum value of the control variable and the acceleration that originates from the brake pedal, which is pressed by the driver. Moreover it communicates with the Engine Control

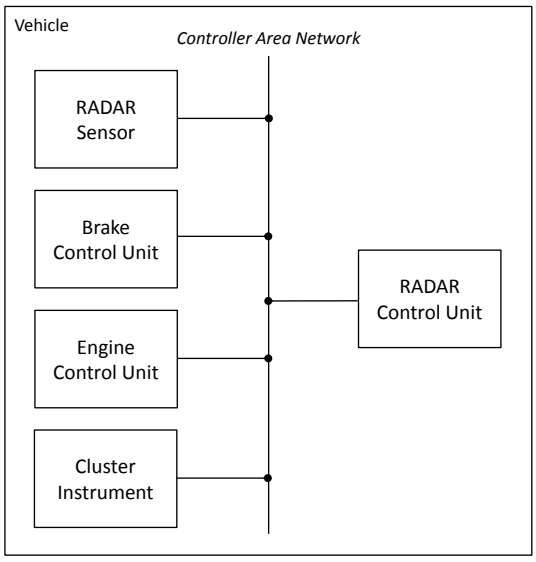


Figure 3.5: Layer 2: Architecture of the control units

Unit to realise a deceleration with the drag torque of the engine. The Instrument Cluster outputs a warning request by the CPS as a visual warning – a red light with a triangular shape, compare Figure 3.2 – and an acoustical signal – an intermittent beep tone. The CPS itself is realised in the RADAR Control Unit.

Layer 3 is described through the architecture of the RADAR Control Unit as presented in Figure 3.6. First, the input signals are read by the Input Interface, which also handles timeouts and data errors like a wrong CRC of the CAN messages. If such an error occurs, the input interface activates a corresponding error strategy, e.g. disabling the system. The main functionality of the Collision Prevention System is implemented in the next component. As shown in the diagram, this is in parallel with other components of different systems that use the same architecture, e.g. the Adaptive Cruise Control System. The Output Interface therefore realises an output arbitration of all components' outputs, so that only one system can be active at a time. Finally, the system's outputs are transmitted to the CAN.

The internal architecture of the component marked with Collision Prevention System in Figure 3.6 also belongs to layer 3. It consists of four subcomponents: Situation Assessment, Driver Assessment, Controller and Activation, as shown in Figure 3.7. The subcomponent Situation Assessment calculates the risk of a collision by using a kinematic analysis of the state vectors of Object and System Vehicle. The subcomponent Driver Assessment classifies a braking manoeuvre, which is executed by the driver, into two groups: comfort braking and emergency braking. While a visual and acoustical warning should be activated if the criticality of the situation is higher

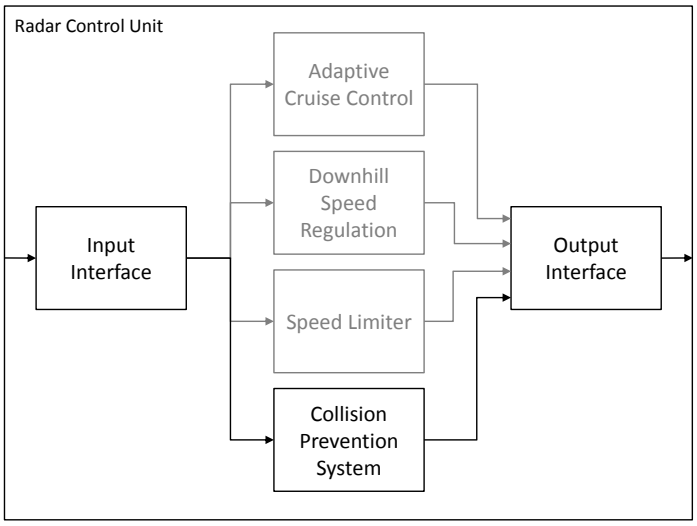


Figure 3.6: Layer 3: Architecture of the software components

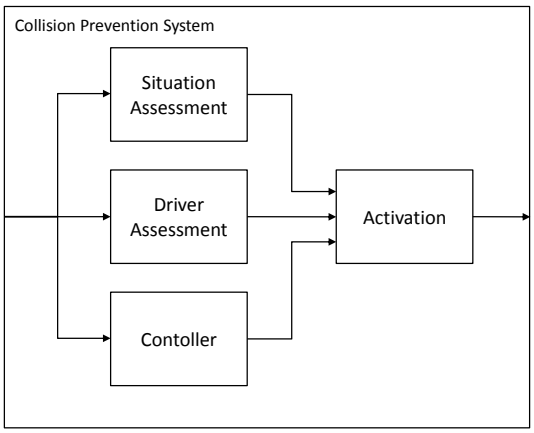


Figure 3.7: Layer 3: Architecture of the CPS subcomponents

than a threshold, the brake support must wait for an emergency braking manoeuvre by the driver before getting activated. This logic should be implemented in the subcomponent Activation. Finally, the control variable, i.e. the required acceleration to prevent the collision, is calculated with the subcomponent Controller.

Layer 4 is defined through the software unit. As the underlying implementation of the subcomponents described above is not known at the beginning of the test-driven development process, we initially consider these four subcomponents as units.

3.3.3 Comparison of the State-Space Approach and the Layer Approach

When we compare the state-space approach and the layer approach, the state-space representation does not consider neither the calculation of the minimum value of the driver's and the system's acceleration (layer 2) nor the activation of the system through the driver and situation assessment (layer 3). Therefore we define the input value of the brake, $u_{brake}(t)$, as the minimal value of both accelerations:

$$u_{brake}(t) = \begin{cases} a_{drv}(t) & \text{if } a_{drv} < a_{cps} \\ a_{cps}(t) & \text{otherwise} \end{cases} \tag{3.40}$$

$a_{drv}(t)$ is the deceleration resulting from the driver's pressure on the brake pedal, $a_{cps}(t)$ is the deceleration requested by the system to prevent a possible collision. This latter deceleration furthermore depends on whether the system was activated or not:

$$a_{cps}(t) = \begin{cases} a_{set}(t) & \text{if } c_{active} \\ 0 & \text{otherwise} \end{cases} \tag{3.41}$$

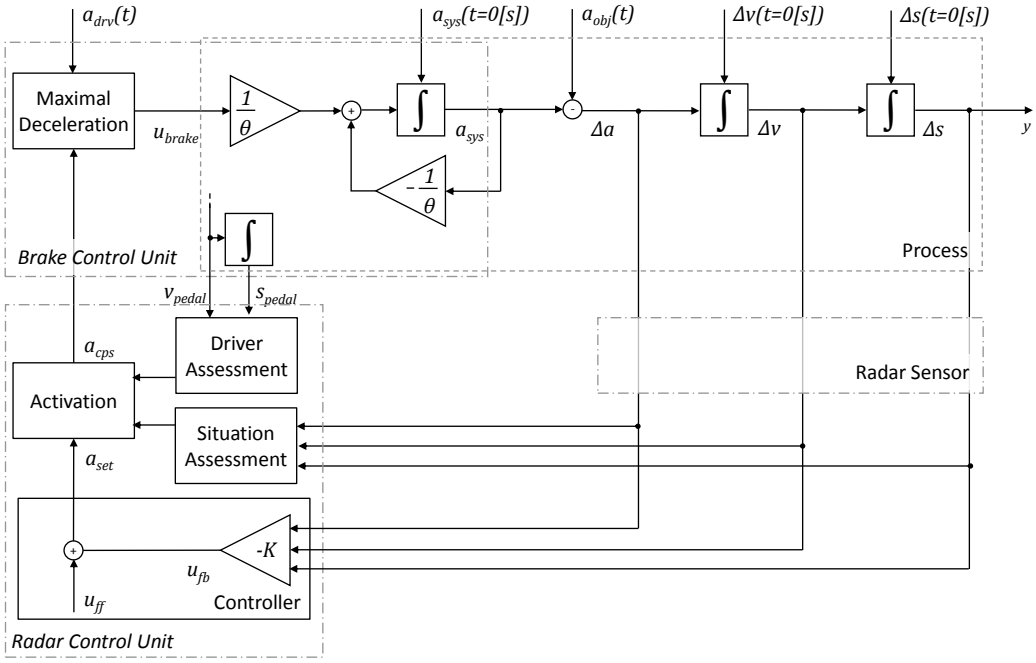


Figure 3.8: Combination of the layer and the state-space approach for the Collision Prevention System

$a_{set}(t)$ is the control variable of the state-space approach as defined by Equation 3.39. The condition c_{active} is generated by the subcomponent Activation, which logically combines the results of the subcomponents Driver Assessment and Situation Assessment. The corresponding block diagram is shown in Figure 3.8. It consists of the block diagram of the state-space representation, in which the subcomponents of the CPS have been integrated. In addition, dashed lines mark the hardware components and the plant model.

To summarise, the combination of the layer approach and state space approach provides a complete description of the system's architecture. On the one hand, it specifies the physical relationship between the input and the output signals. On the other hand, if clarifies the role of each component within the scope of its physical environment and its integration container. With this, we are now able to realise the CPS with a test-driven development process.

3.4 Realisation

This section demonstrates the test-driven development of three software components of the Collision Prevention System: Activation, Driver Assessment and Situation Assessment. These components were chosen as examples for the following objectives:

Logical combinations: The component Activation implements a number of logical combinations of its input signals to create two output signals. These output signals indicate whether the CPS should warn the driver or activate an additional brake support.

Experimental problems: The realisation of the component Driver Assessment is based on an experimental analysis of the driver's braking behaviour.

Mathematical algorithms: With the component Situation Assessment, the criticality of the driving situation is estimated by mathematical algorithms.

In addition, the Controller is described as a fourth component in Chapter 4 to show the application of TDD to control system design.

For all components, the development is driven by the definition and execution of acceptance and unit tests. The acceptance tests specify the requirements of the system from a customer's point of view and with this, the goal of the respective development iteration. After one or more acceptance tests fail, a number of unit tests define the interface, the functionality (what the system should do) and the realisation (how the system is realised) of a software unit. Finally, the component can be implemented as a Simulink model.

3.4.1 Activation

The component Activation should realise one or more logical combinations of its input values to create two Boolean output signals. The first output signal, c_{active}, should be true if the system is active, i.e. the CPS should control the car to prevent a collision. In contrast, the driver should be only alerted by a visual and audible warning if the second output signal, c_{warn}, is true.

Concept

For logical combinations, it is important to ensure that all logical paths are covered by the implementation. Therefore we propose an additional step which transforms the system's requirements into a classification tree. The classification-tree method

has already been introduced in Section 2.3.2. Using a classification tree editor, we define two compositions for the classifications of the input and the output interface of the component. The classifications represent the corresponding Simulink signals. They are refined by classes for these ranges of the signals, which cause the same behaviour. Then we create test cases by logically combining the classes.

The classification tree helps us in two ways. Firstly, it supports the translation of the system's requirements into the system's implementation. To create the tree, we describe the component in terms of input and output signals as well as their relationship. Secondly, we can directly derive acceptance and unit tests from the test cases, cf. Figure 3.9. Therefore it can be seen as the starting point for the realisation of the component.

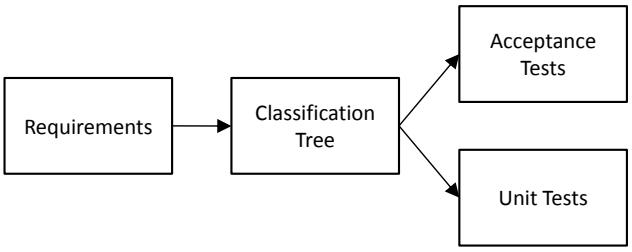


Figure 3.9: Concept for deriving tests from the classification tree

The resulting tree for the component Activation is shown in Figure 3.10. c_{driver} is the result of the component Driver Assessment, *true* means that the driver is performing an emergency braking. t_{react} is the result of the component Situation Assessment. It represents the reaction time which is left to the driver until a collision is unavoidable. a_{set} is the output signal of the component Controller, i.e. the control variable. Please note that some classes are marked with a "?" at certain test cases. If it is set for an input variable, then the variable's value is not relevant for the result of the logical combination. If it set for an output variable, this variable should keep its state, i.e. retain the value of the last computation.

Realisation

The realisation is done by the test-driven development cycle, which has been explained in Section 2.4. The first step of this cycle is the definition of acceptance tests. For the CPS, each acceptance test includes a driving manoeuvre. Such a driving manoeuvre has to be described from the customer's point of view. This means that especially the input stimuli have to be influenceable by the driver. In contrast, the verification of the manoeuvre can be also done by the analysis of internal data, e.g. recorded

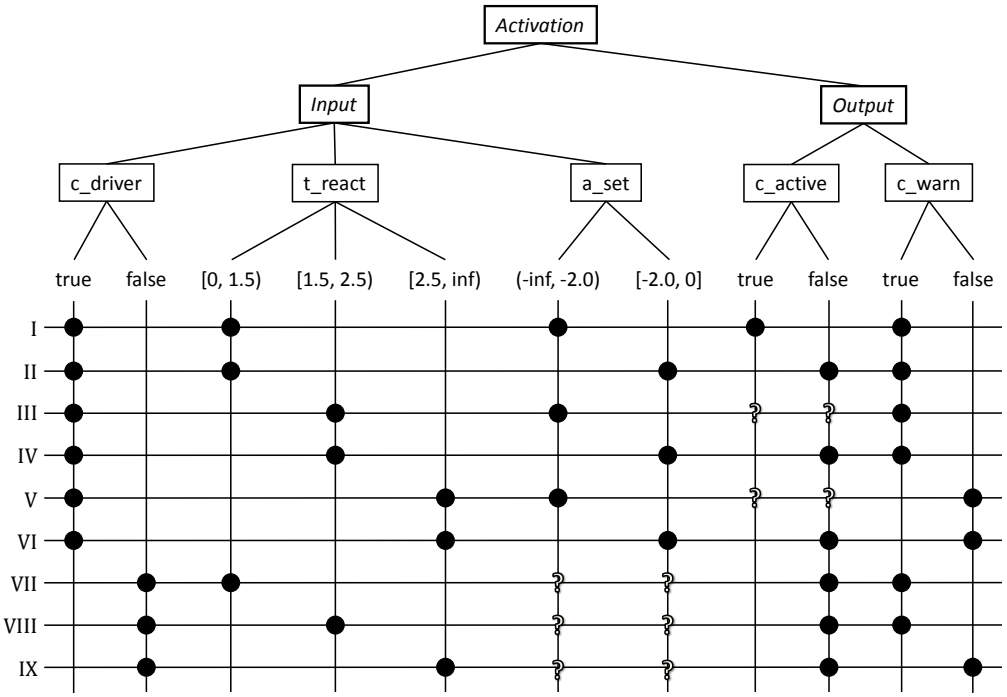


Figure 3.10: Classification tree for the component Activation

by a measurement system. Such an acceptance test for the component Activation is detailed in the following with the acceptance test Activate CPS.

Acceptance Test Activate CPS

Description: The System Vehicle approaches a foam object with a speed of $60[kmh^{-1}]^*$.[1] The reaction time is marked with a gate of pylons, see the setup of Figure 3.11. At this point the driver starts to brake, but only so much that the collision is not avoided by him.

$$\Delta s(t = 0[s]) = 100[m]^*$$

$$\Delta v(t = 0[s]) = -60[kmh^{-1}]^* \begin{cases} v_{obj}(t = 0[s]) = 0[kmh^{-1}]^* \\ v_{sys}(t = 0[s]) = 60[kmh^{-1}]^* \end{cases}$$

$$\Delta a(t = 0[s]) = 0[ms^{-2}]^* \qquad \begin{cases} a_{obj}(t = 0[s]) = 0[ms^{-2}]^* \\ a_{sys}(t = 0[s]) = 0[ms^{-2}]^* \end{cases}$$

$$\Delta s_{gate} \cong 39[m]^*$$

[1] The meaning of the asterisk $(*)$ is explained later in the text, see Page 78.

Assertions: The visual and acoustical warning are activated if the reaction time is smaller than 2.5[s], i.e. $c_{warn} = true$ for $t_{react} < 2.5[s]$. The optical warning is activated by a red light with a triangular shape within the Instrument Cluster, the acoustical warning by an intermittent tone.

The system is activated in terms of braking if the driver executes an emergency braking and the reaction time is smaller than 1.5[s], i.e. $c_{active} = true$ for $(t_{react} < 1.5[s] \wedge c_{driver} = true)$. The value of a_{cps} is then smaller than $-2[ms^{-2}]$. The system stays active until the collision is prevented, i.e. a_{cps} gets greater than $-1[ms^{-2}]$.

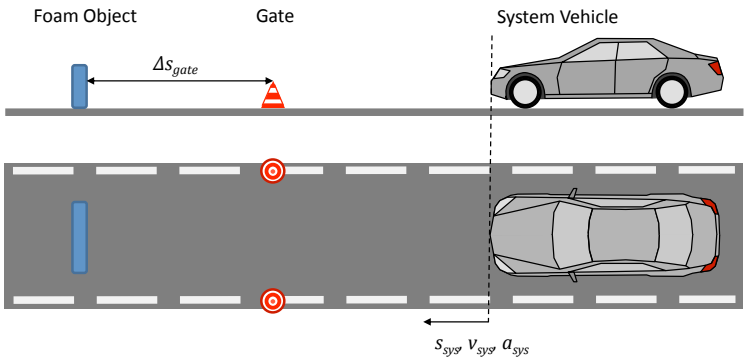


Figure 3.11: Test setup with a foam object and a gate of pylons to mark the reaction time

This test introduces three typical concepts for acceptance tests of the Collision Prevention System:

1. A mock object is used to avoid damage of the System Vehicle. Because of the test-first paradigm, the test is executed before the system is implemented. For the CPS, this means that a collision will take place as the aim of the CPS is the prevention of a collision. For stationary targets we use a simple foam object or a retractable corner reflector. For moving objects we use a so-called *Target Simulator*, which consists of a real car and a dummy object. The dummy object is mounted on the side of the real car with an electronically controlled mechanism to flip it up just before a collision might occur.

2. The evaluation of the assertions can not only be done by measuring internal data of the electronic control unit, but also by the use of external means. This allows to assess the system's behaviour from an objective point of view, i.e. without using knowledge about the structure and components of the system.

The applicability depends on the asserted value and might require a high effort. For the CPS, it is suitable to mark a certain reaction time to a stationary object by a gate of pylons, but this does not work for the reaction time to a moving object. The reason for this is that the gate is placed at a certain position, which has a fixed distance to the Object. Thus for a moving Object, this position can not be marked by pylons. An alternative would be a reference measurement system, e.g. a differential GPS, to detect the distance between Object and System Vehicle and display this distance to the driver.

3. The parameters of the test can be either specific or arbitrary. With the example test, all input stimuli, i.e. the initial values of Object and System Vehicle, are arbitrarily selected. We indicate this with an asterisk ($*$). This means that these parameters can be modified without changing the intention of the test. In contrast, all asserted values are specific for the test and its objective. For instance, the expected threshold value for the reaction time is always $1.5[s]$, because this threshold value was specified within the requirements of the system. Two things have to be noted: Firstly, the threshold values might be hidden by certain values of the input stimuli, e.g. when changing $\Delta s(t = 0[s])$ to $10[m]^*$, the reaction time is already smaller than $1.5[s]$ at the beginning of the test. With this, the assertion would be also fulfilled by a realised threshold value of $3.0[s]$, although this contradicts the system's requirements. Secondly, the complexity of the test's execution might be increased by certain values of the input stimuli. An example is the different effort for marking the reaction time to stationary and moving object, see above.

The next step is the definition of unit tests. These tests can be either one-to-one converted from the classification tree or derived from its decomposition. An example for the direct conversion is the following unit test:

Unit Test Activate

Description: The input signals are set to the following values:

$$c_{driver}(t) = true \tag{3.42}$$

$$t_{react}(t) = 1.5[s]^+ \tag{3.43}$$

$$a_{set}(t) = -2[ms^{-2}]^+ \tag{3.44}$$

Assertions: The following values are asserted for the output signals:

$$c_{active}(t) = true \tag{3.45}$$

$$c_{warn}(t) = true \tag{3.46}$$

The values of t_{react} and a_{set} are only partially arbitrary, thus marked with a +. This means that other values are possible, but have to be within the specified range of the corresponding class. For example, a value of $t_{react} = 1.0[s]$ belongs to the same class, while $t_{react} = 2.0[s]$ does not and therefore results in a different output.

The alternative method to derive a unit test is the decomposition of the classification tree. This is based on the separate analysis of the conditions, which set the different output variables. For instance, c_{warn} shall be always *false* if t_{react} is greater than $2.5[s]$ and always *true* if it is smaller than $2.5[s]$.[2]. Hence the number of tests to drive the realisation of c_{warn} can be reduced from 9 to 2. A similar approach can be applied to c_{active}. c_{active} has to be *false* if either c_{driver} is $false$, or a_{set} is greater than or equal to $-1[ms^{-2}]$.

With the last group of tests, the output signal c_{active} should keep its value from the last cycle (marked with a "?"). The choice of the correct class, i.e. *true* or *false*, depends on the previous state of the signal. If the system is already activated, it should remain active, otherwise it should remain deactivated. In contrast to the previous tests, the system must first be activated before we can test that the state remains active for the given input configuration. Therefore we usually would create a test sequence with the classification tree, which consists of two test steps, cf. Figure 3.12: First c_{active} is set *true* by a corresponding configuration of input stimuli, then this

[2]In practise, we add a hysteresis to the reaction time, but we neglect this here for simplicity.

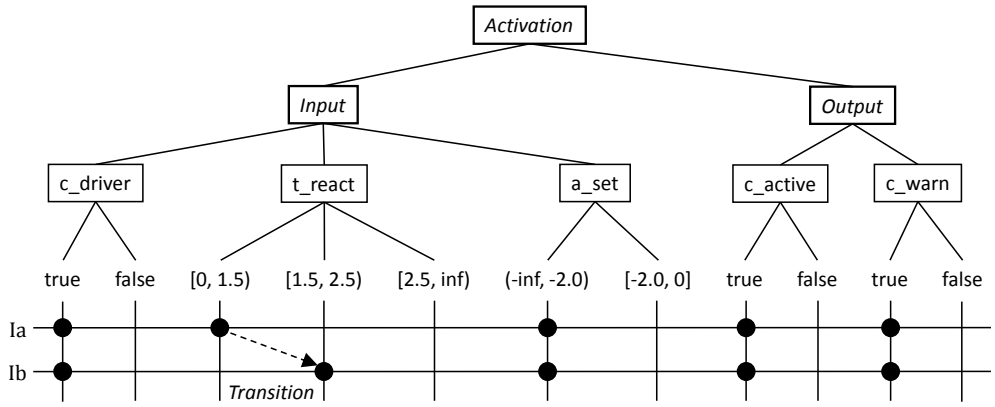


Figure 3.12: Test sequence with the classification-tree method

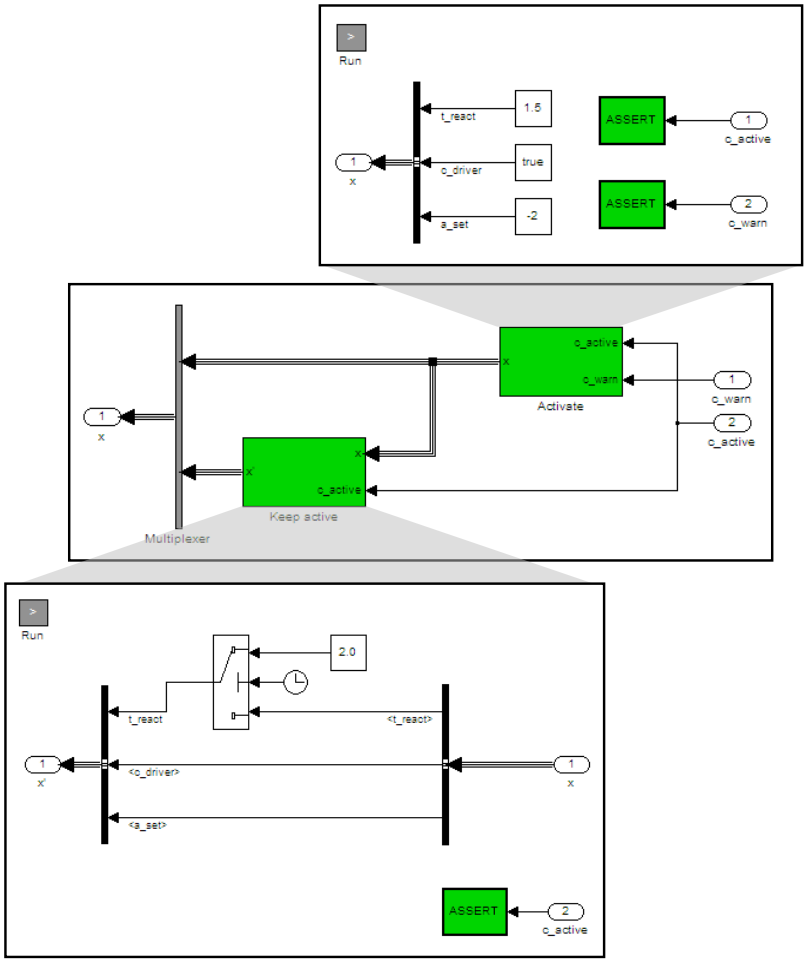


Figure 3.13: Test sequence with slUnit

value should be held for a different configuration of input stimuli. The disadvantage of this approach is the duplication of the logical combination for the activation, cf. line I at Figure 3.10 with line Ia at Figure 3.12. We can prevent this duplication with slUnit by creating a fixture based on the unit test Activate. Then this unit test provides the initial configuration for an additional unit test, Keep active, which modifies the test vector after $1[s]$ of execution, see Figure 3.13.

Unit Test Keep active

Description: This unit test builds a fixture with the unit test Activate. In addition, the input signals are set to the following values:

$$t_{react}(t \geq 1[s]) = 2.0[s]^{+} \tag{3.47}$$

Assertions: The following values are asserted for the output signals:

$$c_{active}(t) = true \tag{3.48}$$

With these three different kinds of unit tests, we are able to iteratively implement the component. After one unit test failed, we implement the logic by the corresponding Simulink blocks. For the test Activate, this results in three relational and one logical operator. Further blocks are added with the next tests. Then the last test, Keep active, requires a SR flip-flop in-between the Logical Condition blocks for c_{active} and the Output Port block. The SR flip-flop is able to remain in a constant state if both of its inputs, S and R, are *false*. It is set to *true*, if S is *true* for at least one cycle and R is *false*. It is set to false, if R is *true* for at least one cycle and independently of the value of S. For our component, the logical combination of the unit test Activate sets the flip-flop, the combinations of all other unit tests reset it. The final component is shown in Figure 3.14.

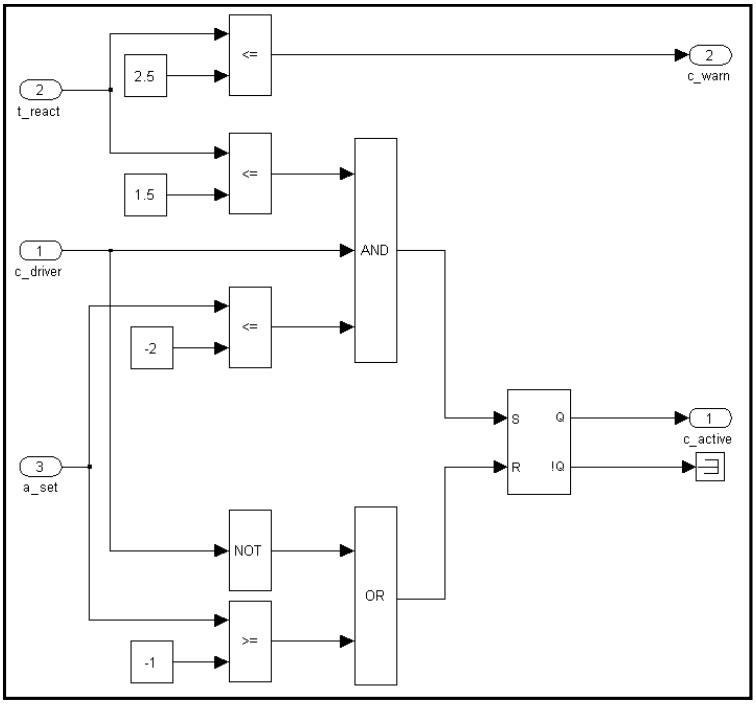


Figure 3.14: The realisation of the component Activation

Finally, the acceptance tests of this component do not pass after its realisation. The reason for this is that the input signals of the component are the output signals of the other three components. c_{driver} is generated by the component Driver Assessment, t_{react} by the component Situation Assessment, and a_{set} by the component Controller.

