University of Kaiserslautern
Department of Computer Sciences
AG Software Engineering: Dependability
Prof. Dr.-Ing. habil. Peter Liggesmeyer

Fraunhofer-Institute for Experimental
Software Engineering IESE
Department of Component Engineering

Evaluation of a Model-Based Development Process for Automotive Embedded Systems

Model-Based Development of an Adaptive Cruise Control System

Diploma Thesis

Jonas Mitschang *(jonas@mitschang.net)*

August 14, 2009

Supervisor:
Prof. Dr.-Ing. habil. Peter Liggesmeyer
Dr.-Ing. Mario Trapp
Dipl. Inf. Donald Barkowski

Technische Universität Kaiserslautern - Postfach 3049 - 67653 Kaiserslautern

Hiermit erkläre ich, Jonas Mitschang, dass ich die vorliegende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Jonas Mitschang,

Kaiserslautern, den 14. August, 2009

Abstract

Nowadays, vehicle control systems such as anti-lock braking systems, electronic stability control, and cruise control systems yield many advantages. The electronic control units that are deployed in this specific application domain are embedded systems that are integrated in larger systems to achieve predefined applications. Embedded systems consist of embedded hardware and a large software part. Model-based development for embedded systems offers significant software-development benefits that are pointed out in this thesis.

The vehicle control system *Adaptive Cruise Control* is developed in this thesis using the model-based software development process for embedded systems suggested in [BST09]. As a modern industrial design tool that is prevalent in this domain, Matlab/Simulink® is used for modeling the environment, the system behavior, for determining controller parameters, and for simulation purposes. Using an appropriate toolchain, the embedded code is automatically generated.

The adaptive cruise control system could be successfully implemented and tested within this short timespan using a waterfall model without increments. The vehicle plant and important filters are fully deduced in detail. Therefore, the design of further vehicle control systems needs less effort for development and precise simulation.

Accordingly, the groundwork has been laid for the development of additional control systems in future. Additionally, development time estimations are provided in this thesis for different scenarios.

Keywords: Adaptive Cruise Control, Model-Based Software Development, Embedded Systems, Automotive

Contents

1. Introduction 9

2. Related Work 13

3. Target System 17
- 3.1. CAN-bus . . . 18
- 3.2. Sensors . . . 18
  - 3.2.1. Remote Control Receiver . . . 19
  - 3.2.2. Wheel Speed Sensors . . . 20
  - 3.2.3. Acceleration Sensors . . . 21
  - 3.2.4. Rotation Sensor . . . 21
  - 3.2.5. Distance Sensors . . . 22
- 3.3. Actuators . . . 22
  - 3.3.1. Throttle . . . 23
  - 3.3.2. Steering Servo . . . 24
- 3.4. Embedded Controller Board . . . 24
  - 3.4.1. Hardware of the Controller Board . . . 24
  - 3.4.2. Software of the Controller Board . . . 24

4. Model-Based Software Development Process for Embedded Systems 27
- 4.1. Requirements Analysis . . . 28
  - 4.1.1. Powering on the System . . . 28
  - 4.1.2. Enabling the Controller . . . 29
  - 4.1.3. Disabling the Controller . . . 30
  - 4.1.4. Adjust Desired Speed . . . 30
  - 4.1.5. Brake . . . 31
  - 4.1.6. Control Speed . . . 31
- 4.2. Functional Design . . . 34
  - 4.2.1. Derivation of the Vehicle Plant . . . 35
  - 4.2.2. Filter for the Wheel Speed Sensors . . . 41
  - 4.2.3. General Control Loop Theory . . . 44
  - 4.2.4. PID Controller . . . 45

4.2.5. Control Speed 49
4.2.6. Control Distance 52
4.3. Software Architecture 57
4.3.1. Static Structure 58
4.3.2. Dynamic Structure and Dynamic Interaction 62
4.4. Software Design 64
4.4.1. Structural Refinement 65
4.4.2. Behavioral Design 70
4.4.3. Platform Independent System Test 76
4.4.4. Platform-Specific Design 82
4.5. Code 85
4.5.1. Real-Time Workshop® Embedded Coder™ 85
4.5.2. Integration and Compilation 86

5. Evaluation of the Development Process 89
5.1. Time Distribution of the Overall Process 90
5.2. Normal Time Distribution for Domain Experienced Personnel 91
5.3. Estimated Time Distribution for Further Developments of Vehicle Control Systems on the same Target Platform 92

6. Conclusion and Future Work 95
6.1. Conclusion 95
6.2. Future Work 96

Bibliography 97

A. Necessary Real-Time Workshop® Configuration Settings 105

B. Concept Car CAN Overview 107

C. Description of the Generated Files 109

D. Tracing Code Back to the Model 111

E. Summary of Files Generated by SimulinkTarget 113

F. Overview of the Simulink Main Function 115

1. Introduction

Nowadays, nearly all electronic devices contain embedded systems. A vast amount of all processors and microcontrollers that are produced these days are applied in embedded systems. They are becoming more and more important in today's life in many ways. Not only are they used in vehicles and airplanes, but also in everyday appliances like coffee machines, mobile phones, and washing machines. Embedded systems are integrated in larger systems which interact with the environment to achieve a set of predefined tasks or applications. Embedded systems consist of embedded hardware and software [BS05]. The interaction of this hard- and software is of fundamental importance.

Taking this vast number of different embedded systems into consideration, one can easily imagine how time-consuming the development of those embedded systems is. On the one hand, product cycles of embedded systems are short. This is specially the case in the consumer section, where some devices are not even available for one year on the market. On the other hand, the time to market pressure is overwhelmingly high. Each product is manufactured in high quantities. Thus, one can conclude that many design problems are solved in the software instead of the hardware because the software just needs to be developed once and hardware is needed in every manufactured piece. Hence, those systems are cheaper to produce which is a crucial advantage on the market. The drawback of this hardware to software shift is that embedded system software complexity increases steadily. Another cause of this are the extended functionalities that are needed to be successful on the market. Furthermore, the required quality of those embedded systems has to meet increasingly high standards. Thus, modern methods for hard- and software development such as model-based development have to be used to fulfill the requirements in a given time and with high reliability.

The scope of this thesis is the model-based software development of a car driver assistance system using the model-based software development process for embedded systems described in [BST09