Hence the acceptance tests can only be fulfilled if these three components are (at least partially) implemented.

3.4.2 Driver Assessment

Already in 1992, developers of the former Daimler-Benz AG found out through tests in the company's driver simulator that most drivers quickly press the brake pedal in emergency situations, but do not reach the maximum brake pedal travel. They can therefore not take advantage of the maximum brake power of their car. This finding led to the development of the Brake Assist System (BAS), which went into series production in 1996 and is today a standard feature of most new cars [187, 188]. It is supposed to help drivers to quickly achieve the maximum brake force in an emergency situation. The activation of the BAS is based on the gradient of the brake pedal travel, i.e. the brake pedal speed. If this brake pedal speed is higher than a threshold value, the system is activated until the driver releases the brake pedal. The BAS does not have any information about the vehicle's surrounding, e.g. an obstacle in front of it, so that it always performs a full emergency braking manoeuvre. As a consequence, the threshold for the brake pedal speed is rather high to prevent annoyance of the driver due to frequent (erroneous) activations.

The component Driver Assessment of our Collision Prevention System should analyse the brake pedal travel and the brake pedal speed to classify a driver's braking manoeuvre into two groups: comfort braking and emergency braking [89, 189]. Comfort braking describes a situation, in which the driver intends to slow down the car in a safe and comfortable way, e.g. when stopping at a red light (anticipatory driving). In other words, the driver does not expect brake support from the CPS in such situations. If, however, the driver presses the brake pedal in a way which indicates an emergency situation, i.e. in a decisive manner, this should be classified as emergency braking. Such a braking manoeuvre does not necessarily have to use maximal brake torque; also a high pedal speed is not required. Instead, emergency braking can be characterized as a progressive increase of the brake force, i.e. the driver is braking in such a way that the derivative of the brake actuation with respect to time is not discontinuous. A common example is a driver who tardily recognises a red light, but still brakes comfortably enough to prevent a cake on the passengers seat from flying into the dashboard.[3]

[3] In fact, this example has often been used to explain to journalists, how they should brake to experience the system.

Concept

With the component Activation, we took the requirements, translated them into a classification tree and derived acceptance and unit tests from it. This approach is not applicable for the component Driver Assessment as we do not know the characteristics of the component's input domain, i.e. the brake pedal travel and brake pedal speed. Thus a partitioning of the input domain into classification and classes is not possible. An alternative would be to build a model, which describes the driver's behaviour in terms of a control system. The driver recognises the situation by his/her human sensors (eyes, ears, etc.), and controls the vehicle by his/her human actuators (feet, legs, etc.). This model would allow to describe the values of brake pedal travel and brake pedal speed for certain situations. However, the effort to create such a model is high, and its results might be not accurate.

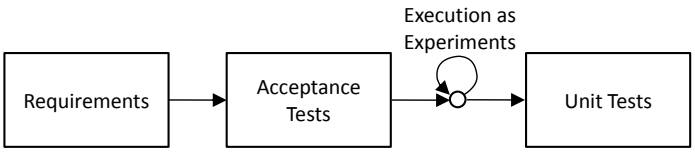


Figure 3.15: Concept for using tests for experimental problems

Therefore we propose the approach shown in Figure 3.15, which considers the component's realisation as an experimental problem. With this, the acceptance tests not only specify the requirements of the system, but also describe the design of experiments. These experiments are used to analyse the characteristics of the input domain for given situations. They serve to identify one or more thresholds for the classification between different classes of behaviour, here between comfort braking manoeuvres and emergency braking manoeuvre. Then a number of unit tests define these thresholds as automated tests, before finally the component is implemented.

Realisation

The starting point of the implementation are the requirements that the system should be activated if the driver performs an emergency braking, and the system should not be activated if he performs a comfort braking. From this, we can directly derive an acceptance tests for the first requirement:

Acceptance Test Emergency braking

Description: The driver drives with the System Vehicle on a test track with a constant speed of $50[kmh^{-1}]$. The driver starts to brake,

when the System Vehicle passes a gate made of two pylons, see the setup of Figure 3.11. The aim of the braking manoeuvre is to stop the car right in front of a stationary obstacle (a foam object). The distance between the gate and the obstacle should be chosen in such a way that the average deceleration is greater than $4[ms^{-2}]$ and smaller than $[8ms^{-2}]$.

$$v_{sys}(t) = 50[kmh^{-1}]$$
$$a_{sys}(t) = 0[ms^{-2}]$$
$$\Delta s_{gate} \cong 15[m]$$

Assertions: The result of the component Driver Assessment, c_{driver}, should be *true*.

Please note that the parameters of this test are not arbitrary, because we do not know whether the driver's braking behaviour is similar at different speeds or not. Furthermore, the position of the gate of pylons will influence the deceleration and thus the brake pedal travel that is necessary to prevent the collision. In fact, if we increase the distance between the gate and the stationary obstacle, we can create another acceptance test for the second requirement.

Acceptance Test Comfort braking

Description: The driver drives with the System Vehicle on a test track with a constant speed of $50[kmh^{-1}]$. The driver starts to brake, when the System Vehicle passes a gate made of two pylons, see the setup of Figure 3.11. The aim of the braking manoeuvre is to stop the car right in front of a stationary obstacle (a foam object). The distance between the gate and the obstacle should be chosen in such a way that the average deceleration is smaller than $4[ms^{-2}]$.

$$v_{sys}(t) = 50[kmh^{-1}]$$
$$a_{sys}(t) = 0[ms^{-2}]$$
$$\Delta s_{gate} \cong 25[m]$$

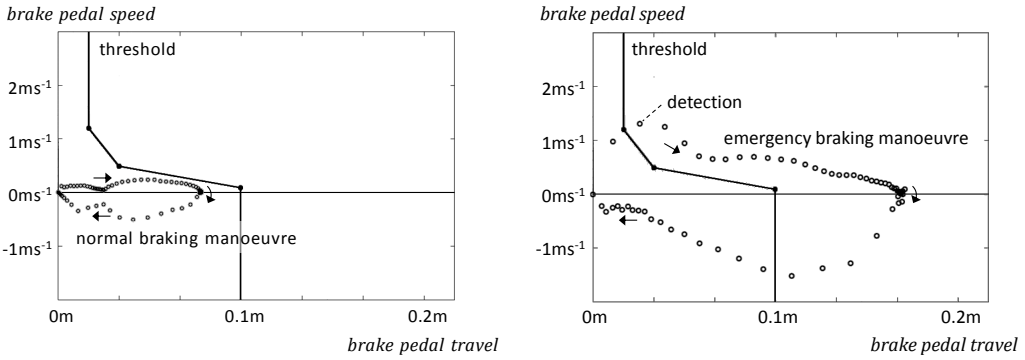
Assertions: The result of the component Driver Assessment, c_{driver}, should be *false*.

This means that the parameter, Δs_{gate}, is specific to each test.

The component should be realised by a threshold which differentiates between comfort and emergency braking manoeuvres. The idea is therefore to do an experiment with a number of drivers. Each driver should execute the tests Comfort braking and Emergency braking a number of times. Figure 3.16 shows two example braking manoeuvres. The x-axis of these plots is defined by the brake pedal travel, the y-axis is defined by the brake pedal speed, i.e. the gradient of the pedal's travel. A braking manoeuvre is described as a trajectory which starts at the origin of the Cartesian coordinate system and then runs clockwise through the first and fourth quadrant. We suppose that a threshold exists which separates both kinds of braking manoeuvres. The trajectory of a comfort braking manoeuvres is then located to the left and below the threshold line, cf. the example in Figure 3.16(a). If it instead exceeds the threshold at some point, it is classified as an emergency braking manoeuvre. An example is shown in Figure 3.16(b), at which the first point exceeding the threshold is marked by the label "detection".

The resulting trajectories of such an experiment with 30 drivers and 3 executions of each test by each driver are displayed with black lines (Comfort braking) and grey lines (Emergency braking) in Figure 3.17. Based on the results, it is possible to specify the interpolation points of the supposed threshold which separates both types of braking manoeuvres as shown by the bold line in Figure 3.17.

With this threshold, we can split the problem into two unit tests. The first tests uses a specific input stimulus, which defines the course of the threshold by a synthetic trajectory of brake pedal travel and brake pedal speed, see Figure 3.18(a). This



(a) Comfort braking manoeuvre (b) Emergency braking manoeuvre

Figure 3.16: Example trajectories for a normal braking manoeuvre and an emergency braking manoeuvre

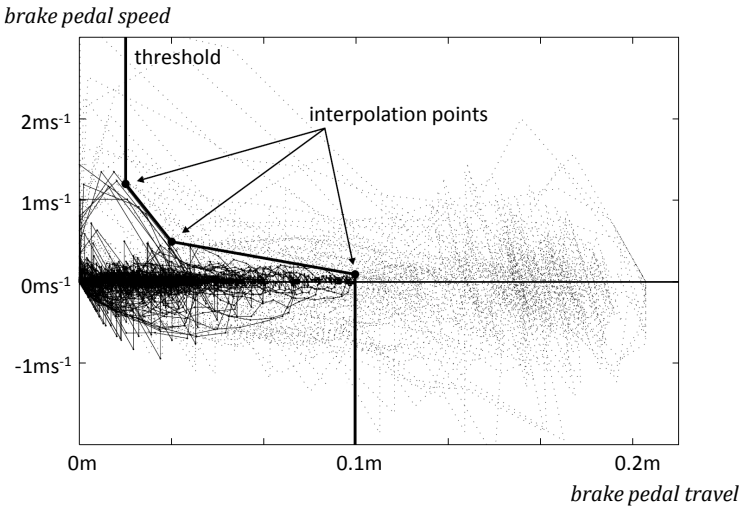


Figure 3.17: Classification of braking manoeuvres into comfort braking (black) and emergency braking (grey)

trajectory is synthetic as it can not be realised in practise. It starts at $t = 0[s]$ with a brake pedal travel of $0.015[m]$ and a brake pedal speed of $3[ms^{-1}]$. Then at $t = 1[s]$ the speed is decreased to $1.2[ms^{-1}]$, but the travel remains the same. Although such a course can not be executed by a driver, it exactly represents the identified threshold with its interpolation points and allows to assert a constant value for c_{driver}.

Unit Test Upper boundary

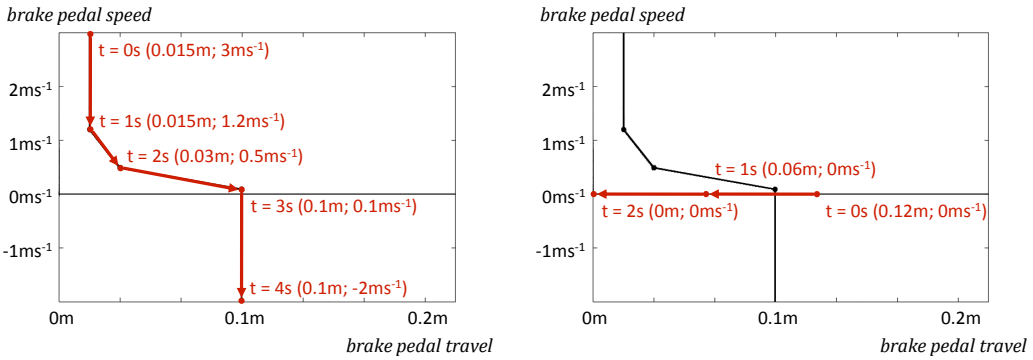
Description: The input values of the component Driver Assessment are stimulated by a synthetic signal, which is defined by the plot shown in Figure 3.18(a)

Assertions: The following values are asserted for the output signals:

$$c_{driver}(t) = true \tag{3.49}$$

The second test applies the same approach, but uses a different input stimulus. For this stimulus, the x-values of the signal are modified by decreasing them by the smallest possible value of the brake pedal travel (defined through the resolution of the signal on the vehicle's data network, e.g. the CAN). Furthermore the output signal is expected to be *false*.

In addition, we can break down the unit tests such as the test Upper boundary into smaller tests, which handle a single segment or point of the threshold. Driven by these tests, the component is realised as a Simulink subsystem, called "Classification".



(a) Threshold (b) Deactivation

Figure 3.18: Defining the threshold by a synthetic trajectory of brake pedal travel and brake pedal speed

The subsystem runs continuously throughout the manoeuvre, and sets c_{driver} to *true* if the trajectory of brake pedal travel and brake pedal speed exceeds the threshold. The functionality is divided into three steps:

1. Firstly, the value of the brake pedal travel is used to determine which segment of the threshold should be used, i.e. to select two interpolation points.

2. Secondly, the y-value of the threshold is calculated with a linear function between these interpolation points. The brake pedal travel is used as the x-value of this linear function.

3. Finally, it is checked whether the value of the brake pedal travel is smaller than the first interpolation point (output value is always *false*) or greater than the third interpolation point (output value is always *true*). Otherwise, the output value is only *true* if the brake pedal speed is greater than the y-value of the linear function.

This realisation has two problems: Firstly, a disturbance of the brake pedal travel or speed might lead to a deactivation of the system, particularly if the trajectory exceeds the threshold only by a small extent. Secondly, the driver will usually feel the support of the system and slightly reduce the pressure on the brake pedal (because of the impression that he has pressed the pedal too much). This might also deactivate the system. Therefore the output signal should only be switched to *false* if the brake pedal travel is reduced by 50 percent of the current trajectory's maximum value. Again we can use a synthetic signal for the braking trajectory, see Figure 3.18(b), to define a unit test. The advantage of such a synthetic signal is that we can easily

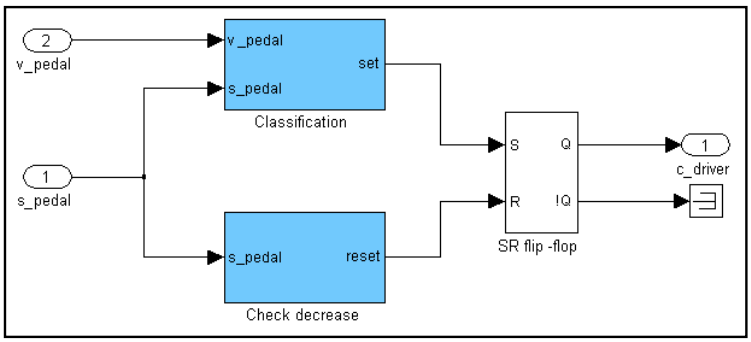


Figure 3.19: Implementation of the component Driver Assessment

define the point in time, at which the output signal should change its value from *true* to *false*. As a result, the component is extended by a SR flip-flop, whose set input is connected to the output signal of the subsystem described above. Another Simulink subsystem, called "Check decrease", then checks the decrease of the brake pedal travel by subtracting the current value from the trajectory's maximum value. If this decrease is larger than a threshold value, the subsystem output signal is set to true. We connect this output to the reset input of the flip-flop. The final component is shown in Figure 3.19.

3.4.3 Situation Assessment

The component Situation Assessment should realise the detection of a potential rear-end collision. It is based on the reaction time that is left to the driver until the System Vehicle reaches the so-called "point of no return". After this point, a collision is considered to be unavoidable by the means of the System Vehicle. For a rear-end collision, the point of no return can be represented as a relative distance between Object and System Vehicle.

In general, the driver has two options to avoid a rear-end collision: steering and braking.[4] Because our system has no information about the space which is or is not available for a steering manoeuvre, the option of steering is disregarded in the following. The reaction time is therefore defined as the time which is left until the driver has to start a braking manoeuvre. This means that the point of no return describes the difference between the braking distance of the System Vehicle and the distance travelled by the Object at the same time. The maximum reaction time then results from a full application of the brake, which is directly dependent on the friction

[4]For other kinds of collisions, especially side-impact collisions, a third option exists in accelerating.

coefficient between the rubber of the vehicle's tyres and the surface of the road.

$$F_{brk} = mg\mu \tag{3.50}$$

$$a_{min} = \frac{F_{brk}}{m} = g\mu \tag{3.51}$$

F_{brk} is the maximal achievable braking force resulting from the friction, g is the acceleration due to gravity, μ is the friction coefficient, m is the mass of the vehicle and a_{min} is the minimal acceleration, i.e. the maximal deceleration, of the vehicle. In the following, we assume a friction coefficient of 1.02 and an acceleration due to gravity of $9.81ms^{-2}$. This corresponds to a minimal acceleration of $-10ms^{-2}$ and a stopping distance of $38.6m$ from a speed of $100kmh^{-1}$, i.e. the common stopping distance of a middle-class car on dry roads.

Concept

The Situation Assessment is realised by solving a mathematical problem. The starting point of our test-driven development process is again the system's requirements. They specify that a certain physical variable should be lower or a higher than a threshold value. Typically, this physical variable has to be calculated from the system's input signals. These input signals might be either measured by sensors or themselves be computed within other electronic control units. For example, the Brake Control Unit measures the System Vehicle's acceleration with an accelerometer, but calculates the System Vehicle's speed from four wheel impulse counters.

For such mathematical calculations, we propose the following approach for the definition of acceptance tests and unit tests, cf. Figure 3.20:

1. Analysis of the problem and derivation of boundary conditions
2. Partitioning into classes which have the same (mathematical) behaviour
3. Definition of acceptance tests for the classes
4. Definition of unit tests in parallel to the derivation of the equations

The process of solving a system of equations by inserting and rearranging mathematical terms is often error-prone, especially if several variables have to be considered. In fact, the rearranging of an equation can be compared to the practise of refactoring in software development processes, see Section 2.4. When refactoring, the functionality of the refactored software must not change; while rearranging an equation, both sides always have to be equal. The aim of the unit tests of the proposed approach is to

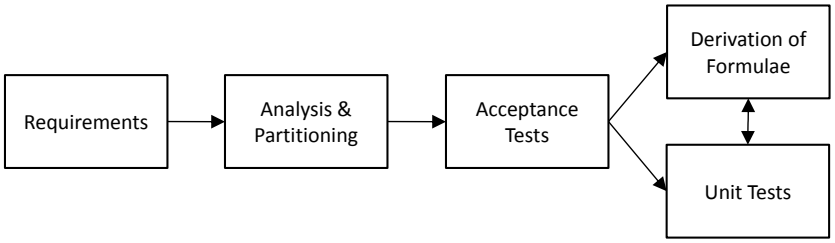


Figure 3.20: Concept for deriving tests for mathematical problems

ensure the correctness of the formulae. They verify that the equation's results are constant during the rearranging by incorporating this mathematical method into the development process.

Realisation

The calculation of the reaction time is defined by a problem space that consists of three translational motions, cf. Figure 3.21.

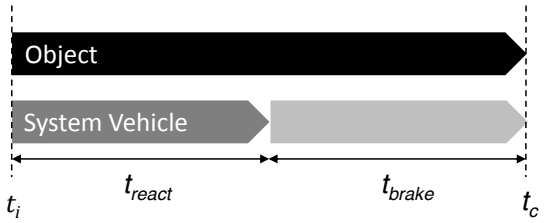


Figure 3.21: Definition of the reaction time

Motion of the System Vehicle during the reaction time t_{react}: We disregard the acceleration of the System Vehicle, i.e. the speed is considered to be constant:

$$a_{sys}([t_i, t_i + t_{react}]) = 0ms^{-2} \tag{3.52}$$

$$v_{sys}([t_i, t_i + t_{react}]) = v_{sys}(t_i) \tag{3.53}$$

t_i denotes the current time.

Motion of the System Vehicle during the time t_{brake}: The acceleration of the System Vehicle is set to the minimal acceleration:

$$a_{sys}([t_i + t_{react}, t_i + t_{react} + t_{brake}]) = a_{min} \tag{3.54}$$

Motion of the Object during the time $t_{brake} + t_{react}$: We assume that the acceleration of the Object is constant during the whole time:

$$a_{obj}([t_i, t_i + t_{react} + t_{brake}]) = a_{obj}(t_i) \tag{3.55}$$

The time t_c describes the point in time, at which the braking manoeuvre is finished and the collision is avoided. It can be defined through the following equations:

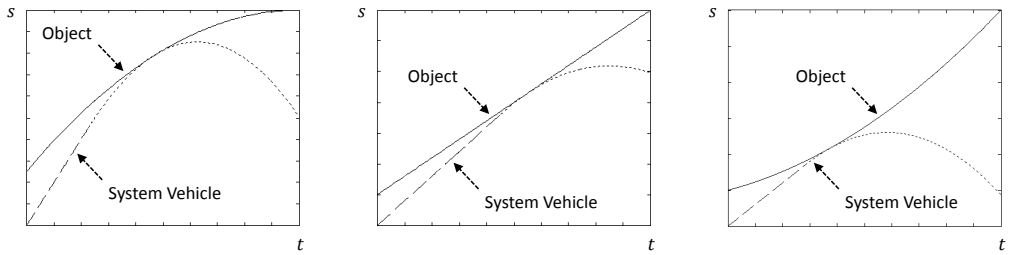
$$s_{obj}(t_c) = s_{sys}(t_c) \tag{3.56}$$

$$v_{obj}(t_c) = v_{sys}(t_c) \tag{3.57}$$

We do not need to specify the relative acceleration at this point, because the System Vehicle's acceleration is always minimal, cf. Equation (3.54).

From a geometrical point of view, these equations describe the point of contact between two curves: the curve of the System Vehicle's motion, which is composed of a linear function and a parabola; and the curve of the Object's motion, whose form depends on the Object's acceleration. For $a_{obj}(t) = 0$ it is a linear function, for $a_{obj}(t) < 0$ and $a_{obj}(t) > 0$ it is a parable.

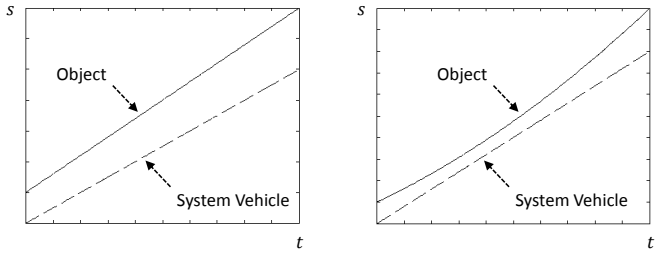
In Figure 3.22 and 3.23 the System Vehicle's motion is shown as a dashed line (linear function) and a dotted line (parable), while the Object's motion is shown as a solid line. There is always a solution for $a_{obj}(t) < 0$, i.e. either the point of contact or an intersection point exists, see Figure 3.22(a). In contrast, this is not necessarily true for $a_{obj}(t) = 0$ and $a_{obj}(t) > 0$ as the Object might be faster than the System Vehicle before a collision would happen. Therefore, two subcases exist for each of these cases, compare Figures 3.22(b) and 3.22(c) as well as 3.23(a) and 3.23(b).



(a) $a_{obj}(t) < 0$ (b) $a_{obj}(t) = 0$ (c) $a_{obj}(t) > 0$

Figure 3.22: Motions of System Vehicle and Object: The trajectories of System Vehicle and Object intersect and a reaction time does exist

With the five subcases, the problem space has been analysed and partitioned into classes with the same behaviour. We continue with the subcases for $a_{obj}(t) = 0$ to



(a) $a_{obj}(t) = 0$ (b) $a_{obj}(t) > 0$

Figure 3.23: Motions of System Vehicle and Object: The trajectories of System Vehicle and Object do not intersect and a reaction time does not exist

explain the further steps of the development approach. The other subcases can be implemented in a similar way.

During the development of the components Activation and Driver Assessment, we have already used a test setup with a foam object. This setup has the advantage that it simplifies our mathematical problem, because the Object is stationary, i.e. $v_{obj}(t) = 0[kmh^{-1}]$ and $s_{obj}(t) = const.$ Moreover a gate of pylons can mark the reaction time as a relative distance to the foam object. This allows the driver to check the value of the reaction time by observing the measurement signal at the moment when the System Vehicle passes the gate.

Acceptance Test Reaction time (stationary object)

Description: The System Vehicle approaches a foam object with a speed of $30[kmh^{-1}]^*$. The reaction time is marked with a gate of pylons, see the setup of Figure 3.11.

$$\Delta s(t = 0[s]) = 100[m]^*$$
$$v_{sys}(t) = 30[kmh^{-1}]^*$$
$$\Delta s_{gate} \cong 3.47[m]^*$$

Assertions: The reaction time should be linear and proportional decreasing with the relative distance between the Object and the System Vehicle. At the gate of pylons, it should have a value of $0[s]$.

The assertion of the test requires that the reaction time is observed while the System Vehicle is approaching the Object. For safety reasons, this can also be done after the execution of the manoeuvre by evaluating the data of a measurement system.

Moreover, the speed and the initial relative distance can be arbitrarily selected to check for different gradients of the reaction time. It has to be noted that the position of the gate and the expected reaction time depend on each other and on the speed of the System Vehicle. If, for example, the gate is moved to $s_{gate} = 7.0[m]$, the expected reaction time would be $0.42[s]$ for $v_{sys}(t) = 30[kmh^{-1}]$, but $0.07[s]$ for $v_{sys}(t) = 40[kmh^{-1}]$.

To continue with the development, we can now focus on the two sections of motion of the System Vehicle, because the Object is not moving in this test. The end of the first section is dependent on the length of the braking manoeuvre. Therefore we start with a unit test for the second section:

Unit Test Full braking (stationary object)

Description: The input signals are set to the following values:

$$\Delta v(t) = -10[ms^{-1}]^* \begin{cases} v_{obj}(t) = 0[ms^{-1}] \\ v_{sys}(t) = 10[ms^{-1}]^* \end{cases}$$

Assertions: The following values are asserted for the output signals:

$$t_{brake}(t) = 1[s]$$

Its expected value is determined in parallel to the derivation of the formulae. Because the test defines $v_{obj}(t) = 0[ms^{-1}]$, t_{brake} is defined as follows:

$$t_{brake}(t_i) = -\frac{v_{sys}(t_i)}{a_{min}} \tag{3.58}$$

With $v_{sys} = 10[ms^{-1}]$ as a test stimulus, we can easily calculate the result of the equation to define the assertion of the above unit test. This means that although the parameters of the test are marked as arbitrary, they are chosen in a way to simplify the calculation of the expected value.

After the test has failed, we are able to realise Equation (3.58) as a new Simulink subsystem, named "Braking time". In addition, more unit tests can be modelled to verify the implementation, e.g. to check for $a_{min} = 0[ms^{-2}]$ or negative speeds. We define such values of the input variables as invalid, because the output of the Simulink subsystem should only be the length of a braking manoeuvre. Therefore we set $t_{brake} = 0[s]$ as a lower limit and $t_{brake} = 10[s]$ as an upper limit. These boundary values also help to define a fix-point scaling for the automatic code generator as well as to prevent a numerical overflow on the target platform.

The next step is the realisation of the reaction time. Using t_{brake}, the point of contact of both vehicles is defined through the following equation:

$$v_{sys}(t_i)t_{react}(t_i) + \frac{1}{2}a_{min}t^2_{brake}(t_i) = \Delta s(t_i) \tag{3.59}$$

The left-hand side defines the motion of the System Vehicle, the right-hand side the motion or, respectively, the relative distance between the Object and the System Vehicle. Only the value of t_{react} is unknown in this equation as we have already derived the formula for t_{brake}. As a consequence, a new unit test can be defined whose expected value is calculated numerically (by hand) with Equation (3.58) and Equation (3.59).

Unit Test Reaction time (stationary object)

Description: The input signals are set to the following values:

$$\Delta s(t = 0[s]) = 10[m]^*$$

$$\Delta v(t) = -10[ms^{-1}]^* \begin{cases} v_{obj}(t) = 0[ms^{-1}] \\ v_{sys}(t) = 10[ms^{-1}]^* \end{cases}$$

Assertions: The following values are asserted for the output signals:

$$t_{react}(t) = 0.5[s]$$

We have two options for the test objective of this test: We can either stimulate only the new Simulink subsystem for the reaction time, called "Reaction time", and specify the time t_{brake} within the test vector. Or we test both Simulink subsystems, i.e. "Braking time" and "Reaction time" together. The disadvantage of the second approach is that an error within the subsystem "Braking time" might influence the results of tests for "Reaction time". For the current test, taking this risk seems unnecessary as the calculation of the braking time is fairly simple. But for the further development of the component, the effort for the creation of the test vector can be substantially decreased, especially when the Object's speed and acceleration are considered.

Finally, the subsystem "Reaction time" is realised by solving Equation (3.59) for t_{react}. The Simulink model of the component Driver Assessment is shown in Figure 3.24. At further development cycles, the component is then extended for moving objects and to consider the other subcases, i.e. $a_{obj}(t) > 0$ and $a_{obj}(t) < 0$. With this, the described unit tests provide a continuous feedback to the developer. They

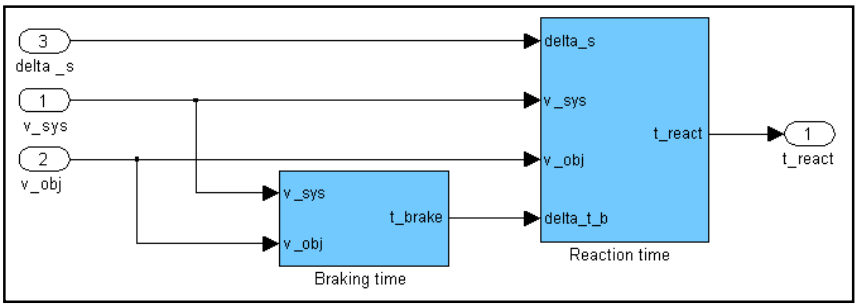


Figure 3.24: Implementation of the component Situation Assessment

show that the equations are correctly derived, because the values which are expected by their assertions remain constant even for more complex formulae. For instance, the expected braking time of the unit test Full braking (stationary object) is always $1[s]$, no matter how the component is extended by further subcases. Thus if the unit tests are carefully selected, i.e. if they cover all identified partitions of the mathematical algorithm, they efficiently prevent errors during the derivation and rearranging of the formulae.

3.5 Summary

This chapter demonstrated the implementation of an automotive safety system to warn and support the driver in situations, which might lead to a rear-end collision. The system was initially defined by a set of requirements, and then analysed for a possible architecture in terms of the physical environment and the integration container. From this architecture, we identified three components, each representing a specific aspect of the design of embedded control systems. The realisation of these components was finally done by different approaches for test-driven development. The approaches were not only based on the methods and tools, which have been introduced in Chapter 2, but also considered the specific objective of the component. With this, we created an efficient system whose specification as well as the way of implementation, e.g. decisions about the design, the execution of experiments, the synthesis of formulae or the choice of parameter values, are documented and verified by automated tests.

Chapter 4

Development of a Longitudinal Vehicle Controller

The components of the Collision Prevention System introduced in Chapter 3, Figure 3.7, can be split into two groups. The first group decides about the activation of the system. Part of this group are the components Driver Assessment, Situation Assessment and Activation. The second group defines the behaviour of the system while it is activated. The only part in this group is the component Controller. It can therefore be considered as the major component defining the behaviour of the system once it has been activated. The controller should control the vehicle in a longitudinal way to support the driver in preventing a rear-end collision. The implementation of such a longitudinal vehicle controller is described in this chapter.

First, a literature review is presented for different approaches for longitudinal vehicle controllers. The component is then realised by a conventional approach as well as a test-driven approach. Furthermore two criteria are introduced which are commonly referred to as the most fundamental design requirements: stability and performance. Finally, the controllers of both approaches are compared by using these criteria and results from the simulation with Simulink are presented.

4.1 Controllers for Longitudinal Control

In the past, a number of different approaches for longitudinal control of vehicle dynamics have been presented. The starting point of these approaches is often the

relationship between the conventional cruise control and the adaptive cruise control (ACC) [190]. This relationship can be described by a diagram, which presents the relative speed between the Object and the System Vehicle (range rate) on the x-axis and the relative distance between them (range) on the y-axis, see Figure 4.1. The

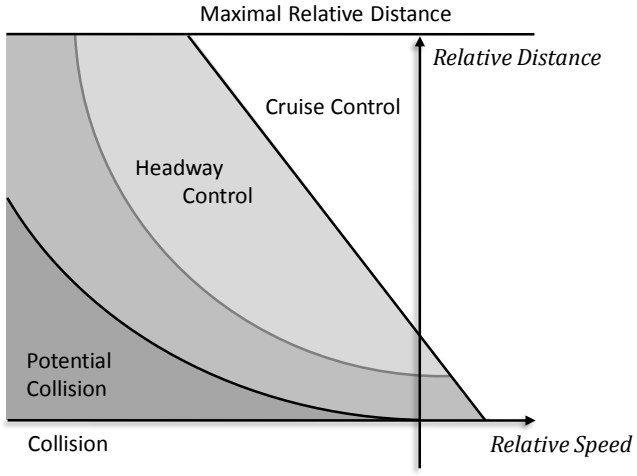


Figure 4.1: Range-vs-Range-Rate diagram (the grey areas indicate the risk for a potential collision)

diagram is therefore called the Range-vs-Range-Rate diagram. The y-axis is limited by the maximal detection range of the sensor used. A linear function divides the diagram into two regions. In the right region, conventional cruise control can be used to keep the vehicle at a constant speed. This region can be considered as a safe area, because there is no risk of a collision between the two vehicles for the configurations within it. The intersection point between the linear function and the y-axis describes the desired relative distance of the ACC system at a steady state. Usually, this distance can be specified by the driver using a potentiometer or menu entry. In the left region, the distance has to be controlled in such a way that the trajectory, composed of range and range rate, leads back to the right region. Otherwise a collision will happen. Two partial parabolas represent the maximal acceleration of the adaptive cruise control system (grey parabola) and the maximal physical acceleration of the System Vehicle (black parabola). To the left of the second parabola, a collision can only be prevented if the Object is changing its predicted motion, e.g. by accelerating to a higher speed or by turning left or right.

In contrast, the Virtual Bumper approach defines two physical regions in front of the System Vehicle [191]. The so-called Region of Interest represents the space which is covered by the sensors, cf. Figure 4.2. When an Object is entering this region,

it is constantly tracked and analysed for its criticality. A subspace of the Region of Interest is the so-called Personal Space, which is positioned close to the System Vehicle. If an Object is detected within the Personal Space, the system computes a force to compress the spring and the damper of a virtual mass-spring-damper system. This force represents a disturbance to the system. It is compensated by a feedback controller, which controls the brake of the System Vehicle to increase the relative distance between the Object and the System Vehicle and, consequently, to decrease the disturbance value.

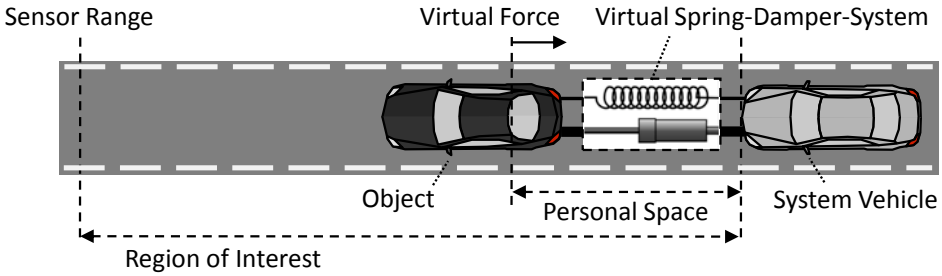


Figure 4.2: Virtual Bumper approach

Another approach is based on fuzzy control, cf. Figure 4.3 [182]. First, empirical data about avoiding a collision is collected from driving manoeuvres with human drivers. This data is converted into a three-dimensional map. The contour lines of this map are reproduced by the creation of so-called fuzzy sets, which transform the input variable of the controller, i.e. the state vector of the control loop, into a domain specific representation (fuzzyfication). Then decisions about the control actions are made by applying a set of rules to the results of the fuzzyfication (inference). Finally, these control actions are transformed to the output of the controller, i.e. the input variable of the control loop (defuzzyfication). With this, the criticality of a situation is determined in relation to the empirical data as a position of a three-dimensional map. Based on this position, one or more rules define how to prevent the collision and result in certain control actions. The major disadvantage of such an approach is that the system's behaviour depends on the characteristics of the three-dimensional maps and the rule base. Thus, even a small change of the requirements might result in a high effort for revising the maps and considering the dependencies between adjacent contour lines.

Similar to fuzzy control, model-predictive control (MPC) uses rules for the synthesis of the controller. These so-called control laws specify a cost function for an optimisation problem. Moreover, the state and output variables of the plant model

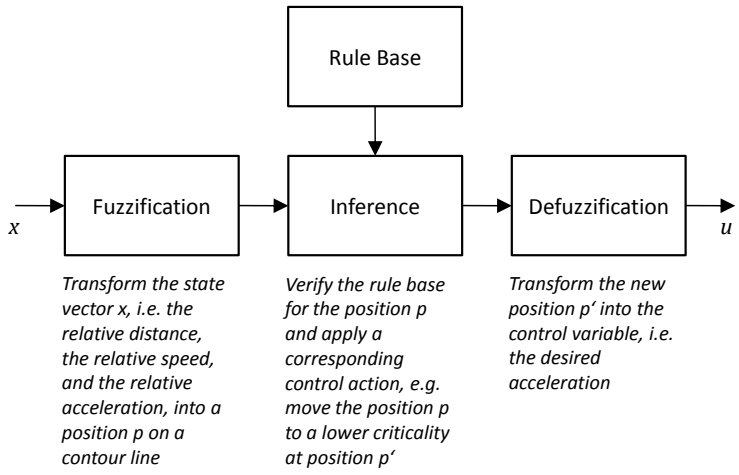


Figure 4.3: Concept of fuzzy control for a longitudinal vehicle controller

are calculated for a certain time horizon. The control variable is derived from this predicted values and the cost function. The process is repeated at every time step with the current state values as the new starting point. This means that the prediction horizon is constantly moved with every time step.[1] For the application of MPC to adaptive cruise control, the problem space is divided into two motions [192]. The *Transitional Manoeuvre* establishes the specified inter-vehicle distance of the ACC system. It starts from a given initial state which is usually defined by a negative relative speed between the Object and the System Vehicle. For this motion, the control laws of model-predictive control aim to prevent the collision while considering the acceleration limits of the system vehicle. Then the *Steady-State Operation* controls the System Vehicle in such a way that the relative distance remains constant. This is typically implemented by a constant time gap spacing policy and the associated control algorithm [193].

Comparing the different approaches, the disadvantage of the Range-vs-Range-Rate approach and the Virtual Bumper approach is that they do not consider the Object's acceleration. In contrast, the fuzzy control approach can be extended by the Object's acceleration, but the realisation of this approach itself requires a high effort as empirical data for a number of driving manoeuvres have to be collected first. Furthermore common mathematical analyses in the frequency or time domain are not possible due to the fuzzy characteristics of the rule base used. Such analyses are possible with the model-predictive control approach, but its disadvantage is that it typically results in

[1] Therefore model-predictive control can be also called receding horizon control.

high computing times. Thus MPC might not be feasible for the ECU of the Collision Prevention System.

4.2 Conventional Approach

As a consequence of the literature review, we regard the presented approaches as not appropriate for the controller to be implemented. The controller will be realised by using a conventional approach, which is based on the state-space representation introduced in Section 3.3.1. In fact, this representation is similar to the state-space representation of the MPC approach described. Two approaches are considered in the following subsections: (i) the dynamics of the closed loop are defined by placing poles with pole placement; (ii) an optimal control approach is realised with LQG.

4.2.1 Synthesis with Pole Placement and LQG Optimisation

The closed-loop system is shown as a block diagram in Figure 3.4 and described by the following equations:

$$\dot{\underline{x}}(t) = A\underline{x}(t) + B\underline{u}(t) + D\underline{d}(t) \tag{4.1}$$

$$\underline{y}(t) = C\underline{x}(t) \tag{4.2}$$

$$\underline{x}(0) = \underline{x}_0. \tag{4.3}$$

with

$$\underline{x}(t) = \begin{pmatrix} \Delta s(t) \\ \Delta v(t) \\ \Delta a(t) \end{pmatrix}. \tag{4.4}$$

The poles or the eigenvalues of the closed-loop system are the zeros of the characteristic polynomial:

$$det[sI - (A - BK)] = det \begin{bmatrix} s & -1 & 0 \\ 0 & s & -1 \\ -\frac{1}{\theta}k_{\Delta s} & -\frac{1}{\theta}k_{\Delta v} & s + \frac{1}{\theta} - \frac{1}{\theta}k_{\Delta a} \end{bmatrix} = 0 \tag{4.5}$$

s denotes the eigenvalues of the matrix $A - BK$. By choosing values for K, the poles of the closed-loop system can be arbitrarily placed. This technique is therefore called pole placement [194, p. 517].

For simplicity, all poles of our closed-loop system will be placed on the same position, λ, with no imaginary part:

$$det[sI - (A - BK)] = (s - \lambda)^3 \tag{4.6}$$

Calculating the determinant and comparing the coefficients results in the following equations for the state feedback:

$$k_{\Delta s} = \theta\lambda^3 \tag{4.7}$$

$$k_{\Delta v} = -3\theta\lambda^2 \tag{4.8}$$

$$k_{\Delta a} = 3\theta\lambda + 1 \tag{4.9}$$

It has to be noted that it is rather uncommon to put all poles on the same position. Usually different positions are preferred as they allow more options for adjusting the controller's behaviour. This is neglected here because the conventional approach should only serve as the basis of comparison with the test-driven approach, but not as a full featured implementation.

As an alternative to pole placement, the feedback matrix can also be derived from a quadratic performance index, i.e. a cost function:

$$J = \int_0^\infty (\underline{x}^T Q \underline{x} + \underline{u}^T R \underline{u}) dt \tag{4.10}$$

The matrices Q and R have to be both quadratic and symmetric. They describe the tradeoff between the regulation performance and the control effort by defining the weighting of the state variables (Q) and the control variable (R). For our closed-loop system, Q is a 3 x 3 matrix, while R is reduced to a scalar as the system has only one control signal. J represents a scalar value which has to be minimised by an optimisation. The feedback control law for such an optimal linear quadratic Gaussian (LQG) regulator is then

$$\underline{u} = -R^{-1} B^T P \underline{x} \tag{4.11}$$

with P being the positive-definite solution of the algebraic matrix Riccati equation [34, p. 600]:

$$A^T P + PA - PBR^{-1}B^T P + Q = 0 \tag{4.12}$$

Typically, this equation is not solved by mathematical analysis, but numerically computed with the help of software, e.g. MATLAB's Control System Toolbox.

4.2.2 Stability and Robustness

Stability is probably the most important criterion for control system design. In general, a system is stable if the output variables exhibit bounded responses to bounded input signals. This is called bounded-input-bounded-output (BIBO) stability. The bounded input signals are usually realised by mathematically well-defined input stimuli such as a unit step or a unit impulse. The system is called asymptotically stable

if the output variables settle at their original steady-state level after the application of an impulse. If they however reach a different, but constant level, or equal a continuous oscillation with a constant, not increasing amplitude, it is said to be marginally stable. Otherwise the system is unstable.

The nominal stability of a system is typically determined by mathematical analysis. Here, nominal means that the analysis uses a (simplified) model with nominal parameters of the real plant. This model is called the nominal plant. A linear time-invariant system is then stable if the poles of the closed-loop system have only negative real parts. With pole placement, this constraint can be directly considered during the design process. In contrast with the LQG approach stability is reached if Q is positive semidefinite and R is positive definite.

Another aspect of stability is robust stability or robustness. It describes how the stability is affected by applying the real plant to the controller. The reason for this is that the real plant often has underlying non-linearities, unknown disturbances and/or parameter uncertainties. The robustness against these influences is typically investigated by methods in the frequency domain, for instance Nyquist or Bode plots. They result in stability margins such as the gain margin and the phase margin, and indicate how far from instability the loop is. The gain margin quantifies the additional gain, by which the open-loop connection of the controller and the nominal plant can be multiplied until it becomes unstable. The phase margin defines the additional phase delay, which can be added to the open-loop connection until it reaches instability. For instance, the LQG approach can result in a gain margin from 0.5 to ∞ and a phase margin of at least $\pm 60^\circ$ [195]. In other words, the closed-loop system remains stable even if the values of the feedback matrix are halved or multiplied with arbitrary large values.

4.2.3 Performance

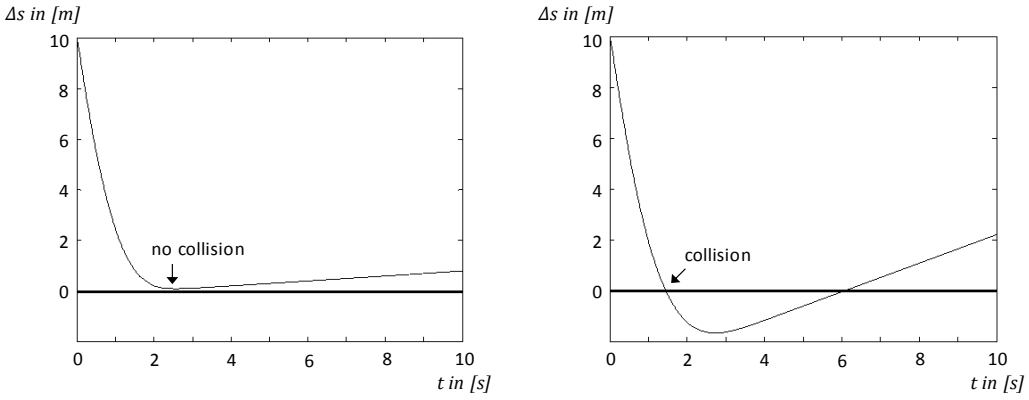
In terms of stability and robustness, the aim of the controller is to compensate for disturbances and plant uncertainties. Another motivation for using a feedback controller is to get consistent performance over a wide range of conditions, or to improve performance compared to the open-loop plant. The performance is usually defined by a number of performance criteria. This section explains the definitions of such criteria and evaluates them for their application to the control loop of the Collision Prevention System.

In the time domain, the way a system responds can be quantified by examining the output response, y(t), when an input or disturbance variable changes from one

steady level to another steady level. Such a change is typically produced by a step function and the corresponding response of y(t) is then called step response. If the height of the step is one, the step function is called unit step. Based on the step response, it is possible to define the following commonly used performance criteria:

Steady-state error: The steady-state error defines the difference between the expected value and the final value of the output response, i.e. the steady-state output. The aim of the controller's design is to minimise this difference. The expected value depends on the variable that is stimulated by the step function. For the reference input of a single-input-single-output (SISO) system, we expect the output response to reach the height of the step function, i.e. the output value follows the desired reference. In contrast, if the unit step is applied to the disturbance value (output disturbance), the controller should compensate for this disturbance, so that the disturbance does not influence the output.

The aim of the Collision Prevention System is not to keep a specific steady-state output value, but only to prevent a potential collision. Thus we use the peak value of the output variable as an alternative criteria. This value represents the relative distance at which a braking manoeuvre is finished. It is positive if the collision was prevented, cf. Figure 4.4(a), and negative if not, cf. Figure 4.4(b). In addition, an arbitrary offset should compensate for inaccuracies of the sensor or perturbances of the plant, so that the collision is even prevented if Δs reaches zero. It is internally added to the value of Δs at the interface of the final software system, i.e. the distance which is received from the radar sensor.



(a) Positive peak value

(b) Negative peak value

Figure 4.4: Relationship between the peak value of the output variable and a collision

Rise time / Settling time: The rise time is the shortest time required for the step response to achieve some specified percentage of the steady-state value. It is usually measured between 10 percent and 90 percent of the steady-state value, because this range is applicable for overshooting and non-overshooting responses. For the stimulation by a disturbance, the rise time can be interpreted as a fall time, i.e. it is measured while the output value falls from 90 percent to 10 percent of the disturbance's amplitude. In contrast, the settling time specifies the time taken for the response to reach and remain within some specified range of its final value. This range is called the allowable tolerance and is normally expressed as a percentage of the step's height. The general difference between both times is that the rise time does not consider the progress of the output signal after it reached the specified percentage, while the settling time is only defined if the signal stays within constant boundaries.

For the CPS, we define the rise time as the time at which the minimal relative distance is reached. It describes how fast the system reacts to a certain initial condition to prevent a collision.

Peak overshoot / Peak time: The peak overshoot specifies the amplitude of the first peak which overshoots the steady-state value, and is normally expressed as a percentage of the steady-state value. The peak time is the time from the initiation of the response to the overshoot.

In terms of the our CPS system, an overshoot of the output value is equal to a collision. Such situations are regarded as *unsafe*. The value of the overshoot might be used to represent the impact energy, but more common for this purpose is the relative speed at the time of the impact.

Another way to describe the performance in the time domain are so-called performance indices. Typically, such indices express the system's performance by integrating the area between the time axis and one or more variables of the closed-loop system. Such a common performance index is the integral of the squared control signal:

$$J = \int_{t_1}^{t_2} u^2(t)dt \tag{4.13}$$

With t_1 specifying the time of application of a step function or a disturbance configuration respectively, and t_2 specifying the time when the minimal relative distance is reached, the value of J states how much effort was used by the controller to prevent the collision. Thus, we use this function to represent the *control effort*. In addition, the minimal value of the control signal, i.e. the maximal deceleration applied by the

controller, can provide further information about the characteristics of the control process, especially because the variable is bounded to certain limits.

4.2.4 Choice of the Parameters

The last step of the conventional approach for the longitudinal vehicle controller is the choice of the parameters, i.e. either the value of λ for the pole placement or the values of Q and R for the LQG optimisation. This design process usually consists of the following steps, which are executed iteratively:

1. Initial values for the parameters are chosen.

   Pole Placement: All poles have to have negative real parts to ensure nominal stability.

   LQG optimisation: Q has to be positive semidefinite and R positive definite. In addition, Bryson's rule can be used [196]:

   $$Q_{ii} = \frac{1}{\text{maximum acceptable value of } \underline{x}_i^2} \tag{4.14}$$

   $$R_{ii} = \frac{1}{\text{maximum acceptable value of } \underline{u}_i^2} \tag{4.15}$$

2. The feedback matrix is derived.

   Pole Placement: The matrix is directly calculated, e.g. for our controller with equations (4.7), (4.8), and (4.9).

   LQG optimisation: The matrix is obtained by either directly solving the Riccati equation (4.12) or by executing a numerical computation of this equation.

3. The closed-loop system is simulated, for instance, as a Simulink model. Particular input stimuli such as a step function are applied to the input, state or disturbance variables.

4. The state and output variables are reviewed with regard to stability and performance criteria.

5. If the requirements for stability and performance are not met, the process is repeated by modifying the parameters and proceeding with step 2.

The modification of the parameters with the last step is typically driven by the experience of the developer accompanied by several evaluation techniques, e.g. frequency methods such as the Nyquist plot.

Pole Placement

In addition to stability and performance criteria, also the functional requirements of the system have to be considered. For our longitudinal vehicle controller, the elements of the feedback matrix, i.e. $k_{\Delta s}$, $k_{\Delta v}$ and $k_{\Delta a}$, do not only influence the system's stability and performance, but also its ability to avoid a collision. For instance, $\lambda = -1$ results in the following feedback vector:

$$K_{\lambda=-1} = \begin{bmatrix} -0.25 & -0.75 & 0.25 \end{bmatrix} \tag{4.16}$$

Due to the logic, which has been realised by the components Activation and Situation Assessment (see Section 3.4.1 and Section 3.4.3), the control signal is only applied to the System Vehicle if

- the relative distance between Object and System Vehicle is greater than or equal to zero, and
- the relative speed between Object and System Vehicle is smaller than zero.[2]

Thus $k_{\Delta s}$ and $k_{\Delta v}$ act against each other. The negative sign of Δv multiplied by $k_{\Delta v}$ decreases the desired acceleration while the positive sign of Δs multiplied by $k_{\Delta s}$ increases the same value. In other words, the larger the relative distance between Object and System Vehicle, the larger is the desired acceleration (and thus, the smaller the desired deceleration) of the controller. On the one hand, this behaviour is in accordance to the CPS' requirements as it avoids that the System Vehicle performs a full braking manoeuvre although the relative distance is still large enough for a more comfortable deceleration. On the other hand, it has to be designed carefully as otherwise the System Vehicle might react too late. For example, if the System Vehicle is approaching a stationary object with $\Delta s(t = 0[s]) = 20[m]$ and $\Delta v(t = 0[s]) = -20[ms^{-1}]$, the initial value of the control variable resulting from Equation (4.16) is $a_{set}(t_i) = -10[ms^{-2}]$. If we move the pole further to the left, the quotient of $k_{\Delta s}$ over $k_{\Delta v}$ changes, see Figure 4.5(a). At $\lambda = -3$ both elements have the same value and as a consequence, the control variable in the same situation would be $a_{set}(t_i) = 0[ms^{-2}]$. As the ideal minimal braking distance is $20[m]$ (assuming $a_{min} = -10[ms^{-2}]$), the example requires an immediate reaction of the system. In Figure 4.5(b), this is achieved by a control signal resulting from $\lambda = -1$ or from $\lambda = -2$, while the control signal resulting from $\lambda \leq -3$ starts with an initial value which is too small in terms of the deceleration.

[2]The CPS might be also activated if the relative speed is initially greater than zero, but decreases due to an Object's acceleration which is smaller than zero. This case is neglected here.

As an alternative, we can define the physically necessary acceleration to avoid the collision by the following equation:

$$a_{phys}(t_i) = a_{obj}(t_i) - \frac{1}{2}\frac{\Delta v(t_i)^2}{\Delta s(t_i)} \tag{4.17}$$

Comparing this acceleration to the control signal, which is derived from the feedback matrix (Equation (3.36)),

$$a_{set}(t_i) = a_{obj}(t_i) - k_{\Delta s}\Delta s(t_i) - k_{\Delta v}\Delta v(t_i) - k_{\Delta a}\Delta a(t_i), \tag{4.18}$$

shows that the quadratic relationship between the relative speed and the relative distance (Equation (4.17)) can not be directly reflected by the coefficients $k_{\Delta s}$ and $k_{\Delta v}$. Using pole placement, a possible solution is the definition of the value of λ dependent on Δv and Δs.

$$\lambda = -\sqrt{\frac{|\Delta v|}{\Delta s}} \tag{4.19}$$

Inserting Equation (4.19) into Equation (4.8) and (4.18) leads to the following control signal, which approximates the quadratic relationship.

$$a_{set}(t_i) = a_{obj}(t_i) - k_{\Delta s}\Delta s(t_i) + 3\theta\frac{|\Delta v|\Delta v}{\Delta s} - k_{\Delta a}\Delta a(t_i). \tag{4.20}$$

The disadvantage of such an approach is the movement of the pole as the relative speed between the Object and the System Vehicle is decreased during the control braking manoeuvre. The pole converges to zero, hence the robustness of the closed-loop system is reduced. We therefore modify the calculation of λ in such a way that

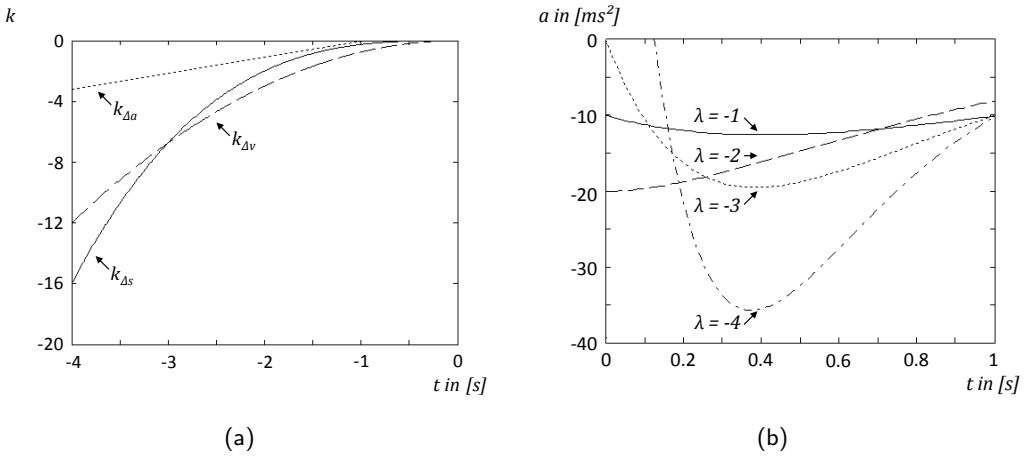


Figure 4.5: Influence of λ on (a) the elements of the feedback matrix K and (b) the control variable u for $x(0) = (20[m]; -20[ms^{-1}]; 0[ms^{-2}])^T$

it is limited to a maximum value of -1.03. This maximum provides sufficient values for the gain margin (2.2) and the phase margin (31°). Furthermore, we calculate the pole resulting from Equation (4.19) only at the beginning of the manoeuvre, i.e. when the CPS is activated. In addition, the pole is moved to the left by a value of 0.5, but limited to a minimum value of -2.0. Both of these values were chosen by simulating the system with various initial conditions, e.g.

$$\underline{x}_{S1}(t = 0[s]) = \begin{pmatrix} 10 & [m] \\ -10 & [ms^{-1}] \\ 0 & [ms^{-2}] \end{pmatrix} \tag{4.21}$$

and evaluating the number of situations, at which the system was able to prevent the collision.

$$l_{pole} = -\sqrt{\frac{|\Delta v(t = t_{act})|}{\Delta s(t = t_{act})}} - 0.5 \tag{4.22}$$

$$\lambda = \begin{cases} -1.03 & \text{if } l_{pole} > -1.03 \\ -2.0 & \text{if } l_{pole} < -2.0 \\ l_{pole} & \text{otherwise} \end{cases} \tag{4.23}$$

t_{act} denotes the time when the CPS is activated. For example, the initial condition in Equation (4.21) leads to a pole at $\lambda = -1.5$ and with equations (4.7), (4.8), and (4.9) to the following feedback matrix:

$$K_{PP} = \begin{bmatrix} -0.84 & -1.69 & -0.13 \end{bmatrix} \tag{4.24}$$

In the following, the feedback controller realised by Equation (4.24) is called SSA-PP controller. SSA is the abbreviation for state-space Approach, PP for pole placement.

LQG Optimisation

In addition to the SSA-PP method, we consider a LQG optimisation as a second variant for the conventional approach. The corresponding controller is then referred to as the SSA-LQG controller. The feedback matrix of this controller is computed as explained above by iteratively setting the elements of Q and R, while observing the system's response to different initial conditions such as Equation (4.21). This process leads to

$$Q = \begin{bmatrix} 0.3 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0.3 \end{bmatrix} \tag{4.25}$$

and

$$R = 0.3, \tag{4.26}$$

which results in the following feedback matrix:

$$K_{LQG} = \begin{bmatrix} -1.00 & -2.64 & -0.82 \end{bmatrix} \tag{4.27}$$

Using Bryson's rule, the choice of Q and R means that we accept larger values for Δs and Δa than for Δv as Q_{11} and Q_{33} are smaller than Q_{22}. This allows us to specify the precedence of the relative speed between Object and System Vehicle in comparison to the relative distance or the relative acceleration.

4.3 Test-Driven Development Approach

In the last section, we have presented two conventional approaches for the longitudinal vehicle controller. In contrast, this section describes a test-driven development approach for the realisation of the same component. Similar to the other components, which have been described in Chapter 3, we first consider the specific aspects of the component's objective to derive a concept for TDD. Then the longitudinal vehicle controller is realised by using this concept.

4.3.1 Concept

The component Controller should show the application of TDD to control system design. Control system design commonly includes the derivation of a number of mathematical formulae to describe the physics of the control system, e.g. by a set of differential equations. Similar to the approach for the mathematical problems, compare Section 3.4.2, we can use this mathematical formulae for the definition of acceptance tests, see Figure 4.6.

The design of a controller is usually done in two phases: analysis and synthesis. Ideally, we want to assign at least one unit test to each of them. The first type of unit test, named *Analysis Tests*, is a direct translation of an acceptance test to the level of the software unit. Analysis Tests specify what the system should do. They contain a plant model, initial conditions, disturbance values as well as an expected behaviour in terms of assertions. Typically the assertions are used in conjunction with one or more performance criteria (cf. Section 4.2.3). The second type of unit tests, named *Design Tests*, then detail how the system is realised. Their parameters are the input stimuli for specific formulae whose results describe the implementation of the controller. Part of the Design Tests can also be the plant model etc. An example

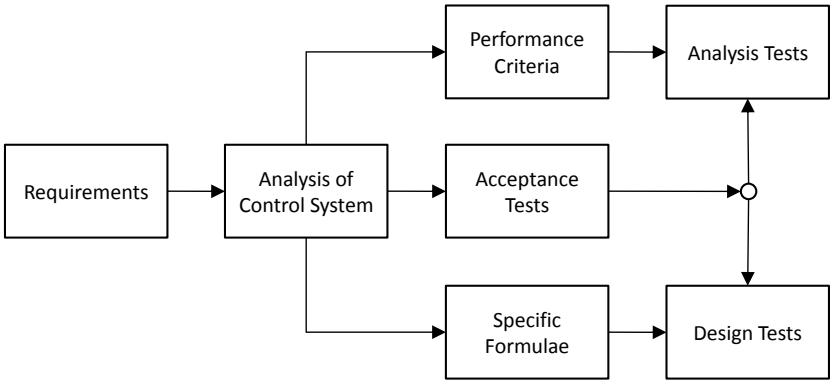


Figure 4.6: Concept for deriving tests for control systems

are the formulae of Ziegler and Nichols. These formulae use the outputs of the plant model, which is stimulated by a step function, to calculate the gain values of a PID controller. Therefore, the Analysis Tests help us to express the requirements of the control system, while the Design Tests drive the implementation of the controller in a systematic and traceable way.

In the following, we will apply this concept in two ways. Firstly, we use the tests to create a basic implementation and document the tuning of the controller. Secondly, this implementation is extended in a manner of incremental design until the system's requirements are fulfilled.

4.3.2 Basic Implementation

The plant model of the CPS has already been described in Section 3.3.1. For the basic implementation of the controller, we use the simplest initial concept by considering the relative acceleration as the only state. This means that the relative speed and the relative distance between Object and System Vehicle are neglected in the model, see Figure 4.7.

$$x(t) = \Delta a(t) = a_{obj}(t) - a_{sys}(t) \tag{4.28}$$

$$y(t) = x(t) \tag{4.29}$$

A steady state is reached if the following condition is fulfilled:

$$\dot{x}(t) = \Delta \dot{a}(t) = 0. \tag{4.30}$$

In addition, a collision is only prevented if the relative speed between the Object and the System Vehicle is zero, i.e.

$$\Delta \dot{v} = \Delta a(t) = 0. \tag{4.31}$$

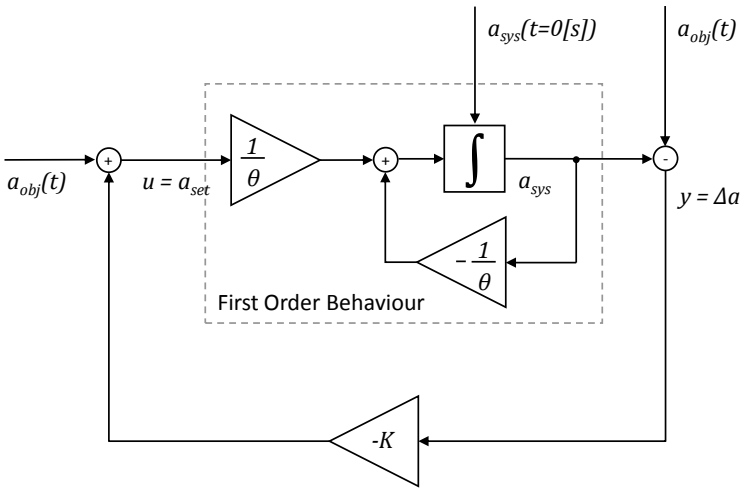


Figure 4.7: Simplified control loop with only one state

In other words, the System Vehicle should brake as much as the Object is braking. From this, we can derive the first acceptance test which specifies a constant motion of both vehicles until the Object starts to brake. It is asserted that the System Vehicle will react with a deceleration of the same value, reaching a steady state two seconds after the beginning of the manoeuvre and with a tolerance of five percent.

Acceptance Test Object is braking

Description: Both the System Vehicle and the Target Simulator (see Section 3.4.1) drive with a constant speed of $30[kmh^{-1}]^*$ and a relative distance of $50[m]^*$. When the Object passes a gate of pylons, it should brake with a constant deceleration of $-1[ms^{-2}]$ until standstill.

$$\Delta s([0, t_g]) = 50[m]^+$$

$$\Delta v([0, t_g]) = 0[kmh^{-1}]^+ \begin{cases} v_{obj}([0, t_g]) = 30[kmh^{-1}]^+ \\ v_{sys}([0, t_g]) = 30[kmh^{-1}]^+ \end{cases}$$

$$a_{obj}([t_g, t_s]) = -1[ms^{-2}]$$

t_g denotes the time when the Object passes the gate, t_s denotes the time when it comes to standstill.

Assertions: The System Vehicle automatically starts to brake when the Object is braking. It should reach the same deceleration after a

settling time of $2[s]$ with an allowed tolerance of 5%.

$$\Delta a(t \geq t_g + 2[s]) \leq 0.05 ms^{-2}$$

It has to be said that from a process' point of view this test does not necessarily have to be defined *after* the analysis of the control loop, because the specified behaviour could also be derived from general thoughts about the system. However, the derivation of the state-space equations helps to express the definition of the test stimuli and the assertions in a more formal way. For the input stimuli, we use a mixture of arbitrary and specific values. The speed of System Vehicle and Object as well as the relative distance are partially arbitrary, because these variables are not part of the plant model and do not directly influence the assertion. The constraint for the speed is that Object and System Vehicle initially have the same speed, i.e. $\Delta v(t = [0]s) = 0[kmh^{-1}]$. The value of the relative distance should be a safe distance between both vehicles, which considers that the test is executed before the controller is implemented due to the test-first approach. In contrast, the Object's acceleration is not arbitrary, but specific to the test by representing a negative unit step. The corresponding assertion describes the performance criterion *settling time*, see Section 4.2.3. The expected value for this time is $2[s]$ with an allowed tolerance of 5%. These values are not part of the system's functional requirements, cf. Section 3.2, but the developer typically specifies them as part of the development process.

Next, we can create a unit test which is similar to this acceptance test, but does not include values for speed or distance as the corresponding states are not a part of our simplified plant model. The test belongs to the group of Analysis Tests.

Unit Test Constant deceleration

Description: The test uses the plant model, which is shown in Figure 4.7, and specifies the initial value of the System Vehicle's acceleration as well as the Object's acceleration.

$$a_{sys}(t = 0[s]) = 0[ms^{-2}]$$
$$a_{obj}(t) = -1[ms^{-2}]$$

Assertions: The following values are asserted for the state variables:

$$\Delta a(t > 2[s]) \leq 0.05[ms^{-2}]$$

In contrast to the unit tests of the previously implemented components, no assertion is defined for the output signals of the test objective, but instead for the output signal

of the plant model. This approach is typical for the testing of closed-loop systems, because the specification of a control system often describes requirements not only for the control variable, $u(t)$, but also for the state variables and the output variables respectively. Therefore, the test uses the initial and the disturbance values of the plant model as its test vector.

The synthesis of our controller should be done by pole placement. With pole placement, the equation for the control signal (Equation (3.39)) is inserted into the first equation of the state-space approach (Equation (3.11)) to derive a formula for the eigenvalues of the matrix $A - BK$.

$$K = 1 + \theta\lambda \tag{4.32}$$

λ is the eigenvalue of the matrix $A - BK$ and equals the pole of the closed-loop system. A stable closed-loop system is obtained when this pole has a negative real part.

The controller is implemented by a constant gain with the value of K. Therefore the exact position of the pole defines not only the behaviour of the system, but also how the controller is realised as λ directly influences the value of K. In fact, it is possible to pass the current unit test with different values of λ, so that we need additional criteria, that is tests, to fully specify the synthesis. One possible way is the specification of additional performance criteria, e.g. through assertions for the rise time and the peak overshoot of the output value to a given input stimulus. Furthermore most signals are subdued by physical restrictions such as the saturation of the system's actuator. By describing these restrictions through one or more tests, we can also isolate a range of valid values for λ.

For the brake system of the CPS, the valid range of the requested acceleration is between $0[ms^{-2}]$ and $-10[ms^{-2}]$. As the minimal value of the Object's acceleration is also $-10[ms^{-2}]$, we can derive the following condition from Equation (3.39):

$$-10[ms^{-2}] < a_{min} - K(a_{min} - a_{sys}(t)) < 0[ms^{-2}] \tag{4.33}$$

a_{min} is the minimal acceleration as defined by Equation (3.51). The maximal value of $a_{sys}(t)$ is $0[ms^{-2}]$, because the driver has to apply the brakes to activate the system[3]. The range of λ is obtained by inserting Equation (4.32) into (4.33) with $\theta = 0.25$.

$$\lambda \in [-4, 0]$$

[3]The case that the driver is braking, but the acceleration of the System Vehicle is still greater than zero, e.g. when driving downhill, is neglected here.

With a second unit test, we can now search for a value within this range that fulfils Equation (4.33), the unit test Constant deceleration, and results in a minimal value for K, i.e. a minimal gain. The test belongs to the group of Design Tests.

Unit Test Choose λ

Description: The test calculates the value of K by using Equation (4.32) and iterating through the range of λ.

$$\lambda \in [-4, 0]$$

Assertions: All of the following assertions have to be *true*.

1. The assertion of the unit test Constant deceleration should be *true*.
2. $-10[ms^{-2}] < a_{min} - K(a_{min} - 0) < 0[ms^{-2}]$
3. The absolute value of K should be minimal.

Its test objective is defined by Equation (4.32) with λ as an input and K as an output value. For two different test objectives we would usually create two test beds, i.e. two Simulink systems. This is neglected here, as on the one hand the test creates a fixture with another test, and on the other hand the testing of these equations is part of the controller design and therefore documented within the test bed as a part of the development process.

The test results in the following position of the pole:

$$\lambda = -1.5. \tag{4.34}$$

The controller can now be implemented with the corresponding value of K. To let also the acceptance tests pass, we set the value of c_{driver} during the development of the component constantly to *true*. With this, the realised controller can be tested without the influence of a driver's braking manoeuvre.

Finally, the system is able to compensate for differences between the Object's and the System Vehicle's deceleration. In contrast, it fails in situations when the relative acceleration is zero, but the System Vehicle is faster than the Object and therefore collides with it. The reason is that the control loop used does not include state variables for the relative speed or the relative distance. Therefore the component is extended in the next section.

4.3.3 Extended Implementation

At the beginning of every development iteration, the first step is to define acceptance tests which translate the requirements into an executable form. To solve the problem described we could use a control loop with two states, the relative acceleration and the relative speed, and proceed similar to the previous iteration. The disadvantage of this approach is that the matrices of the state space equations have to be recalculated and all unit tests must be modified for the additional states. Then this work would be repeated if the third state, the relative distance, will be introduced with another development iteration. Although this approach is an iterative process, the design is not evolved iteratively as the major parts of the design (states, matrices, poles) are changed significantly with every new iteration. This can be compared with the construction of a multi-storey house, where the workers have to rebuild the base whenever a new level is added. As a consequence, either the basic idea of the last iteration is not appropriate for test-driven development, or there is another approach to consider the speed and the distance within the control loop.

The first acceptance test of the controller was that it should compensate for a relative acceleration smaller than zero. With this, a collision is prevented if the relative speed is greater than or equal to zero and the initial relative distance is greater than a certain value (considering the system's reaction time). Thus the corresponding acceptance test only defines an assertion for the relative acceleration, but not for the other states. However, this is not a sufficient assertion if the System Vehicle is faster than an Object which neither accelerates nor decelerates. The new acceptance test should therefore assert that the relative distance must always be greater than zero as this generally expresses the condition for a prevented collision. Moreover the requirement System operation describes a target braking, i.e. to optimally use the relative distance between the Object and the System Vehicle. The controller should brake only as much as necessary to prevent a collision. On the one hand, this allows a maximal reaction time and maximal space for the vehicle which is driving behind the System vehicle. On the other hand, the system will not distract the driver with too high decelerations, e.g. a full braking manoeuvre, although a comfortable distance is still left. In fact, the driver will only feel the support of the system if the requested acceleration is smaller than the acceleration caused by him (as the driver has to apply the brake pedal to activate the system, see Section 3.4.2). Thus we use a second assertion, which defines that the minimum of the relative distance has to be smaller than $1[m]$.

Acceptance Test Faster than Object

Description: The System Vehicle drives with a constant speed of $60[kmh^{-1}]^+$, the Target Simulator drives with a constant speed of $30[kmh^{-1}]^+$. The initial relative distance is $100[m]^+$.

$$\Delta s(t = 0[s]) = 100[m]^+$$

$$\Delta v(t = 0[s]) = -30[kmh^{-1}]^+ \begin{cases} v_{obj}(t) = 30[kmh^{-1}]^+ \\ v_{sys}(t = 0[s]) = 60[kmh^{-1}]^+ \end{cases}$$

Assertions: The System Vehicle automatically starts to brake when the situation is becoming critical. The collision should be prevented, i.e. the relative distance between both vehicles should always be greater than $0[m]$. Moreover, the minimum of the relative distance between both vehicles should always be smaller than $1[m]$.

$$\Delta s(t) > 0[m]$$

$$\Delta(s)(t) < \begin{cases} 1 & \text{if } (\Delta\dot{s}(t) = 0) \wedge (\Delta\ddot{s}(t) \geq 0) \\ \infty & \text{otherwise} \end{cases}$$

We continue with the definition of a unit test for the group of Analysis Tests. The aim of the test is to specify what our system should do (prevent the collision). It is similar to the scenario of the acceptance test Faster than Object and uses the full plant model as described in Section 3.3.1. It therefore defines initial values of the speed of System Vehicle and Object, the relative distance, at which the situation is critical, and a relative acceleration equal to zero.

Unit Test Faster than Object

Description: The test uses the plant model, which is shown in Figure 3.4, and specifies the initial value of the System Vehicle's speed and acceleration, the Object's speed and acceleration as well as the initial relative distance.

$$\Delta s(t = 0[s]) = 15[m]^+$$

$$\Delta v(t = 0[s]) = -10[ms^{-1}]^+ \begin{cases} v_{obj}(t) = 5[ms^{-1}]^+ \\ v_{sys}(t = 0[s]) = 15[ms^{-1}]^+ \end{cases}$$

$$\Delta a(t = 0[s]) = 0[ms^{-2}] \qquad \begin{cases} a_{obj}(t) = 0[ms^{-2}] \\ a_{sys}(t = 0[s]) = 0[ms^{-2}] \end{cases}$$

Assertions: The following values are asserted for the output variable:

$$\Delta s(t) > 0[m]$$

The test fails as the output of the controller, $u(t)$, only depends on the input values of a_{obj} and a_{sys}, which are both zero. We know from the unit test Constant deceleration that the System Vehicle brakes if a_{obj} is stimulated by a negative value. Therefore our current unit test will pass if we manipulate a_{obj} in a similar way, i.e. setting it to $-6[ms^{-2}]$.

Unit Test Manipulated a_{obj}

Description: The test uses the same setup as the unit test Faster than object. In addition, the output signal for a_{obj} is set to $-6[ms^{-2}]$ within the test's model, see Figure 4.8.

$$a_{obj}(t) = -6[ms^{-2}]$$

Note that the Object's acceleration is not modified within the plant model, but only as the test's output signal.

Assertions: The following values are asserted for the state variables:

$$\Delta s(t) > 0[m]$$

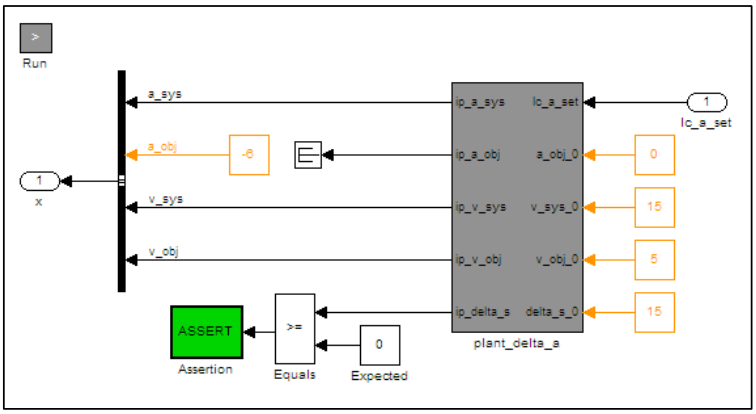


Figure 4.8: Manipulating the Object's acceleration to prevent the collision

Although this test seems to be more like a fake test than like a real test, it is very useful in terms of documenting the implementation idea ("how it is done"). Following

this idea, we modify the definition of the controller as follows:

$$u(t) = a_{obj}(t) - K\Delta a_{ref}(t)) \tag{4.35}$$

$$\Delta a_{ref}(t) = a_{obj}(t) - (a_{sys}(t) + a_{ref}(t)) \tag{4.36}$$

a_{ref} represents a reference value, which is added to the System Vehicle's acceleration in-between the plant model and the controller. Similar to the unit test Manipulated a_{obj} it simulates a negative value of a_{obj} to reach a deceleration of the System Vehicle.

For this reference value, we use a reference model which calculates the physically necessary acceleration to avoid the collision. In fact, such a calculation represents the inversion of the plant model without the first order behaviour. The calculation is done in three steps:

Prediction: At first, the future values of Δs, Δv, v_{obj} and v_{sys} are predicted for a time-horizon of $\theta = 0.25s$, assuming $a_{sys}(t_i) = const$ and $a_{obj}(t_i) = const$. It has to be considered that if the Object is braking, it might happen that the Object will stand still before the end of the time-horizon, as in most cases the Object won't drive backwards after a braking manoeuvre.

$$c_{bak}(v, a) = (a(t_i) < 0) \wedge (v(t_i) < -a(t_i)\theta) \tag{4.37}$$

The Boolean condition $c_{bak}(v_{obj}, a_{obj})$ is *true* if the Object comes to standstill. For this case, the speed of the Object is set to zero at the end of the time-horizon, otherwise the speed of the Object is predicted by the following equation.

$$v_{obj}(t_i + \theta) = \begin{cases} 0 & \text{if } c_{bak}(v_{obj}, a_{obj}) \text{ is } true \\ v_{obj}(t_i) + a_{obj}(t_i)\theta & \text{otherwise} \end{cases} \tag{4.38}$$

A similar formula can be applied to the speed of the System Vehicle:

$$v_{sys}(t_i + \theta) = \begin{cases} 0 & \text{if } c_{bak}(v_{sys}, a_{sys}) \text{ is } true \\ v_{sys}(t_i) + a_{sys}(t_i)\theta & \text{otherwise} \end{cases}. \tag{4.39}$$

The predicted relative speed is then:

$$\Delta v(t_i + \theta) = v_{obj}(t_i + T) - v_{sys}(t_i + \theta). \tag{4.40}$$

If c_{bak} is *true*, we furthermore calculate the distance, which is laid back by the System Vehicle during the time-horizon, i.e.

$$s_{sys}(\theta) = v_{sys}(t_i)\theta + \frac{1}{2}a_{sys}(t_i)\theta^2, \tag{4.41}$$

and by the Object until it reaches stand-still, i.e.

$$s_{obj}(\theta) = -\frac{1}{2}\frac{v_{obj}(t_i)^2}{a_{obj}(t_i)}. \tag{4.42}$$

We subtract these values from each other to get

$$\Delta s(\theta) = s_{obj}(\theta) - s_{sys}(\theta) \tag{4.43}$$

and add it to $\Delta s(t_i)$ to obtain $\Delta s(t_i + \theta)$. If c_{bak} is *false*, we can directly derive the predicted relative distance by using the state values $\Delta s(t_i)$, $\Delta v(t_i)$ and $\Delta a(t_i)$ together with Equation (3.3):

$$\Delta s(t_i + \theta) = \begin{cases} \Delta s(t_i) - \frac{1}{2}\frac{v_{obj}(t_i)^2}{a_{obj}(t_i)} - (v_{sys}(t_i)\theta + \frac{1}{2}a_{sys}(t_i)\theta^2) & \text{if } c_{bak} \text{ is } true \\ \Delta s(t_i) + \Delta v(t_i)\theta + \frac{1}{2}\Delta a(t_i)\theta^2 & \text{otherwise} \end{cases} \tag{4.44}$$

Distinction: If the Object is braking, it might happen that the Object will stand still before the collision takes place[4], as in most cases the Object won't drive backwards after a braking manoeuvre. This is true if the relative speed, $\Delta v(t_i + \theta)$, is greater than or equal to the speed that is necessary to traverse the relative distance, $\Delta s(t_i+\theta)$, during the braking time of the Object, $v_{obj}(t_i+\theta)a_{obj}(t_i)^{-1}$.

$$c_{ca1} = (a_{obj}(t_i) < 0) \wedge (\Delta v(t_i+\theta) \geq \Delta s(t_i+\theta)\frac{a_{obj}(t_i)}{v_{obj}(t_i+\theta)} + \frac{1}{2}v_{obj}(t_i+\theta)) \tag{4.45}$$

If the Object is accelerating, there might be no collision at all. A collision happens if the function of the relative distance has no zero point.

$$c_{ca2} = ((a_{obj}(t_i) \geq 0) \wedge (\Delta v(t_i+\theta) < -\sqrt{2a_{obj}(t_i)\Delta s(t_i+\theta)})) \vee c_{ca1} = false \tag{4.46}$$

The formula to calculate the physically necessary acceleration is the same for an Object, which is accelerating, and an Object, which is decelerating, but does not come to stand-still. Therefore we set c_{ca2} also to *true* if c_{ca1} is *false*.

Calculation: At last, the acceleration is calculated for both cases:

$$a_{ref} = \begin{cases} \frac{v_{sys}(t_i+\theta)^2}{-2\Delta s(t_i+\theta) + v_{obj}(t_i+\theta)^2 a_{obj}(t_i)^{-1}} & \text{if } c_{ca1} \text{ is } true \\ a_{obj}(t_i) - \frac{1}{2}\frac{\Delta v(t_i+\theta)^2}{\Delta s(t_i+\theta)} & \text{if } c_{ca2} \text{ is } true \\ 0 & \text{otherwise} \end{cases} \tag{4.47}$$

[4]Notice the difference between the treatment of the Prediction and of the Distinction.

For the first case, i.e. the Object comes to stand-still, the formula is derived from the point of contact of the stand-still position of the Object and a parabola for the System Vehicle with the predicted speed, $v_{sys}(t_i + \theta)$, as the parabola's gradient at the origin. For the second case, the reference value is built from the Object's acceleration minus the positive acceleration that is necessary to traverse the relative distance, $\Delta s(t_i + \theta)$, with the predicted relative speed, $\Delta v(t_i + \theta)$. If none of the cases are true, the reference value is set to zero.

The derivation and the realisation of these formulae is done similarly to the test-driven development of the component Situation Assessment, which has been described in Section 3.4.3. Therefore the details are omitted here. After the implementation has been completed, the unit test as well as the acceptance test with the name Faster than Object pass. The Simulink model of the component Controller is shown in Figure 4.9.

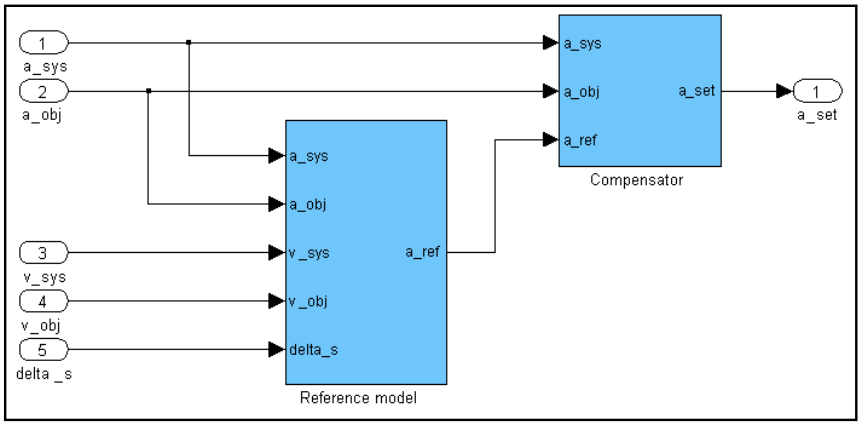


Figure 4.9: Implementation of the component Controller

Finally, the following problem occurred during the verification of the component by system testing. The System Vehicle was driven on a wet road, so that the friction coefficient was less than the expected by the system (see Section 3.4.3). The CPS has no sensors to detect wet roads, therefore all of its components always assume a maximal friction coefficient. Despite this limitation, it is possible to describe the occurred behaviour on wet roads by a unit test:

Unit Test Wet roads

Description: The test uses the same setup as the unit test Faster than object. In addition, the control variable is limited to $-5[ms^{-2}]$ at the input of the plant model and the time constant of the first order

behaviour is doubled.

$$u_{wet}(t) = \begin{cases} u(t) & \text{if } u(t) > -5[ms^{-2}] \\ -5[ms^{-2}] & \text{otherwise} \end{cases}$$

$$\theta_{wet} = 0.50[s]$$

Assertions: The following values are asserted for the output variable:

$$\Delta s(t) > 0[m]$$

The unit test uses the same plant model as the last tests, but limits $u(t)$ to a minimum value of $-5[ms^{-2}]$ and doubles the time constant of the first order behaviour to simulate the conditions of a wet road. As a consequence, the values are not arbitrary, but describe a specific setup for wet roads. The time constant, θ, is only changed in the implementation of the plant model, but not of the controller. The reason is the already mentioned lack of knowledge of the CPS about the friction coefficient.

The test fails because of the controller's progressive strategy, by which the required acceleration is steadily increased until the collision can be avoided. In contrast, students learn a defensive driving style at the driving school. The initial deceleration is chosen higher than actually necessary and then steadily decreased until standstill. The realisation of such a strategy for the controller can be reached by multiplying the value of a_{ref}, which represents the physically necessary acceleration, with a gain factor.

$$\Delta a_{ref}(t) = a_{obj}(t) - (a_{sys}(t) + K_{ref} a_{ref}(t)) \tag{4.48}$$

Figure 4.10(a) shows the graph of a_{cps} and a_{ref} while executing the unit test Wet roads with $K_{ref} = 1$. With this, the required acceleration is always greater than the reference value, i.e. $a_{cps}(t) > a_{ref}(t)$, which leads to a progressive curve of the reference value. In contrast, a_{cps} is always smaller than a_{ref} if K_{ref} is set to a value greater than one. This means that the System Vehicle is decelerating more than predicted by the reference model, but it allows to prevent the collision even in situations with unexpected conditions, e.g. a lesser friction coefficient. As the architecture of the component Controller consists of two units, the basic controller and the reference model, we can define an assertion for an additional unit test to describe this requirement.

Unit Test Defensive braking manoeuvre

Description: The test uses the same setup as the unit test Wet roads.

Assertions: The required acceleration, $a_{cps}(t)$, is smaller than the reference variable, $a_{ref}(t)$, for $t > 0.1[s]$.

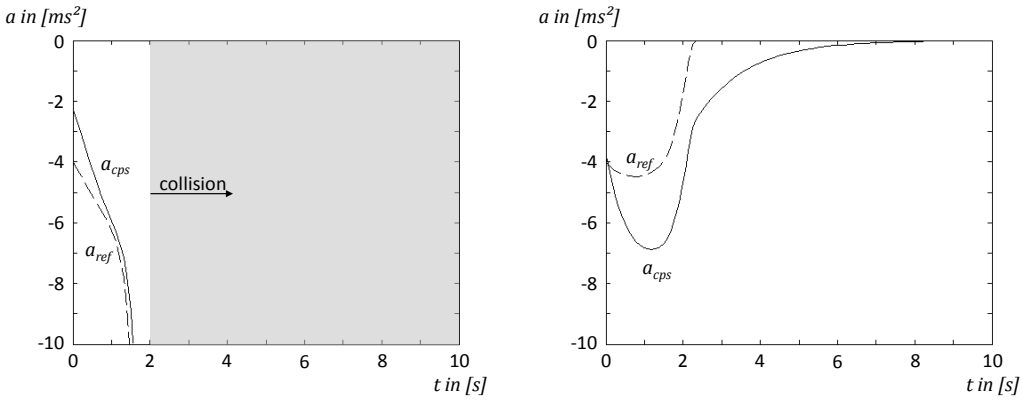
$$a_{cps}(t) \geq a_{ref}(t) \; \forall \; t > 0.1[s]$$

The condition $t > 0.1[s]$ specifies the allowed phase shift of the controller, which is caused by the first order behaviour of the plant.

With $K_{ref} = 1.5$ the requirements of the unit tests Wet roads and Defensive braking are fulfilled, but it impairs the assertion of the unit test Faster than Object as the controller now brakes that much that the minimum of the relative distance is higher than $1[m]$. The reason is the same sign of a_{sys} and a_{ref} in Equation (4.48). If the CPS is activated, the absolute value of a_{sys} is increased and therefore amplifies the control variable. Then the absolute value of the reference value is decreased (because of the prediction, see above) until the control loop is stabilized. Figure 4.10(b) shows the corresponding plot. We can handle this behaviour by adding a gain factor for a_{sys} which is smaller than one.

$$\Delta a_{ref}(t) = a_{obj}(t) - (K_{sys} a_{sys}(t) + K_{ref} a_{ref}(t)) \tag{4.49}$$

These last two tests did not originate from one of the system's requirements, but from a finding during the execution of system tests. Although the system has no



(a) $K_{ref} = 1.0$ (b) $K_{ref} = 1.5$

Figure 4.10: Plots of the control variable and the reference variable for different values of K_{ref}

means to measure the friction coefficient, we are able to describe the expected behaviour by an Analysis Test, i.e. the unit test Wet roads, and then drive the development by a Design Test, i.e. the unit test Defensive braking. With this, the design and the realisation of the controller is incrementally extended. Further development iterations during the PhD project covered the behaviour in additional situations, especially for different manoeuvres of the System Vehicle and the Object and the interaction with a driver's braking manoeuvre. During this process, the structure of the controller remained similar to that of Figure 4.9.

4.4 Comparison

This section focusses on the comparison of the three controllers: the SSA-PP controller, the SSA-LQG controller and the TDD controller. Firstly, we evaluate the stability of the closed-loop system with each controller. Secondly, the performance of these systems is explained by analysing results from the simulation with Simulink. The intention of this comparison is not to prove that one of the approaches realises a controller with a better or worse performance. Instead, it should be shown that the TDD controller leads to comparable results as the SSA-PP and SSA-LQR controller, i.e. test-driven development is suitable for control system design.

4.4.1 Stability and Robustness

The concepts and methods of stability have already been introduced in Section 4.2.2. Nominal stability is reached if all poles of the closed-loop system have only negative real parts. For our system, this position of the poles is accomplished in different ways dependent on the type of the controller:

SSA-PP controller: The dynamic position of the pole is limited to a range of $[-2.0, -1.03]$ (Equation (4.23)), thus the closed-loop system is stable.

SSA-LQG controller: Stability is reached if the matrix Q is positive semidefinite and the matrix R is positive definite. This is fulfilled as the eigenvalues of Q are 0.3, 0.3 and 1.0 (Equation (4.25)) and R is a scalar with a value of 0.3 (Equation (4.26)).

TDD controller: The closed-loop system with the basic implementation of the TDD controller is stable as the pole is placed at -1.5 (Equation (4.34)). In contrast, it is not possible to directly determine the position of the poles for the extended implementation of the TDD controller.

The influence of the real plant is determined by analysing the robustness in terms of the gain margin and the phase margin. Table 4.1 shows the margins' values of the different approaches for the longitudinal vehicle controller. Typically, a gain margin of 2 and a phase margin of 30° provide a sufficient robustness of the system. All three controllers fulfil these requirements. Again, the extended implementation of the TDD controller is not part of the evaluation as its structure does not allow a conventional analysis in the frequency domain.

	SSA-PP	SSA-LQG	TDD (Basic Impl.)
Gain Margin	2.2	18.3	∞
Phase Margin	31°	180°	112°

Table 4.1: Gain and phase margins of the different approaches for the longitudinal vehicle controller

In fact, the closed loop with the extended implementation of the TDD controller is unstable at certain situations, e.g. if all state variables are initially zero. Then a bounded input, such as a unit step to the control variable, results in an unbounded response of the relative distance. The reason for this is that the reference model, which is used by the extended implementation to calculate the physically necessary acceleration, considers only accident-critical situations. Furthermore, also the SSA-PP and SSA-LQG controller lead to an unstable behaviour when the real plant is applied to them, because the only actuator of the CPS is the brake. The control variable is limited to negative values, i.e. the real plant has a non-linear behaviour. Thus the gain margin and the phase margin do not give an adequate characterisation of the robustness for regions of the input domain which result in positive values of the control variable.

Evaluating Robustness by Testing

As a consequence, we introduce another approach for checking the robustness of the closed-loop system by systematically testing the system's response to all possible combinations of input values. Testing stands here for the classical techniques of verification and validation of a software system, see Section 2.3. It is based on the simulation of the closed-loop system as a Simulink model. In terms of the testing techniques, which have been described in Section 2.3.1, the simulation implements a Model-in-the-Loop setup for unit testing.

The total number of tests for the closed-loop system can be represented by an

n-dimensional field. This field theoretically includes an infinite number of tests as each test is defined by a set of real numbers. However in practise, this number is limited by the use of discrete variables in the electronic control unit. Still the number is so high that never all tests can be executed within an acceptable amount of time. For example, if the speed is discretised in steps of $1[kmh^{-1}]$, the combination of the System Vehicle's speed and the Object's speed in a range of $0[kmh^{-1}]$ and $255[kmh^{-1}]$ results in a set of $62,500$ tests. Therefore the classification-tree method was chosen for our approach, because it partitions the input domain of the system into classes that describe the same expected behaviour, cf. Sections 2.3.2. For the testing of the robustness, the classes are defined by the initial values of the state, input and disturbance vectors. The combination of these classes into test cases allows the decomposition of the n-dimensional field into separate regions. We assume that a small number of tests for each of the regions allows us to draw conclusions about the whole region. As explained above, there might be also partitions of the input domain, i.e. regions of the n-dimensional field, which lead to an unstable behaviour, but are not relevant in terms of collision avoidance.

In addition, the plant model can be replaced by a more realistic model than the one used for the state-space representation (Section 3.3.1). The new plant model considers the motions of Object and System Vehicle independently of each other. The speed of both vehicles is limited to only positive values, so that they are not driving backwards after a braking manoeuvre. The output values of all controllers are restricted to a range of $-10[ms^{-2}]$ to $0[ms^{-2}]$ as the input value of the Brake Control Unit is limited to these values. Moreover it is possible to add noise with different levels to the state variables. With this approach, we are able to verify that the SSA-PP, the SSA-LQG and the (extended implementation of the) TDD controller result in a stable and robust behaviour for the relevant regions of the input domain. Example configurations for those regions are shown in the following section.

4.4.2 Performance

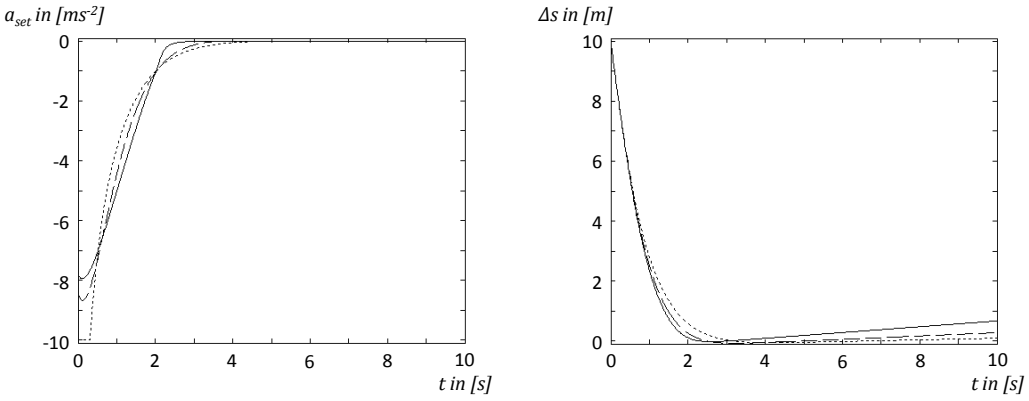
The performance of a control system is commonly evaluated by a set of performance criteria, see Section 4.2.3. In the following, these criteria are applied to the results of three examples. Each example describes a different behaviour of the system and hence represents a member of one of the regions described above.

The first example specifies a linear configuration in the sense that none of the limits

or saturations are reached. It has to be noted that the following configuration

$$\underline{x}_{R1}(t = 0[s]) = \begin{pmatrix} 10 & [m] \\ -10 & [ms^{-1}] \\ 0 & [ms^{-2}] \end{pmatrix} \tag{4.50}$$

can be either linear or non-linear depending on the values of v_{obj} and v_{sys}. For instance, the System Vehicle has to stop to avoid the collision with values of $v_{obj} = 0[ms^{-1}]$ and $v_{sys} = 10[ms^{-1}]$. This stopping manoeuvre is non-linear as the acceleration immediately jumps to zero when the speed reaches zero. In contrast, the configuration is linear if v_{obj} is greater than $0[ms^{-1}]$. For the example, such a linear configuration is realised with $v_{obj} = 10[ms^{-1}]$ and $v_{sys} = 20[ms^{-1}]$. Figure 4.11 shows the corresponding performance. All three controllers provide a safe result, i.e. they prevent the collision.



(a) Control signal (b) Output variable

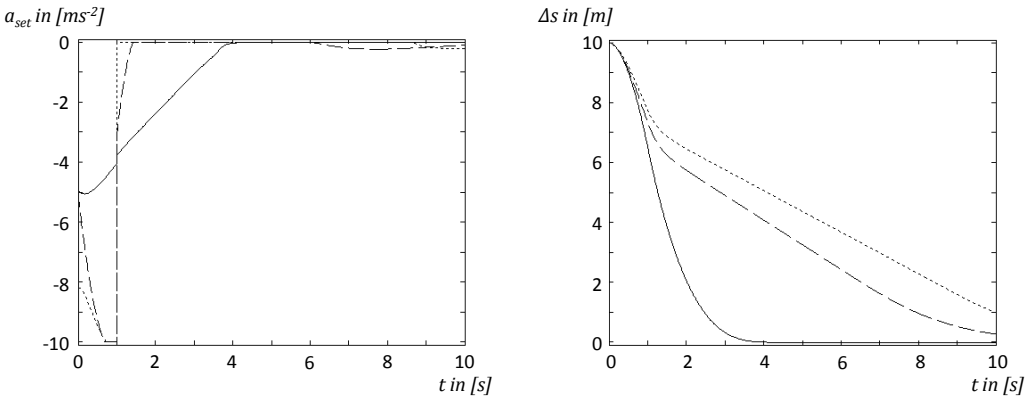
Figure 4.11: Simulation of the TDD controller (solid line), SSA-PP controller (dashed line) and SSA-LQG controller (dotted line) for the initial configuration of Equation (4.50)

The second example describes a non-linear configuration with a negative acceleration of the Object such as the following initial state vector,

$$\underline{x}_{R2}(t = 0[s]) = \begin{pmatrix} 10 & [m] \\ 0 & [ms^{-1}] \\ -10 & [ms^{-2}] \end{pmatrix} \tag{4.51}$$

with $v_{obj} = 10[ms^{-1}]$ and a disturbance value of $a_{obj} = -10[ms^{-2}]$. This configuration is non-linear as the Object decelerates and – similar to the System Vehicle – the Object's acceleration jumps to $a_{obj} = 0[ms^{-2}]$ at $v_{obj} = 0[ms^{-1}]$. Figure 4.12 shows

the results of the example. Again all controllers prevent the collision. The control signal of the TDD controller is only slightly influenced by the non-linearity, while both SSA controllers immediately stop braking and "reconfigure" their reactions to the relation between Δv and Δs, cf. Figure 4.12(a).



(a) Control signal

(b) Output variable

Figure 4.12: Simulation of the TDD controller (solid line), SSA-PP controller (dashed line) and SSA-LQG controller (dotted line) for the initial configuration of Equation (4.51)

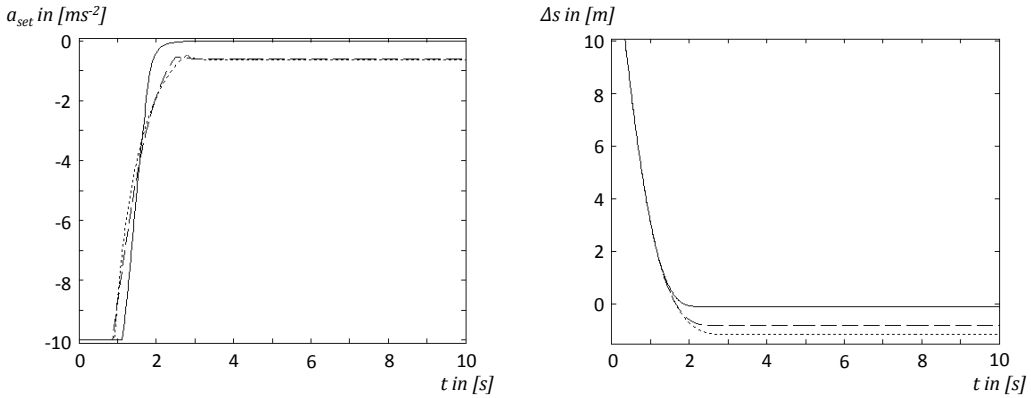
Furthermore, also the limitation of the control variable, $u(t)$, leads to non-linear configurations. With the following initial configuration,

$$\underline{x}_{R3}(t = 0[s]) = \begin{pmatrix} 15 & [m] \\ -15 & [ms^{-1}] \\ 0 & [ms^{-2}] \end{pmatrix} \tag{4.52}$$

the three controllers reach the lower limit of $u(t)$, cf. Figure 4.13. When we look at the curves of $u(t)$, all controllers start with the maximal deceleration, but hold it for different time spans. Only the TDD controller prevents the collision, while the SSA-PP controller as well SSA-LQG controller result in a negative minimal relative distance between the Object and the System Vehicle.

4.4.3 Conclusions

For the comparison of the three examples, Table 4.2 shows the values of the performance criteria for the peak value, the rise time, the performance index and the minimum value of the control signal. Only the TDD controller is able to prevent the collision in all example configurations, i.e. it has always a positive peak value.



(a) Control signal

(b) Output variable

Figure 4.13: Simulation of the TDD controller (solid line), SSA-PP controller (dashed line) and SSA-LQG controller (dotted line) for the initial configuration of Equation (4.52)

The SSA-PP controller and SSA-LQG controller fail at the last example, because the saturation of the control signal has not been considered during the design of these controllers. In general, it is possible to regard this non-linear behaviour also with a conventional approach, but this increases the complexity of such approaches. Thus the better performance of the TDD controller does not directly result from the test-driven development process. Instead, the advantage of test-driven development is that it allows us to analyse the system's behaviour with different plant models. We can specify a linear plant model within a test case, derive a controller from it, and then create another test case that takes the saturation of the control signal into account. Moreover, the concrete description of the system's requirements by translating them into tests helps us to focus on what the controller should do. The TDD controller was therefore designed to perform a target braking manoeuvre. As a consequence, it also achieves the smallest performance index and the largest value of the applied control signal, i.e. the smallest applied deceleration, in all examples.

Finally, the discussion of the example results is supported by the diagrams shown in Figure 4.14. The first diagram in Figure 4.14(a) represents a range-vs-range-rate diagram. The three controllers were simulated for values of Δs from $0[m]$ to $100[m]$ and values of Δv from $-50[ms^{-1}]$ to $0[ms^{-1}]$. For a_{obj}, two distinct values were chosen: $0[ms^{-2}]$ and $-10[ms^{-2}]$. The plotted curves divide the configurations into two areas for each controller. In the bottom-left area, a collision was not prevented, in the top-right area, no collision occured. With this, all curves have a shape similar to the ideal curve, cf. Figure 4.1, but the TDD controller results in the smallest bottom-

	SSA-PP	SSA-LQG	TDD
Example 1	0.00	0.02	0.08
Example 2	0.10	0.10	0.14
Example 3	−0.70	−1.05	0.02

(a) Peak value

	SSA-PP	SSA-LQG	TDD
Example 1	3.2	4.1	2.6
Example 2	14.0	16.9	3.3
Example 3	2.5	2.8	2.2

(b) Rise time

	SSA-PP	SSA-LQG	TDD
Example 1	59.4	63.5	58.8
Example 2	78.2	87.0	45.7
Example 3	136.1	133.6	139

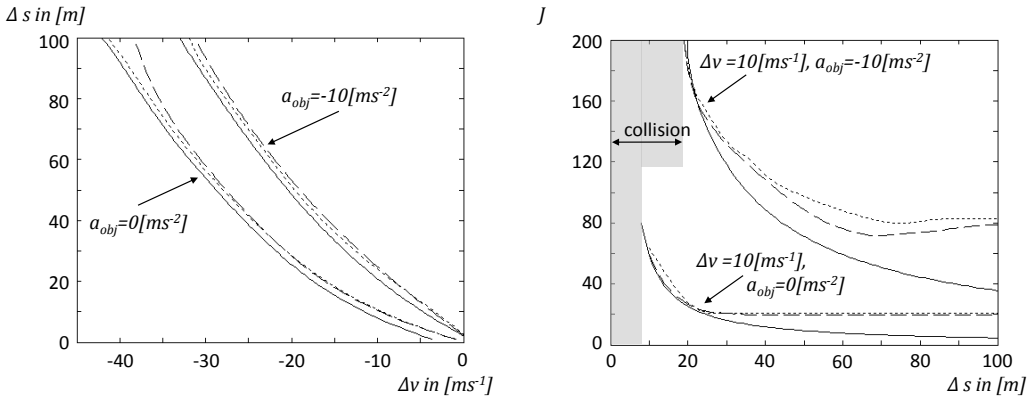
(c) Performance index

	SSA-PP	SSA-LQG	TDD
Example 1	−10.0	−8.7	−7.9
Example 2	−10.0	−10.0	−6.8
Example 3	−10.0	−10.0	−10.0

(d) Minimum of the control signal

Table 4.2: Comparison of performance criteria for the three examples

left area, i.e. the smallest number of configuration with collisions. a_{obj} influences the curves in moving them along the y-axis to the top and along the x-axis to the right without changing their main characteristics. In other words, the number of configurations at which a collision can be prevented is decreased. This is in accordance with the expected physical behaviour.



(a) Range-vs-Range-Rate

(b) Performance Index

Figure 4.14: Simulation of the TDD controller (solid line), SSA-PP controller (dashed line) and SSA-LQG controller (dotted line) for different values of Δv and Δs

The second diagram, see Figure 4.14(b), depicts the performance indices of the three controllers as a function of the relative distance for a given relative speed, i.e. $\Delta v = -10[ms^{-1}]$ and $a_{obj} = 0[ms^{-2}]$. In other words, it represents the performance index for a vertical line through the range-vs-range-rate diagram. All performance indices have similar values at low relative distances as only a full braking manoeuvre results in a prevented collision. With the increase of the initial relative distance, the performance index of the TDD controller declines faster than the performance of the other controllers. Moreover the values of both SSA-controllers settle at certain levels, i.e. the control effort remains the same no matter how far the Object is away. This confirms the results of the three examples, at which the TDD controller results in the smallest performance index and the minimal input value. The influence of a negative value of a_{obj} is again a translation of the curves to the top right corner without changing its main characteristics.

4.5 Summary

This chapter described two approaches for the realisation of a longitudinal vehicle controller, which represents the main component of the Collision Prevention System. The first approach used conventional methods of control system design, while the second approach showed the application of test-driven development to such a component. The results of both approaches were evaluated by comparing the implemented controllers in terms of stability and performance. From this, we draw the following conclusions:

- Both the realisation of the controller with conventional approaches and with test-driven development led to a sufficient performance in order to avoid a collision. The maximal number of collisions prevented is higher with the TDD controller. Test-driven development allows to precisely describe the system's requirements by executable tests. This helps the developer to explore the requirements, the dynamics of the derived plant model, and the difference between the plant model and the real plant. In addition, the complete set of tests ensure that no functionality is broken during the iterative development process. Thus the developer is able to focus on a single requirement, e.g. to avoid the collision, and then modify the resulting controller for another requirement, e.g. to perform a target braking manoeuvre, without violating the first requirement.

- It was not possible to analyse the stability and robustness for the TDD controller by using conventional methods of control theory. In general, it is possible to

derive a controller with test-driven development in such a way that conventional methods are applicable. However, the iterative process of TDD and its style of implementing a feature by making a test pass might circumvent these methods more often. We therefore presented a testing technique to evaluate the stability and robustness by stimulating the system's input and verify the output values. Moreover, the number of tests was reduced by partitioning the input domain into classes with the same behaviour.

To summarise, the aim of test-driven development is not to replace the conventional approaches of control system design. Instead, TDD complements these approaches by a test-centric process which encourages the solutions of the two main development issues: what the system should do, and how the system should be implemented.

Chapter 5

Evaluation of the Collision Prevention System

The focus of this chapter is the analysis and evaluation of the automotive safety system realised in the previous two chapters. The first section describes methods and results of trials on a driving simulator. The aim of these trials was to prove the effectiveness of the Collision Prevention System, i.e. to show that it really helps the driver to prevent a collision. The second section then focusses on experiments with prototype vehicles. These experiments represent the verification of the CPS in terms of system testing and acceptance testing. Finally the results are summarised.

5.1 Trials on a Driving Simulator

5.1.1 Methods

The driving simulator consists of a complex architecture which can be divided into two basic parts: the hydraulic cylinder-powered dome and the simulation system. The driver sits inside the dome within a real car, whose user interfaces such as the steering are disconnected from their physical counterparts and instead connected to the simulation system. While driving, he/she looks to a 180 degree screen which is mounted at the inner wall of the dome showing a visual representation of real driving scenarios. These driving scenarios are created by the simulation system with a cluster of several computers. Like in a computer game, the system is able to render a high

number of polygons with textures for different types of road surfaces, houses, road signs, pedestrians and other vehicles. Furthermore the dome is moved and rotated in lateral and longitudinal directions to simulate the vehicle's motion. With this, the driver gets the approximate impression of driving a real car in a real situation.

Figure has been removed due to Copyright restrictions.

For the trials of the CPS, over 100 subjects completed a 40-minute test-drive in the simulator. The subjects were not involved with the development project or any associated embedded system. To reflect the typical age distribution of Mercedes' drivers, also pensioners were invited to participate as test drivers.

Three accident-critical scenarios occurred during each test drive:

1. The System Vehicle is driving in a convoy on a highway at $130[kmh^{-1}]$. At a certain point, the Object in front of the System Vehicle initially brakes moderately, then suddenly starts a full braking manoeuvre until a complete stop.

2. Same as scenario 1, but on a country road at $80[kmh^{-1}]$.

3. The System Vehicle is driving on a highway at $130[kmh^{-1}]$ on the left lane and passes a slower convoy on the right lane. Suddenly, an Object pulls out of the convoy into the lane of the System Vehicle.

The sequence of the three scenarios has been randomly rotated for the different test drivers, so that a statistical equipartition of drivers and scenarios could be reached. Such a rotation was necessary to exclude psychological effects such as an increased awareness of the driver after the first accident-critical situation of one of the three scenarios.

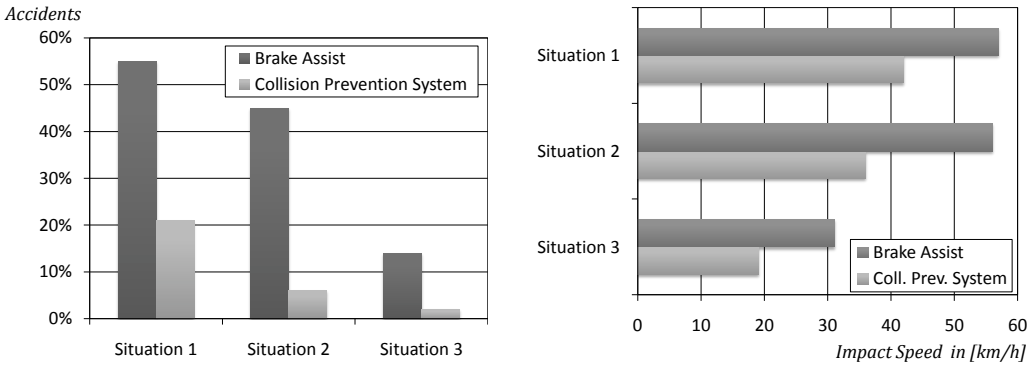
The test drivers were split into two groups. The first group was driving a conventional car with the Brake Assist enabled, but the Collision Prevention System disabled. The other group had both the BA and the CPS activated, thus got a visual and acoustical warning preliminary to an imminent collision, and if they applied the brake pedal, brake support by means of the realised TDD controller.

In terms of the testing techniques which are described in Section 2.3.1, the driving simulator realises a Hardware-in-the-Loop environment. It can be used for either System Testing or Acceptance Testing, compare Figure 2.11. The trials realise a special scenario of Acceptance Testing with two roles. The first role is the test driver, which represents a customer. He/she drives the car and tests the system as its designated user. The second role is the developer, who organises and evaluates the

trials after their execution. His/her goal is to prove the effectiveness of the system by comparing the accident rates of the trials with the new system and without it on a statistical basis. Thus, this approach does not directly verify the system's functional requirements, but analyses the system's impact on its use cases.

5.1.2 Results

The average accident rate of the first group over all scenarios was 38% of 150 accident-critical situations. This number could be reduced by the second group to just 10% of 150 accident-critical situations. This means that the CPS results in 74% fewer accidents. The biggest reduction could be achieved in the second scenario. Here 45% of the first group had a rear-end collision in 50 accident-critical situations, in contrast to only 6% of the second group in 50 accident-critical situations. The accident rates of each scenario and group are shown in Figure 5.1(a). If still a collision happened, the collision speed was decreased from an average of 57$[kmh^{-1}]$ to 39$[kmh^{-1}]$. Again, the second scenario led to the highest benefit of the CPS as the collision speed was reduced by 20$[kmh^{-1}]$, cf. Figure 5.1(b).



(a) Accident rate

(b) Collision speed

Figure 5.1: Results from the Driving Simulator

5.2 Experiments with Prototype Vehicles

5.2.1 Introduction

Besides the simulation with Simulink and the trials in the driving simulator, tests with real vehicles have been very important for the development of the Collision Prevention System. Car makers typically release a new system such as the CPS

together with a new model of a vehicle. Therefore the system can not be tested with the final product during most of the time of its development. Instead the tests rely on prototype vehicles. These vehicles are either based on the model's predecessor or a prototype of the new vehicle. The predecessor car is usually equipped only with some of the new components, which are necessary to execute the system. This kind of vehicle is called an *Aggregate Carrier*. In contrast, the real *Prototype Vehicle* consists of all new electronical and mechanical components that are designated for the new model. Thus both kinds of vehicles provide a platform for target testing, but the Prototype Vehicle provides a higher integration level of the system in terms of the complete vehicle.

Figure has been removed due to Copyright restrictions.

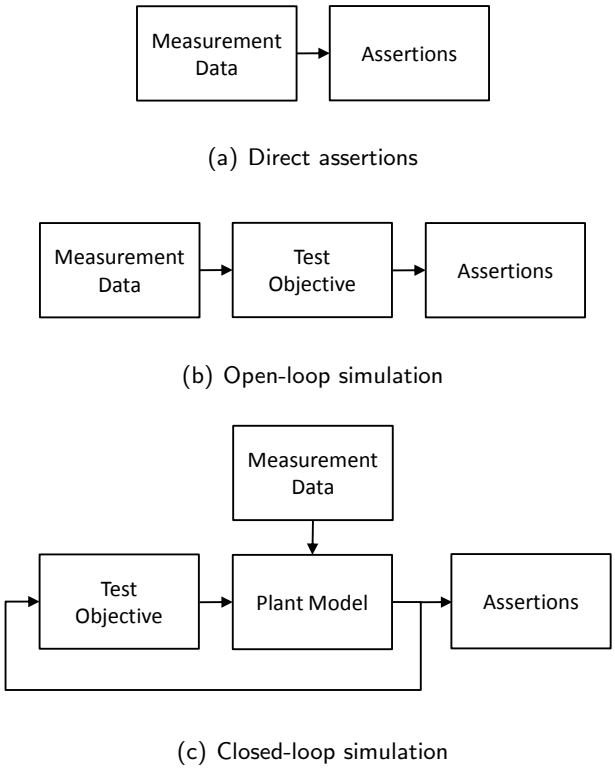
5.2.2 System Testing on Test Tracks

During the development and prior to every release, system testing was accomplished on test tracks. For this, a comprehensive catalogue of about 300 driving manoeuvres was derived with the help of the Classification-Tree Method. The aim of these manoeuvres was the verification of the system's requirements. This also included situations where the System Vehicle had to collide with the Object. Therefore mock objects were used similar to the acceptance tests which are described in Chapter 3. During the manoeuvres, all internal data of the ECU and signals on its external interface were recorded (overall more than 1,500 channels).

The evaluation of the system tests followed a two-step process. The first step was that the test driver marked each manoeuvre as "passed" or "failed" directly after its execution. For this assessment, the manoeuvre catalogue included a description of the expected behaviour, which should be experienced by the driver. A typical example is:

> The optical warning is activated by an illuminated red triangle within the cluster instrument.

After the accomplishment of all manoeuvres, the measurement data was reviewed in the second step. This review was done manually by the developer as well as automatically by the assertions. For this, the data is read into Simulink so that the channels are available as usual Simulink signals. These signals are connected to a set of assertions for every manoeuvre. Figure 5.2(a) shows the specific architecture for this approach.



(a) Direct assertions

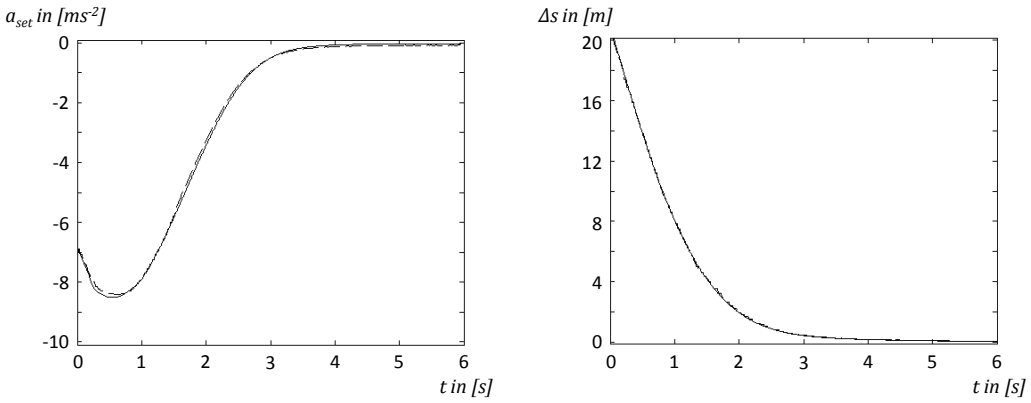
(b) Open-loop simulation

(c) Closed-loop simulation

Figure 5.2: Different approaches for the verification and reproduction of measurement data

A similar approach is used when a driving manoeuvre has failed. Again, the data is read into Simulink, but now as a stimulation for the test objective, cf. Figure 5.2(b). The test objective can be the whole (model-based) system, or smaller components down to the smallest entity, i.e. a unit. Ideally, a data acquisition point is set for each signal of the interface between two units. Hence one measurement channel can be directly connected to the test objective in Simulink. With this open-loop simulation, the developer can observe the outputs of either the test objective or every other Simulink block that is part of the test objective. Moreover assertions should be defined which reflect the cause of the error. In other words, a failing unit test is created and then the test objective is modified to make this test pass. The disadvantage of this approach is that invariant measurement data are used for the stimulation of the test objective. The inputs do not change even if the test objective is interacting with the actuators of the system differently than during the performed manoeuvre. For instance, if the System Vehicle collided with a mock object at the test track, it will always collide during the open-loop simulation no matter how the control signal is increased.

Therefore we propose another setup, see Figure 5.2(c). Here, the measurement data is used only at a single point in time to define the initial values of a plant model. The further simulation is implemented as a closed-loop between the test objective and the plant model. In addition, it might be necessary to modify the plant model in such a way that it has the same behaviour as the measurement data. Hence, we simulate the closed loop, compare the output as well as the control variable against the measured signals, and change the plant model until the simulated variables reproduce the real data. An example is shown in Figure 5.3, where the simulated input variable as well as the output variable nearly match the measurement data.



(a) Control signal (b) Output variable

Figure 5.3: Example of the comparison between simulation results (solid line) and measurement data from a prototype vehicle (dashed line) for a stationary object with $\Delta v = 50[kmh^{-1}]$ and $\Delta s = 20[m]$

5.2.3 Acceptance Testing in Real Life

In parallel to the development of the next software release, the current release was installed on a fleet of vehicles for acceptance testing in real life. Overall, the fleet consisted of 28 prototype vehicles on different continents, including a taxi in Stuttgart, Germany. The vehicles were driven by about 200 drivers for nearly 500.000km. Each vehicle was equipped with a measurement system, which has been installed in the trunk and continuously recorded over 50 data acquisition channels. The data channels included internal data of the RADAR Control Unit, e.g. the calculated reaction time, but also data of other ECUs such as the distance from the RADAR sensor or the yaw rate from the Brake Control Unit. Part of the measurement system was also a video camera, whose picture was automatically saved in a relevant situation. The

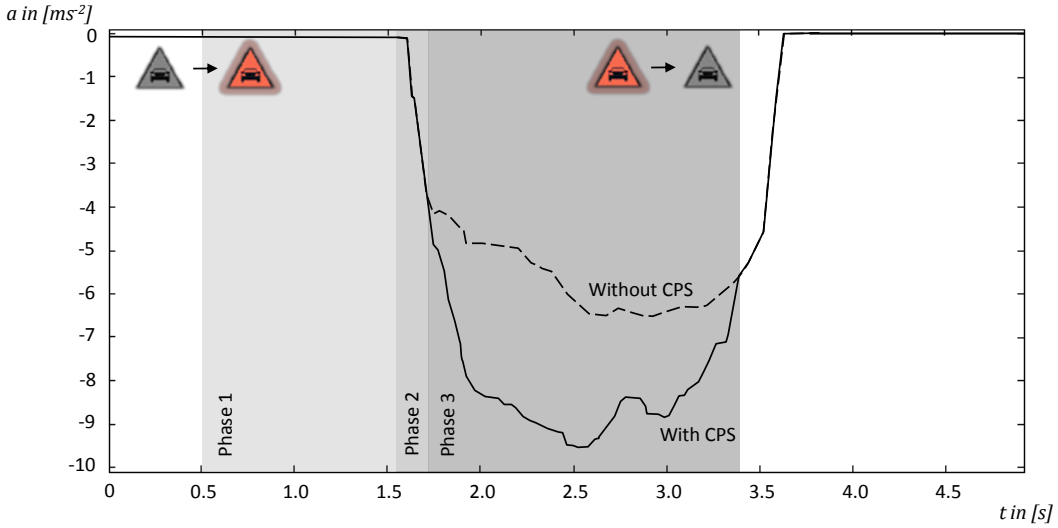
relevance of a situation was determined according to several criteria. For instance, if the driver pressed the brake pedal for more than 50% of its maximum travel, a trigger was started to save the last 20 seconds and the following 10 seconds of the situation to a measurement file. In addition, the driver could manually set a trigger by pressing a button.

Every time one of the test vehicles returned to its base station, the data of the measurement system was copied to a dedicated file server. Then a set of MATLAB scripts automatically processed the data and inserted a summary into a database. This database was periodically reviewed by the system's developers to analyse valid and invalid situations. Whether a situation was valid or invalid could be evaluated through evaluating the video clip in combination with the recorded data. Furthermore, invalid situations could be divided into two groups: false positives and false negatives. We speak of a false positive situation when the system was activated although the requirements were not met for an activation. For instance, the system was braking, but no Object was in front of the System Vehicle. In contrast, a false negative situation describes a missing activation, i.e. the system was not activated although the requirements were fulfilled.

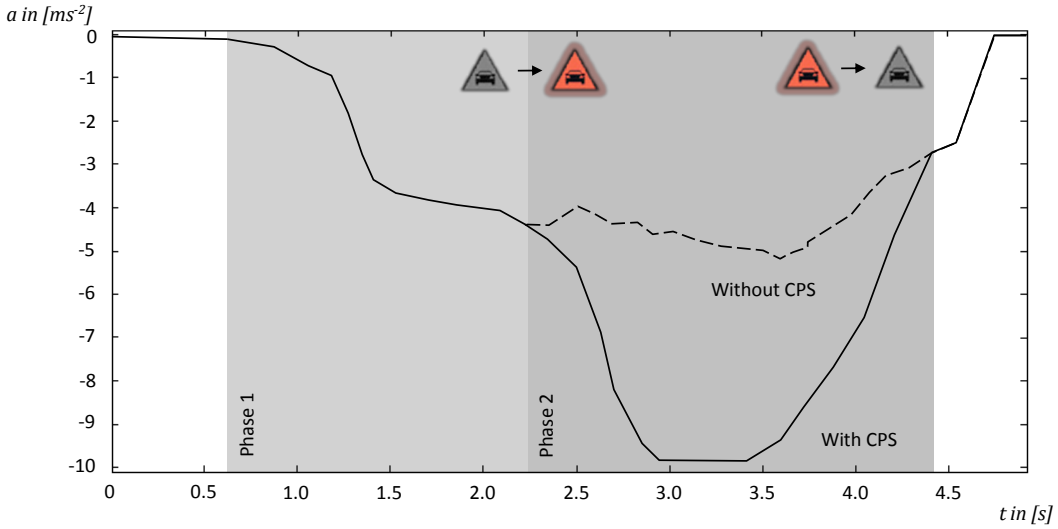
Figure 5.4 shows two examples for accident-critical situations, in which the CPS supports the driver for preventing the collision. In the first situation (Figure 5.4(a)), the System Vehicle is approaching a parked car, i.e. a stationary object, at about $60 kmh^{-1}$. The optical and acoustical warning is issued at $t = 0.5[s]$, when the relative distance between Object and System Vehicle is about $55[m]$. This phase is named *Phase 1*. Then, the driver reacts to the warning and starts to brake at approximately $t = 1.6[s]$ (*Phase 2*). $0.2[s]$ later, the CPS activates its brake support so that the resulting deceleration nearly reaches its maximal value (*Phase 3*). In addition, the figure also includes the theoretical deceleration without the CPS. This curve is based on a plant model, which translates the brake pedal travel originating from the driver's reaction into a deceleration. It has to be noted that this deceleration does not give evidence whether a collision would have happened without the CPS, because the driver's behaviour might have been influenced by the system. At $t = 3.0[s]$, the reaction time has increased above the threshold value, so the warning is deactivated although the system is still supporting the driver to finish the manoeuvre.

In the second situation, cf. Figure 5.4(b), the System Vehicle is driving in a convoy on a highway with a speed of $80[kmh^{-1}]$. The Object starts a slight braking manoeuvre at $t = 0.5[s]$. The driver of the System Vehicle follows the manoeuvre immediately by applying a similar deceleration. As the situation is not critical at this point of time, the warning is not activated in *Phase 1*. Then, the Object suddenly

increases its deceleration to the maximum value at $t = 2.2[s]$. Because the brake lights of the Object were already activated, the driver is not able to recognise the change of the situation until the relative distance between Object and System Vehicle has been considerably decreased. In contrast, the CPS is able to measure the change of the relative speed due to the Doppler effect and identify the change of the Object's acceleration. It therefore activates the brake support and controls the deceleration of the System Vehicles to prevent the collision (*Phase 2*). At the same time, the



(a)



(b)

Figure 5.4: Two examples for situations, in which the CPS prevented a collision

warning is activated to alert the driver of the criticality of the situation.

The fleet's measurement data was also simulated with the approaches described above, see Figure 5.2. If an invalid situation was fixed by a new release and this fix could be proven by simulation, the situation was marked accordingly in the database. With this, we were not only able to create statistics about valid and invalid situations, but also continuously update these statistics during the development cycles of the system. In consequence, the real-life testing allowed the constant optimisation of the system's performance, and confirmed its correct functioning in real traffic situations.

5.3 Summary

To summarise, this chapter presented different testing scenarios for the Collision Prevention System. The effectiveness of the CPS was proven by trials in a driving simulator. The results of these trials showed that system significantly reduces the accident rates, when compared to a car without the system. For this, three scenarios were defined which inevitably lead to an accident-critical situation. As each test driver had to manage all three scenarios during her/his test drive in the simulator, the density of rear-end collisions per kilometre was much higher than in reality. Thus the driving simulator allows to evaluate the impact of the system on the rate of rear-end collisions in a shorter time than in reality, and without the risk of a real collision.

Prototype vehicles were used to perform experiments on test tracks and in real life. The test-track experiments served for the verification of the system's functional requirements by using contrived situations, for which it is possible to exactly described the expected behaviour. In addition, the real-life experiments were used to validate the test-track results in real-life situations. Their focus was laid on creating statistics about valid and invalid activations of the system, and on decreasing the number of invalid situations to a minimum value.

After the release of the system in 2005, researchers of Europe's largest automobile club – the ADAC (Allgemeiner Deutscher Automobil-Club e.V.) – tested and analysed the system in different situations. In their report, the researchers identified a number of scenarios, in which the system might help to prevent a collision. These scenarios were divided into two groups: (i) attentive drivers, and (ii) inattentive drivers. For both groups, the report states a major safety benefit of the system when driving in a convoy. Especially if the Object slightly starts to brake and then increases its deceleration immediately, the CPS provides a big potential for preventing the collision (similar to the example shown in Figure 5.4(b)). For the second group, a benefit also

occurs when approaching the end of a traffic jam at speeds lower than $70[kmh^{-1}]$, or when the Object is cutting into the lane of the System Vehicle. With this, the researchers of the ADAC approved that the realised Collision Prevention System is working and, in doing so, improving the vehicle's safety.

Chapter 6

Analysis of the Test-Driven Development Process

The purpose of this chapter is the analysis of the novel test-driven development process for embedded control systems which has been introduced in Chapter 2 and applied to four different components in Chapters 3 and 4. First the applicability of design patterns to control system design and the test-driven development process is evaluated and similarities as well as differences are identified. This discussion results in the definition of two new design patterns which identify and summarise the role of TDD for control system design. Moreover the process is assessed in terms of advantages and disadvantages when compared to other approaches for the development of control systems.

6.1 Comparison to Design Patterns

In the following, a selection of design patterns is used to analyse and evaluate the methods and results of the presented approach for test-driven control system design. The discussion is divided into two subsections: In the first section, different patterns for test-driven development are described to define the role of TDD as a tuning process for controllers. In contrast, the second section explains how the well-known Model-View-Controller pattern can be used to derive the role of TDD as a design process for control systems. To distinguish a pattern's name within the text, the first letter is a capital and the whole word is underlined.

6.1.1 TDD Tuning Process

In the first book about test-driven development Kent Beck also presented a pattern language for test-driven development [66, pp. 123ff]. This language is divided into several groups: (i) for problems and solutions when a red bar or a green bar occurs, (ii) for testing in general, (iii) for testing with xUnit (see Section 2.2.2), (iv) for refactoring and (v) for the process itself. The book also covers the application of TDD to a subset of the object-oriented design patterns by Gamma et al. [72].

The fundamental pattern of TDD is Test First, which describes when a test is written, i.e. before the code is changed. Furthermore, all tests have to be automated (Test) and independent of each other (Isolated Test). The automation is typically realised by a member of the xUnit family of testing frameworks. The patterns of this family and the realisation with the slUnit testing framework have already been described in Section 2.2. As a consequence, we are able to write tests, automatically execute them and verify their results in a model-based development environment with a graphical programming language such as Simulink.

The pattern Assert First requires to write the assertions of a test first. This helps the developer to think about the problem which is to be solved. For instance, an assertion specifies that the rise time should be smaller than a specified value. Such an assertion leads to two questions: What is the input stimulus for which the response achieves the rise time specified? What is the plant model? The answers to both questions are part of the new test. The combination of input stimulus, plant model and expected results makes the test understandable (Test Data) as well as explicitly describes their relationship (Evident Data). For this reason, TDD does properly support the tuning process of controllers.

As an example, we can define a pattern for LQG control as it is commonly described in textbooks on control theory:

> *Pattern LQG control*
>
> ***Context***
>
> *The system is described by a state-space representation:*
>
> $$\begin{aligned} \dot{\underline{x}}(t) &= A\underline{x}(t) + B\underline{u}(t) \\ \underline{y}(t) &= C\underline{x}(t) + D\underline{u}(t) \\ \underline{x}(0) &= \underline{x}_0 \end{aligned}$$
>
> *The pair (A, B) is stabilisable.*

Problem

The input variable should be defined by state feedback:

$$\underline{u} = -K\underline{x}$$

K should be chosen to minimize a linear-quadratic performance index.

$$J = \int_0^\infty (\underline{x}^T Q \underline{x} + \underline{u}^T R \underline{u}) dt$$

Q and R define the tradeoff between the regulation performance and the control effort.

Solution

K is calculated by

$$K = -R^{-1} B^T P$$

with P being the positive-definite solution of the algebraic matrix Riccati equation:

$$A^T P + PA - PBR^{-1} B^T P + Q = 0$$

To realise this pattern, the developer starts with some initial values for Q and R. Then, the performance index is typically not minimized by solving the Ricatti equation directly, but by using a numerical calculation to retrieve the values of K. These values are evaluated, for instance by a simulation of the closed-loop system, to check whether the system fulfils the requirements. If not, Q and R are modified and the calculation is started again. TDD can be incorporated into this process as it is iterative, starts with a requirement, and is finished when the actual behaviour matches the expected behaviour, as specified in the requirements. Thus a pattern is defined which describes the role of TDD in driving a tuning process:

Pattern TDD Tuning Process

Context

A control system is given by a plant with a measurable output and a controllable input. A number of requirements exists, which describe the desired performance of the control system. The plant input is determined by a feedback controller.

Problem

The parameters of the controller should be chosen so that all requirements can be fulfilled.

Solution

The controller is realised with the following iterative process which is repeated until all requirements are fulfilled:

1. *Create a new test with a plant model, input stimuli and assertions for output variables. Typically the assertions are used in conjunction with one or more performance criteria.*
2. *Run all tests. The new test has to fail.*
3. *Tune the controller with a (common) tuning technique, so that all tests pass. Example: Pole-Placement, LQG optimisation (see the pattern LQG control), PID tuning with Ziegler-Nichols.*
4. *Remove duplication.*

In the context of Beck's patterns, this role is supported by the pattern One Step Test. It describes how to choose the next test from the to-do list of tests. Basically, always a test should be selected which is suggestive and easy to implement. A constant and steady progress with only small steps should lead to a growing and evolutionary design, while preventing solutions which are too complex. Beck describes this as a known-to-unknown metaphor: The developer usually has some knowledge, which allows him to realise the first steps, but also to gain experience and draw conclusions about the further course of the development. This approach is directly applicable to control engineering. We typically start with a known method like pole-placement to implement a simple controller. This controller allows us to learn more about the closed-loop system, especially if the characteristics of the real plant are only partially known.

The to-do list of tests can be generated implicitly or explicitly from the set of requirements which should be implemented (Test List). Furthermore items might be added to the list during the development cycle because of a defect, either detected by another test or externally reported, i.e. not detected by automated tests (Regression Test). If a test already fails, a new test is only necessary if the reason of the failure is beyond the scope of the failed test. Moreover, the to-do list can be extended by new tests during the development due to the known-to-unknown metaphor

(Learning Test) or other issues which are rather specific to the implemented software unit, but not included in the requirements. A common example is the check for a division by zero. To summarise, four origins of a new test exist as shown in Figure 6.1.

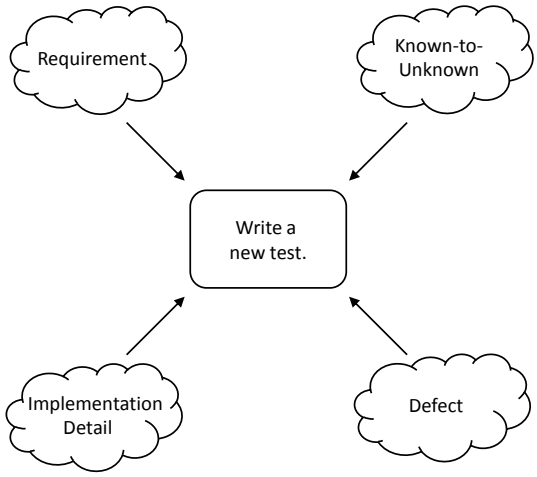


Figure 6.1: Origins of a new test

6.1.2 TDD Design Process

A frequently used architectural pattern for software engineering is the Model-View-Controller pattern (MVC) [197]. Please note that the terms "Model" and "Controller" here have a different meaning than in control theory. To distinguish the different meanings, the first letter is a capital when we speak of the pattern meaning, and a lower-case letter otherwise.[1] The Model contains the core of the application and represents its domain-specific information. The View is the output interface to the user and displays the Model's data. Its counterpart is the Controller, which implements the input interface. With the Controller, the user is able to access the model's functions. In terms of embedded systems, the Controller and the View are typically realised as one or more different electronic control units and physically connected to the electronic control unit which implements the model, see Figure 6.2(a). Furthermore the Controller and the View do not only interface to the user, but also with the system's environment through sensors and actuators. TDD supports the development process of this pattern by replacing these components with mock interfaces, i.e. test stimuli for the input variables and assertions for the output values.

[1] Thus model and controller are terms of control theory, Model and Controller terms of the MVC pattern.

An example for such a system is the Collision Prevention System introduced in this thesis, which uses the Instrument Cluster for the View and the brake pedal as well as other input devices for the Controller. Both are connected via a CAN bus to the Radar Control Unit which realises the Model.

The main disadvantage of this description is that the plant is not explicitly included in one of the three MVC components. Its potential position would be in-between the Controller and the View, thus connecting sensors and actuators. Therefore the MVC pattern is modified to characterise an embedded control system in the context of control theory. Here, the Model represents the plant and the Controller represents the implemented software including, for example, a feedback controller. The View comprises the input and output interfaces only to the user, but not to the sensors and actuators as they are part of internal interface between Controller and Model. The block diagram of the modified MVC pattern is shown in Figure 6.2(b).

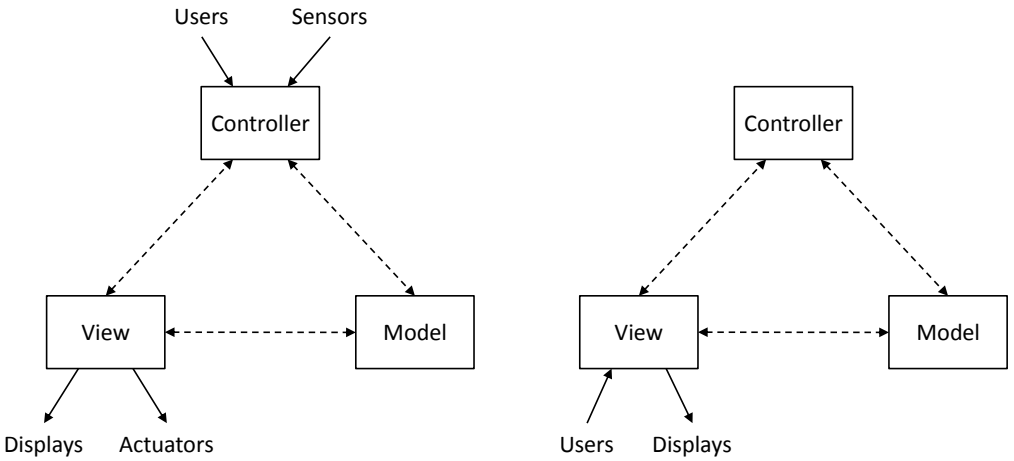


Figure 6.2: The Model-View-Controller pattern

The advantage of the modified pattern is that it helps us to better understand the role of TDD as a software process. The TDD approach can be divided into two simple elements: the Test and the Test Objective. While a test objective realises a part of the system's requirements, a test drives its design and verifies its realisation. Therefore both elements constitute an executable form of the system's requirements. Compared to MVC, we identify the Controller as the test objective and the Model as well as the View as its design guide and its verification. The Model can be described by the real plant, i.e. by system tests in a real environment, but also by plant models.

In fact, a single unit test usually specifies only a part of the Model or even only a single aspect of it. The same applies to the View. The major difference between the Model and the View is that the Model is typically tested in a closed loop together with the Controller due to the use of plant models. Instead, the View is tested as an open loop, i.e. the user's input is specified independently of the system's response. An example has been given with the development of the component Driver Assessment in Section 3.4.2.

Furthermore, the iterative cycles of TDD allow to analyse and to react to unknown or incompletely known characteristics of the plant. When we start with an unknown plant, a new test case describes an input stimulus of the plant, while the outputs are observed by the developer. In Figure 6.3, a test case is represented by a circle within the Model or the View. The geometrical form of a circle was chosen, because the plant's characteristics are never fully known. Then a part of the Controller is added to make the failing test pass. As the Controller can be fully implemented in terms of its requirements, its rectangle is filled with squares. The process continues with the next test which explores another, more complex behaviour of the plant. This is supported by the initial implementation of the Controller as it makes it possible to stimulate the plant as a closed-loop system. In general, it can be said that

- with each new test a part of the real plant can be explored or known characteristics can be described; this is shown in Figure 6.3 by the increase of circles within the rectangles of Model and View; and
- with each modification or extension of the Controller, the number and the complexity of the realised requirements is rising; this is shown in Figure 6.3 by the increase of squares within the rectangle of the Controller.

As a result of this discussion, a second pattern for the test-driven development process can be defined:

> *Pattern TDD Design Process*
>
> ***Context***
>
> *A control system should be described by the MVC pattern. The Model is given by a plant with a measurable output and a controllable input, the View is given by a user interface. The characteristics of the Model and the View might be only partially known. A number of requirements exists, which describe the desired behaviour and features of the system from the user's point of view.*

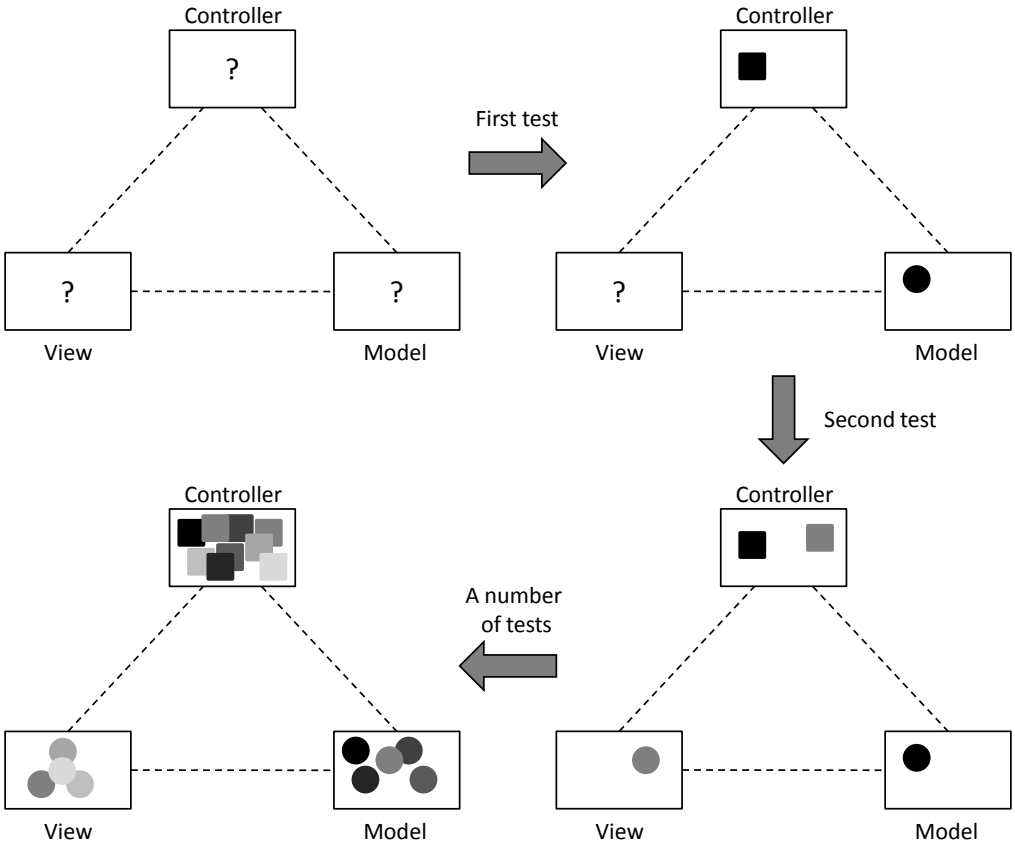


Figure 6.3: Test-Driven Development as a Design Process

Problem

The Controller should be implemented.

Solution

The Controller is realised with the following iterative process which is repeated until all requirements are fulfilled:

1. *Create a new test which either represents a part of the View or the Model. The test can include a plant model, input stimuli and assertions for output variables. Typically the combination of input stimuli and assertions describe the desired behaviour of the test objective. Another aim of the tests is to explore the characteristics of the real plant or the user interface as shown in Figure 6.3*

2. *Run all tests. The new test has to fail.*

3. *Implement the Controller so that all tests pass.*

4. *Remove duplication.*

To summarise, we have defined two patterns for the test-driven development of control system: the TDD Tuning Process and the TDD Design Process. Both patterns are complementary to each other. They describe the role of TDD in terms of control system design, while focussing on different levels of the system.

6.2 Comparison to the Traditional V-Model Approach

When the role of TDD is compared to other (software) development processes, the key features of TDD are its short and iterative development cycles, and the test-first paradigm. These features allow the application of TDD to both the tuning of a controller and the design of the whole control system.

In contrast, traditional software development models can not be applied to a tuning process, e.g. the tuning of a PID controller, because of their long development cycles and the strict sequence of process steps. In the following, we consider the V-model as an example for such a traditional process model. In theory, the shortest cycle of the V-model consist of three steps: The definition of a requirement, its implementation by writing code and its verification by testing. An example requirement for a PID controller could be that the rise time is smaller than a specified value. The requirement is then realised by, for instance, modifying the gain of the controller. This transition between the requirement and its implementation is not specified within the V-model. Therefore the gain value is usually chosen using initial constraints such as the formulae of Ziegler and Nichols [26] and the experience of the developer. Especially with model-based development, the developer might also create a kind of test case by stimulating the inputs of the closed-loop system and observing the outputs. Although this step is not part of the V-model, it can often be observed in the typical work flow of an embedded system's developer. Finally, the implementation is verified by one or more tests (which might or might not be based on the previously created test case). Following the path of the V-Model, the developer is not allowed to jump back to the implementation phase if one test fails. Instead he/she has to pass another iteration, which again starts with the definition of a requirement. For our example, this requirement would be to pass the test. This leads to a trial-and-error process, which most developers try to avoid by *unofficial* testing. The problem of

such testing is that it does not follow any patterns or rules, e.g. the specification of expected values, but depends on the habits of the developer. Moreover it is usually not included into activities such as configuration management or change management as the provisional testing is not specified within the process.

Test-driven development avoids these problems by requiring a test for every new requirement and its implementation. It structures the work of the developer in a natural way. Compared to the simplified V-model, first a requirement is defined, then a test is written and the system is implemented in such a way that the test passes. As a consequence, the test represents the transition between the requirement and its implementation. The developer is not forced to create a temporary test bed to evaluate his ideas, but instead provided with systematic feedback from the test cases which have been written in advance. The test beds are part of any process management activities, e.g. saved in a revision control system. In fact, it can be said that the implemented units of a system *live* within their test beds. By using referenced models, this *biotope* is not even destroyed during the integration phase. Instead, the developer is always able to return to a particular unit and its test bed for making modifications. During such modifications, the tests provide a safety net which ensures that none of the previously realised functionality is broken.

As already pointed out, TDD can also be used as a design process for whole control systems. Here the tests do not only represent the requirements of the controller, but also help to explore the characteristics of the plant and to describe them in terms of executable models. In addition, a set of acceptance tests defines the goal for the end of the development cycle. The resulting product fulfils all requirements, which where covered by the acceptance tests. When a geometrical representation for this process similar to the V-model is created, it results in two circles – the outer circle for acceptance testing, the inner circle for unit testing together with the implementation of the system, cf. Figure 6.4.

The test-first paradigm of test-driven development causes also the major drawback of this process: It must be possible to test a unit before it exists. This disadvantage is especially important for the development of hardware parts. A simple example is the test for the existence of a resistor. If such a test is executed before the resistor is installed, a short circuit might occur and with this, other components might get damaged. Furthermore the change of the hardware's layout or interface requires much more effort than with software and is usually limited by certain constraints such the maximum number of I/O ports of the microprocessor used. The hardware development process therefore typically follows much longer cycle times than a software development process. A solution of the problem is the combination of the V-model

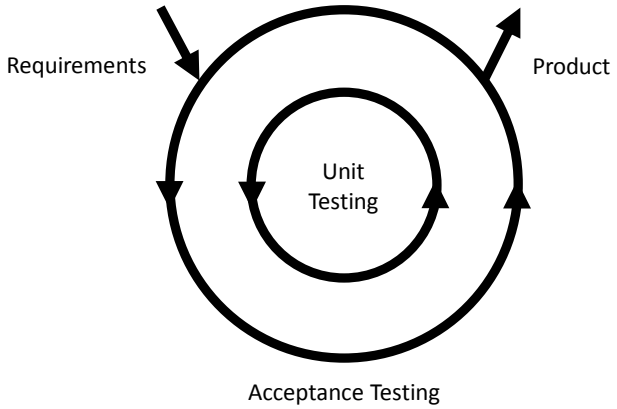
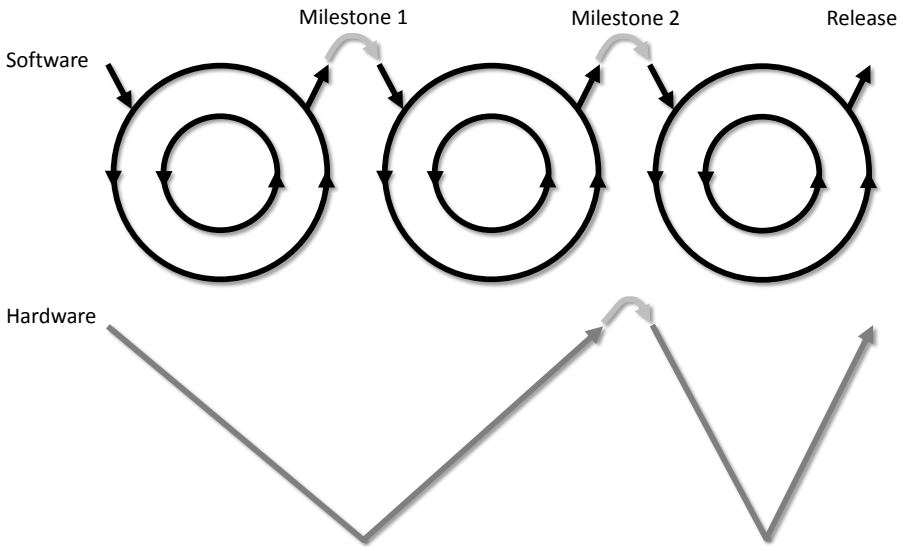


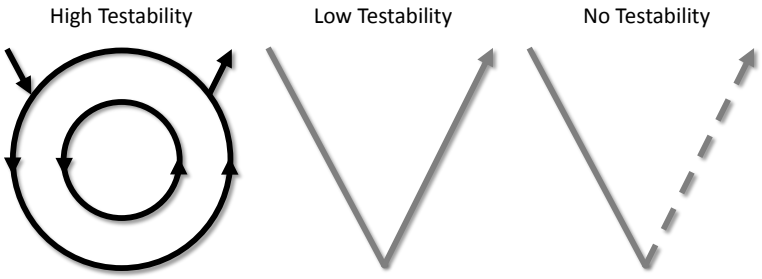
Figure 6.4: The Test-Driven Process

and the test-driven process into a joint process model, as shown in Figure 6.5(a). The V-model is used for the hardware development, the test-driven process is employed for the software development. The software development process can have shorter cycle times, i.e. a new software can be released without a new hardware. The opposite case, i.e. a new hardware without a new software, is also possible, e.g. for cost optimisation, but more common is a simultaneous release of both parts. In Figure 6.5(a), the software has three releases compared to two release of the hardware. The hardware is not available at the first milestone, thus the complete system can only be tested by simulation in a Model-in-the-Loop or Software-in-the-Loop setup. The second milestone and the final release comprises both software and hardware, hence all testing setups are possible.

Another disadvantage of TDD is the requirement of testing frameworks for the different parts of the system. While such frameworks are available for nearly all programming languages, they are not applicable to all parts of the embedded software. Especially software components which directly access the system's hardware are difficult to test. The common approach is the use of mock objects, but the realisation of such objects might require a much higher effort than the component itself. The same problem applies to certain hardware components. In general, the availability of a testing framework and with this, the testability of the system and its components, is not only an issue for test-driven development. Usually all process models include activities for system verification and therefore need at least some kind of testing facility. Typically, if a testing facility for a component is not available, the component is only tested at higher levels, i.e. integrated in a group of components. At least the whole system is always testable in terms of acceptance testing, because otherwise the



(a) Software and Hardware



(b) Testability

Figure 6.5: Usage of different process models depending on the component and its testability

customer would not be able to verify their requirements. Therefore the solution for this problem is again to fall back to a V-model. In Figure 6.5(b), the right arm of this V-model is drawn with a dashed line to indicate the limited possibilities for testing. Thus, the joint model considers the specific process requirements and testability of different kinds of components. It allows the application of TDD even to systems that are not completely testable.

Chapter 7

Conclusions and Outlook

7.1 Conclusions

This thesis described the use of test-driven development as a design method for embedded control systems with the focus on automotive safety systems. The approach of test-driven development was chosen because it has gained importance as a modern software development process in recent years. The main idea of test-driven development is to use tests not only for the verification of the system, but also as a design method. The tests are written before any part of the system is realised or modified. The implementation evolves together with the definition of tests, which not only serve to specify the expected behaviour, but also to explore unknown characteristics of the system and its environment. This evolving design can lead to a better quality and fewer defects of the realised system.

The basic prerequisite for test-driven development is the availability of an automated testing framework as tests are executed very often. Such testing frameworks had been developed for nearly all programming languages, but not for the graphical, signal driven language Simulink. Simulink has become very important for the design of control systems, mainly in the automotive industry, the aerospace industry and others industries such as process control. It allows a model-based development process, which is based on and centred around the model as the central artefact, supports an iterative development style and supplements the traditionally mathematical approaches to control system design. In this thesis the process of model-based devel-

opment was therefore analysed and the concepts and realisation of a testing framework for Simulink has been addressed. Furthermore, different testing techniques, testing setups and test design techniques have been introduced. The description of these methods is completed by the definition of the test-driven development cycle and its usage at different layers of an automotive safety system.

During the PhD project, the author worked as an employee of the former Daimler-Chrysler AG at the Mercedes-Benz Technology Center, Department for Driver Assistance Systems. One main focus of the department was the development of longitudinal vehicle control systems based on RADAR sensors. In addition to systems for conventional and adaptive cruise control, semi-autonomous parking, and night vision, a series project started for a new collision warning and avoidance system in 2003. A similar system, which is named Collision Prevention System in this thesis, was used as an example for the application of test-driven development to an automotive safety system. The definition and the requirements of the system were introduced and a suitable system architecture was derived. As a result, three software components of the system represented problems of particular areas for the realisation of control systems, i.e. logical combinations, experimental problems and mathematical algorithms. For each of these problems, a concept to systematically derive test cases from the requirements was presented. Finally, the realisation of the components has been described and illustrated by example test cases.

In addition, the longitudinal controller of the Collision Prevention System was considered as a fourth component which was the focus of a separate chapter to examine the specific background of control theory. First, two conventional approaches for the derivation of the controller were introduced, before the controller was developed in a test-driven way. Again, a concept to derive test cases has been presented and the realisation was then accompanied by example test cases. Furthermore both, the conventional and test-driven approaches were compared in terms of their stability and performance by using results from the simulation in Simulink. This comparison showed that test-driven development led to a stable and robust controller, which fulfils the system's requirements and is therefore capable of preventing a collision in most of the cases. It was furthermore easier to incorporate the effects of external logic and non-linearities in the TDD approach.

The Collision Prevention System was implemented and evaluated in trials on a driving simulator which showed that the system leads to a significant reduction of the accident rate for rear-end collisions. In addition, experiments with prototype vehicles on test tracks were presented to verify the system's functional requirements within a system testing approach. The results of these test-track experiments have

also been validated by a field test with a fleet of prototype vehicles.

Figure and Text have been removed due to Copyright and Access restrictions.

7.2 Outlook and Further Work

Amongst other things, the future development of automotive safety systems will focus on the data fusion of multiple environmental sensors, especially of RADAR sensors and vision-based sensors. The use of different sensor technologies not only allows to build systems with an extended functionality, e.g. to detect pedestrians or crossing vehicles, but also to increase the reliability of the object detection. This is necessary to realise systems which will execute more severe actions without being activated by the driver, for example an autonomous full braking manoeuvre. Moreover, the reliability of such autonomous systems will not only be improved by better or more sensors, but also by improvements in the development process. One of the most important trends in the development of automotive systems is the application of the international standard *IEC 61508*, which covers the functional safety of electrical and electronic safety-related systems. This standard, which has its origin in the process control industry sector, covers the complete safety life cycle of a product, i.e. analysis, design, implementation, modification, operation and maintenance. It is currently adopted for automotive systems by a working group of the International Organization for Standardization (ISO). The working draft is called *ISO-WD 26262*.

In the future, the test-driven development of embedded control systems could be an important factor for the functional safety of such systems. The method of specifying tests as the transition between requirements and their corresponding implementation not only allows the design and exploration of unknown characteristics, but also the seamless documentation and verification of the implemented system. Thus it is possible to trace back each requirement or design decision made by a developer, as these decisions are represented as executable tests and the corresponding set of assertions. As a consequence, also the testability of the system and its components is inevitably increased. The further development of test-driven development for embedded control systems will therefore focus on three main objectives:

Linking: Nowadays, part of the common workflow of model-based development is the creation of links between the description of the requirements and their implementation as Simulink models or subsystems. The links allow a $m-to-n$ relationship between the requirements and the models or subsystems, which state that m requirements are implemented by n models or subsystems. With

TDD, these links directly refer to the executable tests. Hence the semantics are changed so that m requirements are represented by n test cases, and n test cases stimulate and verify the results of o models or subsystems.

With slUnit the test objective and the tests are both part of the test bed. Thus, it should be possible to create links also between a single test and the corresponding parts of the test objective. Such a linkage might be even automatically determined by the analysis of the statement or branch coverage that results from the execution of the test.

In addition, both kinds of links can be added to the automatically generated source code to support the process step of code reviews. During code reviews the participants, usually divided into the author and the reviewers, attend a number of meetings and review code line by line. Such inspections are very thorough and have been proven effective at finding defects. With embedded links, the reviewer would be able to jump between the generated source code, the model, the requirement and the test cases, and compare them to each other. Moreover, it could be possible to interactively walk through the code by executing a particular test and observing the results of statements or the values of variables.

Fault Injection: A typical issue of testing is the question whether the tests cover all logical paths through the unit which is being tested. Several test design techniques to resolve this issue have already been presented in Section 2.3.1. An alternative to these techniques is the manual or automatic injection of faults into the realised system. One aim of this approach is to activate the functional paths which cover the system's error handling. For automotive systems, a common example is the injection of failures to the vehicle's communication network by creating a timeout, modifying a message's checksum or setting the data to invalid values. In contrast, fault injection can also be used to verify the coverage of the specified tests. Initially, all tests must be successfully passed. Then the model or code is modified at a specific position, e.g. by changing a mathematical operation from "plus" to "minus" or a relational operator from "equals" to "not equals". After each of these code mutations, all tests are executed. The assertion is now that at least one test fails. Otherwise, the mutation is not covered by the current set of tests.

A basic integration of fault injection into Simulink can be easily realised for two reasons: (i) Simulink already provides a set of methods to obtain the handles

of a specific set of blocks, e.g. all relational operators, and (ii) the blocks can be modified by directly accessing their parameters. Thus (and in contrast to common programming languages) no separate parser is needed for the source code. Moreover, the blocks and logical paths that are covered by a particular test can be automatically marked, for instance, by modifying their background colour. Using the colour spectrum, it is even possible to indicate the statistical coverage, i.e. by how many tests a specific block is covered.

Continuous Integration: Continuous integration is a software engineering technique, which emerged in the extreme programming community, and consists of one basic rule: every modification of the implemented system is immediately committed to a revision control system. The integration of all changes is done by an automated server process, which monitors the revision control system for changes, then automatically runs the build process and executes all tests. This might also include additional steps such as generating documentation and code statistics. The advantage of this process is that integration problems, broken code and conflicting changes are detected early and fixed continuously.

Although continuous integration seems to be unusual for embedded systems, it is suitable due to the widely existing simulation environments. With Simulink a model can be automatically committed to a revision control system by using a callback function after it has been saved. Then a process on a dedicated server machine can load these files to a MATLAB background process and execute all unit tests with slUnit. If all tests passed, code is generated and compiled by the build process. The process can also download the generated binary file to the target platform and run additional tests, for instance, on a Hardware-in-the-Loop facility. The results of each phase, i.e. unit testing, code generation, build process, and system testing, are included into a report, which can be published on a web server or wiki, sent via email, or itself committed to a revision control system. With this, the progress of the development process is successively documented and allows a rollback to nearly each step of the design or implementation.

With the implementation of these objectives and the continuous improvement of the development processes, the automotive industry will be able to realise even more complex safety systems on their way to the vision of accident-free driving.

Bibliography

[1] C. Benz, "Vehicle powered by a gas-engine (Fahrzeug mit Gasmotorbetrieb, in German)," Patent 37 435. German Patent Office (Reichspatentamt), 1886.

[2] N. Georgano, *The Complete Encyclopaedia of Motorcars : 1885 – 1968*. Ebury, 1969.

[3] N. Georgano, Ed., *The Beaulieu Encyclopedia of the Automobile*. Routledge, 2000.

[4] The International Organization of Motor Vehicle Manufacturers, "The World's Automotive Industry," 2006. Online: http://www.oica.net/htdocs/statistics/OICA_depliant-final.pdf

[5] UPI Umwelt und Prognose-Institut Heidelberg e.V., "Consequences of a global mass motorisation (Folgen einer globalen Massenmotorisierung, in German)," 1995. Online: http://www.upi-institut.de/upi35.htm

[6] National Highway Traffic Safety Administration, USA, "Traffic Safety Facts 2005," 2005.

[7] Statistisches Bundesamt Deutschland, "Traffic Accidents – Time Series 2005 (Verkehrsunfälle – Zeitreihen 2005, in German)," 2005.

[8] DaimlerChrysler, "Precise and Safe," *Hightech Report*, no. 1, pp. 60–61, 2005.

[9] DaimlerChrysler, "Safety Pioneer," 2006. Online: http://www.daimlerchrysler.com/Projects/c2c/channel/documents/898181_daimlerchrysler_safety_pioneer.pdf

[10] Robert Bosch GmbH, "From Innovation to Standard. 25 Years ABS." 2003. Online: http://www.bosch.de/start/media/BOSCH_ABS_Infowand_eng.pdf

[11] T. Costlow, "Shifting into the active mode," *Automotive Engineering*, 2007.

[12] F. Dudenhöffer, "Driver Assistance Systems – Market with a Fivefold Potential (Fahrerassistenzsysteme – Markt mit fünffachem Potenzial, in German)," *Automobil Industrie*, vol. 49, no. 3, pp. 74–75, 2004.

[13] G. Hanser, "Future market for driver assistance systems (Zukunftsmarkt Assistenzsysteme, in German)," *Automotive*, no. 11-12, pp. 22–24, 2004.

[14] Verband Deutscher Wirtschaftsingenieure, "Intelligent cars of tomorrow (Die intelligenten Autos von morgen, in German)," *technologie & management*, no. 7-8, pp. 20–22, 2005.

[15] B. Hedenetz, "Design of distributed fault-tolerant electronic architectures for automobiles (Entwurf von verteilten fehlertoleranten Elektronikarchitekturen in Kraftfahrzeugen, in German)," Ph.D. dissertation, University of Tübingen, Germany, 2001.

[16] DaimlerChrysler, "Fatigue at the wheel: Mercedes-Benz developing warning system for motorists," 2006. Online: http://www.daimlerchrysler.com.au/dc_australia/ 0-172-65707-1-659238-1-0-0-0-0-0-11386-65707-0-0-0-0-0-0-0.html

[17] R. Schöneburg, K.-H. Baumann, and R. Justen, "PRE-SAFE – The next step in the enhancement of Vehicle Safety," in *Proceedings of the 18th International Technical Conference on the Enhanced Safety of Vehicles*. Nagoya, Japan, National Highway Traffic Safety Administration, USA, May 2003.

[18] T. Dohmke, "Automotive networks – CAN, FlexRay and MOST (Bussysteme im Automobil – CAN, FlexRay und MOST, in German)," Technical University of Berlin, Technical Report, 2002. Online: http://thomas.dohmke.de/dokumente/bussysteme.pdf

[19] G. Airy, "On the Regulator of the Clock-Work for Effecting Uniform Movement of Equatorials," *Memoirs of the Royal Astronomical Society*, vol. 11, 1840.

[20] J. Maxwell, "On Governors," in *Proceedings of the Royal Society London*, vol. 16, London, Great Britain, 1868, pp. 270–283.

[21] M. Lyapunov, "Problème général de la stabilité du mouvement," *Annales de la Faculté des Sciences de Toulouse*, vol. 9, 1907.

[22] H. Black, "Stabilized Feedback Amplifiers," *Bell System Technical Journal*, vol. 13, 1934.

[23] H. Nyquist, "Regeneration Theory," *Bell System Technical Journal*, vol. 11, 1932.

[24] H. Bode, "Feedback Amplifier Design," *Bell System Technical Journal*, vol. 19, 1940.

[25] N. Minorsky, "Directional Stability and Automatically Steered Bodies," *Journal of the American Society of Naval Engineers*, vol. 34, 1922.

[26] J. Ziegler and N. Nichols, "Optimum settings for automatic controllers," *Transactions of the ASME*, vol. 62, pp. 759–768, 1942.

[27] W. Evans, "Graphical Analysis of Control Systems," *Transactions of the AIEE*, vol. 67, 1948.

[28] R. Kalman and J. Bertram, "Control System Analysis and Design via the Second Method of Lyapunov," *ASME Journal of Basic Engineering*, vol. 82, 1960.

[29] J. Ragazzini and L. Zadeh, "The Analysis of Sampled-Data Systems," *Transactions of the AIEE*, vol. 71, no. 2, 1952.

[30] K. Aström, *Introduction to Stochastic Control Theory*. Academic Press, 1970.

[31] H. Rosenbrock, *Computer-Aided Control System Design*. Academic Press, 1974.

[32] A. MacFarlane and I. Postlethwaite, "The Generalized Nyquist Stability Criterion and Multivariable Root Loci," *International Journal of Control*, vol. 25, 1977.

[33] J. Doyle and G. Stein, "Multivariable Feedback Design: Concepts for a Classical/Modern Synthesis," *IEEE Transactions on Automatic Control*, vol. 26, no. 2, 1981.

[34] K. Dutton, S. Thompson, and B. Barraclough, *The Art of Control Engineering*. Addison-Wesley, 1997.

[35] O. Föllinger, *Control engineerung (Regelungstechnik, in German)*. Hüthig, 1994.

[36] J. Doyle, B. Francis, and A. Tannenbaum, *Feedback Control Theory*. Macmillan Publishing, 1997.

[37] K. Aström, "Control System Design," 2002. Online: http://www.cds.caltech.edu/~murray/courses/cds101/fa02/caltech/astrom.html

[38] L. Ljung, *System Identification*. Prentice Hall, 1999.

[39] F. Lewis, *Applied Optimal Control and Estimation*. Prentice-Hall, 1992.

[40] T. Klein, B. Rumpe, and B. Schätz, Eds., *Model-Based Development of Embedded Systems (Modellbasierte Entwicklung eingebetteter Systeme, in German)*. Wadern, Germany, Technical University of Braunschweig, 2005.

[41] H. Giese, B. Rumpe, and B. Schätz, Eds., *Model-Based Development of Embedded Systems II (Modellbasierte Entwicklung eingebetteter Systeme II, in German)*. Wadern, Germany, Technical University of Braunschweig, 2006.

[42] M. Conrad, H. Giese, B. Rumpe, and B. Schätz, Eds., *Model-Based Development of Embedded Systems III (Modellbasierte Entwicklung eingebetteter Systeme III, in German)*. Wadern, Germany, Technical University of Braunschweig, 2007.

[43] B. Schätz, A. Pretschner, F. Huber, and J. Philipps, "Model-Based Development of Embedded Systems," Technical University of Munich, Technical Report, 2002.

[44] A. Rau, "Model-based Development of Embedded Automotive Control Systems," Ph.D. dissertation, University of Tübingen, Germany, 2002.

[45] The Mathworks, "Simulink," 2007. Online: http://www.mathworks.com/products/simulink/

[46] W. W. Royce, "Managing the development of large software systems: Concepts and techniques," in *Proceedings of the IEEE WESCON*. Los Angeles, CA, USA, IEEE, August 1970, pp. 1–9.

[47] B. W. Boehm, "A spiral model of software development and enhancement," *Computer*, pp. 62–72, 1988.

[48] A. Bröhl and W. Dröschel, *The V-model: The Standard for Software Development (Das V-Modell: Der Standard für die Software-Entwicklung, in German)*. Oldenbourg, 1993.

[49] A. Spillner, "From V-Model to W-Model – Establishing the Whole Test Process," in *Proceedings of the 4th Conference on Quality Engineering in Software Technology*, Nuremberg, Germany, September 2000, pp. 222–231.

[50] B. Broekman and E. Notenboom, *Testing Embedded Software*. Addison-Wesley, 2003.

[51] K. Beck, "Embracing Change with Extreme Programming," *Computer*, vol. 32, no. 10, pp. 70–77, 1999.

[52] K. Beck, *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

[53] G. Keefer, "Extreme Programming Considered Harmful for Reliable Software Development," 2002. Online: http://www.stickyminds.com/getfile.asp?ot=XML&id=3248&fn=XDD3248filelistfilename1%2Epdf

[54] K. Beck, "Aim, fire [test-first coding]," *IEEE Software*, vol. 18, pp. 87–89, 2001.

[55] D. North, "Test-Driven Development is not about Testing," 2003. Online: http://www.sys-con.com/story/?storyid=37795

[56] B. George and L. Williams, "An Initial Investigation of Test-Driven Development in Industry," in *ACM Symposium on Applied Computing*, Madrid, Spain, March 2002.

[57] B. George, "Analysis and Quantification of Test Driven Development Approach," Master's thesis, North Carolina State University, North Carolina, USA, 2002.

[58] E. Maximilien and L. Williams, "Assessing test-driven development at IBM," in *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*. Portland, OR, USA, IEEE, May 2003, pp. 564–569.

[59] D. Janzen and H. Saiedian, "A leveled examination of test-driven development acceptance," *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, pp. 719–722, May 2007.

[60] D. Janzen and H. Saiedian, "Does test-driven development really improve software design quality?" *IEEE Software*, vol. 25, no. 2, pp. 77–84, 2008.

[61] L. Williams, E. Maximilien, and M. Vouk, "Test-driven development as a defect-reduction practice," in *Proceedings of the 14th International Symposium on Software Reliability Engineering*. Denver, CO, USA, IEEE, November 2003, pp. 34–45.

[62] D. Janzen and H. Saiedian, "Test-Driven Development: Concepts, Taxonomy, and Future Direction," *IEEE Computer*, vol. 38, no. 9, pp. 43–50, 2005.

[63] H. Erdogmus, M. Morisio, and M. Torchiano, "On the effectiveness of the test-first approach to programming," *IEEE Transactions on Software Engineering*, vol. 31, no. 3, pp. 226–237, 2005.

[64] A. Geras, M. Smith, and J. Miller, "A Prototype Empirical Evaluation of Test Driven Development," in *Proceedings of the 10th IEEE International Symposium on Software Metrics*. Chicago, IL, USA, IEEE Computer Society, September 2004, pp. 405–416.

[65] B. George and L. Williams, "A structured experiment of test-driven development," *Information and Software Technology*, vol. 46, no. 5, pp. 337–342, 2004.

[66] K. Beck, *Test-Driven Development: By Example*. Addison-Wesley, 2002.

[67] P. Hamill, *Unit Test Frameworks*. O'Reilly, 2004.

[68] G. Meszaros, *XUnit Test Patterns*. Addison-Wesley, 2007.

[69] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, and I. Fiksdahl-King, *A Pattern Language – Towns Buildings Construction*. Oxford University Press, 1977.

[70] C. Alexander, *The Timeless Way of Building*. Oxford University Press, 1979.

[71] K. Beck and W. Cunningham, "Using Pattern Languages for Object-Oriented Programs," in *Proceedings of OOPSLA '87*, Orlando, FL, USA, October 1987.

[72] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[73] M. Kircher and M. Völter, "Software Patterns," *IEEE Software*, vol. 24, no. 4, pp. 28–30, 2007.

[74] R. Sanz and J. Zalewski, "Pattern-based control systems engineering," *IEEE Control Systems Magazine*, vol. 23, no. 3, pp. 43–60, 2003.

[75] M. J. Pont, *Patterns for time-triggered embedded systems: Building reliable applications with the 8051 family of microcontrollers*. Addison-Wesley, 2001.

[76] M. J. Pont and M. P. Banner, "Designing embedded systems using patterns: a case study," *Journal of Systems and Software*, vol. 71, no. 3, pp. 201–213, 2004.

[77] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. Wiley, 2000.

[78] M. Kirchner, P. Jain, D. Schmidt, and A. Corsaro, "Patterns and pattern languages for oo distributed real-time and embedded systems," in *Proceedings of OOPSLA '01*, Tampa Bay, FL, USA, October 2001.

[79] M. Nelson, "A design pattern for autonomous vehicle software control architectures," in *Proceedings of the 23rd International Computer Software and Applications Conference (COMPSAC '99)*, Phoenix, AZ, USA, October 1999, pp. 172–177.

[80] B. Selic, "An architectural pattern for real-time control software," in *Proceedings of the 3rd Annual Pattern Languages of Programming Conference*, Urbana-Champaign, IL, USA, September 1996.

[81] J. Zalewski, "Real-Time Software Design Patterns," in *Proceedings of the 9th Polish Conference on Real-Time Systems*, Ustron, Poland, September 2002, pp. 23–42.

[82] R. McKegney and D. T. Shepard, "Design patterns and real-time object-oriented modeling (poster session)," in *Proceedings of OOPSLA '00*. Minneapolis, MN, USA, ACM Press, October 2000, pp. 55–56.

[83] D. Lea, "Design Patterns for Avionics Control Systems," State University of New York, Technical Report, 1994.

[84] T. Dohmke and H. Gollee, "Test-Driven Development of a PID Controller," *IEEE Software*, vol. 24, no. 3, pp. 44–50, 2007.

[85] B. Danner, T. Dohmke, J. Hillenbrand, V. Schmid, and A. Spieker, "Method for Identifying Critical Collision Situations from the Rear," Patent WO/2006/040 032. German Patent Office (Deutsches Patent- und Markenamt), 2006.

[86] B. Danner, T. Dohmke, J. Hillenbrand, V. Schmid, and A. Spieker, "Method for Identifying Rear End Collision-Critical Situations in Lines of Traffic,"

Patent WO/2006/048 148. German Patent Office (Deutsches Patent- und Markenamt), 2006.

[87] B. Danner, T. Dohmke, J. Hillenbrand, V. Schmid, and A. Spieker, "Method and Vehicle Assistance System for Preventing Collisions or Reducing the Severity of a Vehicle Collision," Patent WO/2006/053 652. German Patent Office (Deutsches Patent- und Markenamt), 2006.

[88] B. Danner, T. Dohmke, J. Hillenbrand, V. Schmid, and A. Spieker, "Method for Operating a Collision Avoidance System of a Vehicle and Associated Collision Avoidance System," Patent WO/2006/053 654. German Patent Office (Deutsches Patent- und Markenamt), 2006.

[89] B. Danner, T. Dohmke, J. Hillenbrand, V. Schmid, and A. Spieker, "Method for Operating a Braking Assistance System in a Vehicle," Patent WO/2006/053 667. German Patent Office (Deutsches Patent- und Markenamt), 2006.

[90] B. Danner, T. Dohmke, J. Hillenbrand, V. Schmid, and A. Spieker, "Method for Avoiding a Collision or for Reducing the Consequences of a Collision and Device for Carrying Out Said Method," Patent WO/2006/053 655. German Patent Office (Deutsches Patent- und Markenamt), 2006.

[91] B. Danner, T. Dohmke, J. Hillenbrand, V. Schmid, and A. Spieker, "Method for Adapting Intervention Parameters of an Assistance System of a Vehicle," Patent WO/2006/061 106. German Patent Office (Deutsches Patent- und Markenamt), 2006.

[92] B. Danner, T. Dohmke, J. Hillenbrand, V. Schmid, and A. Spieker, "Method for Operating a Collision Avoidance System or Collision Sequence Reducing System of a Vehicle, and Collision Avoidance System or Collision Sequence Reducing System," Patent WO/2006/072 342. German Patent Office (Deutsches Patent- und Markenamt), 2006.

[93] B. Danner, T. Dohmke, J. Hillenbrand, V. Schmid, and A. Spieker, "Method for Operating a System for Avoiding Collisions or for Reducing the Consequences of a Collision for a Vehicle and a Corresponding System for Avoiding Collisions or for Reducing the Consequences of a Collision," Patent WO/2006/097 467. German Patent Office (Deutsches Patent- und Markenamt), 2006.

[94] T. Dohmke and V. Schmid, "Method for Operating an Assist System for a Vehicle and Park Assist System," Patent WO/2006/117 064. German Patent Office (Deutsches Patent- und Markenamt), 2006.

[95] A. Fuggetta, "A Classification of CASE Technology," *Computer*, vol. 26, no. 12, pp. 25–38, 1993.

[96] D. C. Schmidt, "Model-Driven Engineering," *Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[97] T. Bruckhaus, N. H. Madhavji, I. Janssen, and J. Henshaw, "The Impact of Tools on Software Productivity," *IEEE Software*, vol. 13, no. 5, pp. 29–38, 1996.

[98] C. Jones, *Applied Software Measurement*. MacGraw Hill, 1997.

[99] International Electrotechnical Commission (IEC), *IEC 61131: Programmable controllers – Part 3: Programming languages*, Standard, 2003.

[100] Manufacturer Supplier Relationship (MSR), *Standardization of library blocks for graphical model exchange*, Standard, 2001.

[101] I. Nassi and B. Shneiderman, "Flowchart techniques for structured programming," *SIGPLAN Notices*, vol. 8, no. 8, pp. 12–26, 1973.

[102] IBM, "Flowcharting Techniques," Technical Report, 1969.

[103] International Organization for Standardization (ISO), *ISO 5807: Information processing – Documentation symbols and conventions for data, program and system flowcharts, program network charts and system resources charts*, Standard, 1985.

[104] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987.

[105] B. Meyer, "Reusability: The Case for Object-Oriented Design," *IEEE Software*, vol. 4, no. 2, pp. 50–64, 1987.

[106] D. C. Rine and B. Bhargava, "Object-Oriented Computing," *Computer*, vol. 25, no. 10, pp. 6–10, 1992.

[107] International Organization for Standardization (ISO), *ISO/IEC 19501: Unified Modeling Language (UML) Version 1.4.2*, Standard, 2005.

[108] Object Management Group (OMG), *Unified Modelling Language (UML) Version 2.0*, Standard, 2005.

[109] Luqi and W. Royce, "Status Report: Computer-Aided Prototyping," *IEEE Software*, vol. 9, no. 6, pp. 77–81, 1992.

[110] K. Balasubramanian, A. Gokhale, G. Karsai, J. Sztipanovits, and S. Neema, "Developing Applications Using Model-Driven Design Environments," *Computer*, vol. 39, no. 2, pp. 33–40, 2006.

[111] A. Pretschner, "Model-based testing," in *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*. St. Louis, MO, USA, IEEE Computer Society, May 2005, pp. 722–723.

[112] W. Grieskamp, "Multi-paradigmatic Model-Based Testing," in *First Combined International Workshops on Formal Approaches to Software Testing and Runtime Verification*, Seattle, WA, USA, August 2006, pp. 1–19.

[113] D. Gluch and J. Brockway, "An Introduction to Software Engineering Practices Using Model-Based Verification," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Technical Report, 1999.

[114] B. Schätz, T. Hain, F. Houdek, W. Prenninger, M. Rappl, J. Romberg, O. Slotosch, M. Stecker, and A. Wisspeintner, "CASE Tools for Embedded Systems," Technical University of Munich, Technical Report, 2003.

[115] S. Toeppe, D. Bostic, S. Ranville, and K. Rzemien, "Automatic Code Generation Requirements for Production Automotive Powertrain Applications," in *Proceedings of the IEEE International Symposium on Computer Aided Control System Design*. Kohala Coast, HI, USA, IEEE, August 1999, pp. 200–206.

[116] P. Hansen, "Model-Based Tools Update," *The Hansen Report on Automotive Electronic*, vol. 14, no. 5, 2001.

[117] C. Moler, "The Growth of MATLAB and The MathWorks over Two Decades," 2006. Online: http://www.mathworks.com/company/newsletters/news_notes/clevescorner/jan06.pdf

[118] The Mathworks, "MATLAB," 2007. Online: http://www.mathworks.com/products/matlab/

[119] T. Riemer, G. Baumann, P. Kappelmann, D. Hötzer, and A. Kröhnert, "Evaluation of Code-generators based on Simulink models," in *Proceedings of 6th Stuttgart International Symposium on Automotive and Engine Technology*, Stuttgart, Germany, February 2004.

[120] R. Hammarström and J. Nilsson, "A Comparison of Three Code Generators for Models Created in Simulink," Master's thesis, Chalmers Univerity of Technology, Göteborg, 2006.

[121] K. Beck, "Simple Smalltalk testing," *The Smalltalk Report*, vol. 4, no. 2, pp. 16–18, 1994.

[122] K. Beck and E. Gamma, "Test Infected: Programmers Love Writing Tests," *The Java Report*, vol. 3, no. 7, pp. 37–50, 1998.

[123] T. Dohmke, "mlUnit: MATLAB Unit Testing," 2005. Online: http://mlunit.dohmke.de/

[124] M. Beine, M. Conrad, M. Eschmann, I. Fey, K. Lamberg, and R. Ottenbach, "Model-Based Testing of Embedded Automotive Software using Mtest," in *Proceedings of the SAE World Congress*, Detroit, MI, USA, March 2004.

[125] dSPACE, "MTest," 2007. Online: http://www.dspace.ltd.uk/ww/en/ltd/home/products/sw/expsoft/mtest.cfm

[126] The Mathworks, "SystemTest," 2007. Online: http://www.mathworks.com/products/systemtest/

[127] R. Systems, "Model-based Testing and Validation of Control Software with Reactis," Technical Report, 2003.

[128] Reactive Systems, "Reactive Tester," 2007. Online: http://www.reactive-systems.com/tester.msp

[129] M. Blackburn and R. Busser, "T-VEC: a tool for developing critical systems," in *Proceedings of 11th Annual Conference on Computer Assurance (COMPASS '96)*, Gaithersburg, MD, USA, June 1996, pp. 237–249.

[130] T-VEC, "Tester for Simulink," 2007. Online: http://www.t-vec.com/solutions/simulink.php

[131] Y. Zhang, "Test-Driven Modeling for Model-Driven Development," *IEEE Software*, vol. 21, no. 5, pp. 80–86, 2004.

[132] T. Dohmke, "The slUnit Testing Framework," 2006. Online: http://slunit.dohmke.de

[133] K. Meffert, "JUnit 4 – New version supporting Java 5 features," *Java Magazin*, no. 5, 2006.

[134] The Mathworks, "Simulink V6.0," 2004. Online: http://www.mathworks.com/access/helpdesk/help/toolbox/simulink/rn/bqmg7gf-1.html

[135] MathWorks Automotive Advisory Board, "Controller Style Guidelines for Production Intent Development Using MATLAB, Simulink, and Stateflow," Technical Report, 2001.

[136] IEEE, *Standard Computer Dictionary*, Standard 610, 1990.

[137] G. Myers, *The Art of Software Testing*. John Wiley and Sons, 1979.

[138] W. C. Hetzel, *The Complete Guide to Software Testing*. John Wiley and Sons, 1984.

[139] M. Hutcheson, *Software Testing Fundamentals*. Wiley, 2003.

[140] M. Grochtmann and K. Grimm, "Classification Trees for Partition Testing," *Software Testing, Verification & Reliability*, vol. 3, no. 2, pp. 63–82, 1993.

[141] M. Grochtmann, K. Grimm, and J. Wegener, "Tool-Supported Test Case Design for Black-Box Testing by Means of the Classification-Tree Editor," in *Proceedings of the 1st European International Conference on Software Testing, Analysis and Review (EuroSTAR '93)*, London, Great Britain, October 1993, pp. 169–176.

[142] J. Wegener and M. Grochtmann, "Computer-aided test case design for functional testing with the classification-tree editor (Werkzeugunterstützte Testfallermittlung für den funktionalen Test mit dem Klassifikationsbaum-Editor CTE, in German)," in *Proceedings of the GI-Symposium Softwaretechnik 93*, Dortmund, Germany, November 1993.

[143] M. Grochtmann, "Test Case Design using Classification Trees," in *Proceedings of the 3rd International Conference on Software Testing, Analysis and Review (STAR '94)*, Washington, DC, USA, May 1994.

[144] M. Grochtmann, J. Wegener, and K. Grimm, "Test Case Design Using Classification Trees and the Classification-Tree Editor CTE," in *Proceedings*

of the 8th International Software Quality Week, San Francisco, CA, USA, May 1995, pp. 1–11.

[145] E. Lehmann and J. Wegener, "Test Case Design by Means of the CTE XL," in *Proceedings of the 8th European International Conference on Software Testing, Analysis and Review (EuroSTAR '00)*, Kopenhagen, Denmark, December 2000.

[146] H. Sthamer, A. Baresel, and J. Wegener, "Evolutionary Testing of Embedded Systems," in *Proceedings of the 14th International Internet and Software Quality Week*, San Francisco, USA, May 2001.

[147] J. Wegener and O. Bühler, "Evaluation of Different Fitness Functions for the Evolutionary Testing of an Autonomous Parking System," in *Proceedings of the Genetic and Evolutionary Computation Conference*. Seattle, WA, USA, Springer, June 2004.

[148] O. Bühler and J. Wegener, "Evolutionary Functional Testing of a Vehicle Brake Assistant System," in *Proceedings of the 6th Metaheuristics International Conference*, Vienna, Austria, August 2005.

[149] H. Zhu, P. Hall, and H. May, "Software unit test coverage and adequacy," *ACM Computing Surveys*, vol. 29, no. 4, pp. 336–427, 1997.

[150] S. Vilkomir, K. Kapoor, and J. Bowen, "Tolerance of control-flow testing criteria," in *Proceedings of the 27th Annual International Computer Software and Applications Conference (COMPSAC '03)*, Dallas, TX, USA, November 2003.

[151] R. H. Cobb and H. D. Mills, "Engineering Software Under Statistical Quality Control," *IEEE Software*, vol. 7, no. 6, pp. 44–54, 1990.

[152] H. L. Guen and T. Thelin, "Practical Experiences with Statistical Usage Testing," in *Proceedings of the 11th Annual International Workshop on Software Technology and Engineering Practice*. Amsterdam, The Netherlands, IEEE Computer Society, September 2003, pp. 87–93.

[153] J. C. Huang, "An Approach to Program Testing," *ACM Computing Surveys*, vol. 7, no. 3, pp. 113–128, 1975.

[154] W. E. Howden, "Methodology for the Generation of Program Test Data," *IEEE Transactions on Computers*, vol. 24, no. 5, pp. 554–560, 1975.

[155] M. Pol, R. Teunissen, and E. V. Veenendaal, *Software Testing, a Guide to the TMap Approach*. Addison-Wesley, 2002.

[156] P. Herman, "A data flow analysis approach to program testing," *The Australhlian Computer Journal*, vol. 8, no. 3, pp. 92–96, 1976.

[157] S. Ntafos, "On required element testing," *IEEE Transactions on Software Engineering*, vol. 10, no. 6, pp. 795–803, 1984.

[158] J. Laski, "On data flow guided program testing," *ACM SIGPLAN Notices*, vol. 17, no. 9, pp. 62–71, 1982.

[159] J. W. Laski and B. Korel, "A Data Flow Oriented Program Testing Strategy," *IEEE Transactions on Software Engineering*, vol. 9, no. 3, pp. 347–354, 1983.

[160] R. Floyd, "Assigning meanings to programs," in *Proceedings of Symposia in Applied Mathematics*, vol. 19. Providence, RI, USA, American Mathematical Society, Providence, R.I., 1967, pp. 19–32.

[161] C. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1968.

[162] B. Meyer, "Applying 'Design by Contract'." *IEEE Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[163] D. S. Rosenblum, "Towards a method of programming with assertions," in *Proceedings of the 14th International Conference on Software Engineering (ICSE '92)*. Melbourne, Australia, ACM Press, May 1992, pp. 92–104.

[164] D. S. Rosenblum, "A practical approach to programming with assertions," *IEEE Transactions on Software Engineering*, vol. 21, no. 1, pp. 19–31, 1995.

[165] J. M. Voas, "How Assertions Can Increase Test Effectivenes," *IEEE Software*, vol. 14, no. 2, pp. 118–119,122, 1997.

[166] M. Conrad and E. Sax, "Mixed Signals," in *Testing Embedded Software*, B. Broekman and E. Notenboom, Eds. Addison-Wesley, 2003, pp. 229–249.

[167] J. Boensch, "Implementation of Functions for Test and Diagnostics of Chassis Control Systems (Implementierung von Test- und Diagnosefunktionen für Fahrwerksteuergeräte, in German)," Master's thesis, University of Tübingen, Germany, 1999.

[168] A. Rau, "Using Assertions and Watchdogs in a Model-Based Development Process (Verwendung von Zusicherungen in einem modellbasierten Entwicklungsprozess, in German)," *it+ti – Informationstechnik und Technische Informatik*, 2002.

[169] H. Wiesbrock, M. Conrad, I. Fey, and H. Pohlheim, "A New Automated Evaluation Method for Regression and Back-to-back Tests (Ein neues automatisiertes Auswerteverfahren für Regressions- und Back-to-Back-Tests eingebetteter Regelsysteme, in German)," *Softwaretechnik-Trends*, vol. 22, no. 3, 2002.

[170] M. Conrad, I. Fey, and H. Pohlheim, "Automated Test Evaluation for ECU Software (Automatisierung der Testauswertung für Steuergerätesoftware, in German)," in *Proceedings of the Electronic Vehicle Systems Conference*, Baden-Baden, Germany, September 2003, pp. 299–315.

[171] C. Ritter, J. Willibald, E. Sax, and K. Müller-Glaser, "Accompanying the Design Process by Tests for the Model-based Development of Embedded Systems (Entwurfsbegleitender Test für die modellbasierte Entwicklung eingebetteter systeme, in German)," in *Proceedings of the 13th Workshop Test Methods and Reliability of Circuit and Systems*, Miesbach, Germany, February 2001.

[172] R. Jeffries, A. Anderson, and C. Hendrickson, *Extreme programming installed*. Addison-Wesley, 2001.

[173] L. Crispin and T. House, *Testing Extreme Programming*. Addison-Wesley, 2002.

[174] F. Fraikin, M. Hamburg, S. Jungmayr, T. Leonhardt, A. Schönknecht, A. Spillner, and M. Winter, "The Illusion of Reliability with the Green Bar (Die Trügerische Sicherheit des Grünen Balkens, in German)," *OBJEKTspektrum*, no. 1, 2004.

[175] International Organization for Standardization (ISO), *ISO 3888-1: Passenger cars – Test track for a severe lane-change manoeuvre – Part 1: Double lane-change*, Standard, 1999.

[176] International Organization for Standardization (ISO), *ISO 3888-2: Passenger cars – Test track for a severe lane-change manoeuvre – Part 2: Obstacle avoidance*, Standard, 2002.

[177] R. Johnson and W. Opdyke, "Refactoring: An aid in designing application frameworks and evolving object-oriented systems," in *Proceedings of the Symposion on Object-Oriented Programming Emphasizing Practical Applications*, Poughkeepsie, NY, USA, September 1990.

[178] W. Opdyke, "Refactoring Object-Oriented Frameworks," Ph.D. dissertation, University of Illinois, USA, 1992.

[179] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[180] H.-E. Schurk and H. Fournell, "Onboard diagnosis of electronics, a contribution to vehicle reliability," in *Proceedings of the 37th IEEE Vehicular Technology Conference*, vol. 37, Tampa, FL, USA, June 1987, pp. 343–350.

[181] B. Lammen, "Automatic Collision Avoidance for Automobiles (Automatische Kollisionsvermeidung für Kfz, in German)," Ph.D. dissertation, University of Dortmund, Germany, 1993.

[182] U. Lages, *Research of Active Accident Avoidance (Untersuchungen zur aktiven Unfallvermeidung, in German)*. VDI Verlag, 2000.

[183] F. Mildner, "Research into the Detection and Prevention of Automotive Collisions (Untersuchungen zur Erkennung und Vermeidung von Unfällen für Kraftfahrzeuge, in German)," Ph.D. dissertation, University of the German Federal Armed Forces, Hamburg, 2004.

[184] O. Ararat, E. Kural, and B. Güvenc, "Development of a Collision Warning System for Adaptive Cruise Control Vehicles Using a Comparison Analysis of Recent Algorithms," in *Proceedings of the Intelligent Vehicles Symposium*, Instanbul, Turkey, June 2006, pp. 194–199.

[185] B. Krogh, "A Generalized Potential Field Approach to Obstacle Avoidance," Society of Manufacturing Engineers, Michigan, USA, Technical Report, 1984.

[186] Telelogic, "DOORS," 2007. Online: http://www.telelogic.com/products/doors/doors/index.cfm

[187] W. Kiesewetter, W. Klinkner, W. Reichelt, and M. Steiner, "The New Brake Assistance of Mercedez Benz: Active Driver Aid in Emergency Circumstances," *Automobiltechnische Zeitschrift ATZ*, vol. 99, no. 6, pp. 330–339, 1997.

[188] V. Schmid, W. Bernzen, J. Schmitt, and D. Reutter, "A new dimension of Active and Passive Safety with PRE-SAFE and Brake Assist BAS PLUS in the new Mercedes-Benz S-Class," in *Proceeding of the Electronic Vehicle Systems Conference*. Baden-Baden, Germany, VDI, October 2005, pp. 215–224.

[189] J. Hillenbrand, K. Kroschel, and V. Schmid, "Situation assessment algorithm for a collision prevention assistant," in *Proceedings of the Intelligent Vehicles Symposium*, Las Vegas, NV, USA, June 2005, pp. 459–465.

[190] P. Fancher and Z. Bareket, "Evaluating Headway Control Using Range Versus Range-Rate Relationships," *Vehicle System Dynamics*, vol. 23, pp. 575–596, 1994.

[191] B. Schiller, V. Morellas, and M. Donath, "Collision Avoidance for Highway Vehicles Using the Virtual Bumper Controller," in *Proceedings of the IEEE International Conference on Intelligent Vehicles*, Stuttgart, Germany, October 1998, pp. 149–155.

[192] V. L. Bageshwar, W. L. Garrad, and R. Rajamani, "Model Predictive Control of Transitional Maneuvers for Adaptive Cruise Control Vehicles," *IEEE Transactions on Vehicular Technology*, vol. 53, no. 5, pp. 1573–1585, 2004.

[193] R. Caudill and W. Garrard, "Vehicle-follower longitudinal control for automated transit vehicles," *Journal of Dynamic Systems, Measurement, and Control*, vol. 99, pp. 241–248, 1977.

[194] G. F. Franklin, J. D. Powell, and A. Emami-Naeini, *Feedback Control of Dynamic Systems*. Addison-Wesley, 2002.

[195] W. L. Brogan, *Modern control theory*. Prentice Hall, 1991.

[196] A. E. Bryson and Y.-C. Ho, *Applied optimal control*. Ginn, 1969.

[197] G. Krasner and S. Pope, "A cookbook for using the model-view controller user interface paradigm in Smalltalk-80," *Journal of Object-Oriented Programming*, vol. 1, no. 3, pp. 26–49, 1988.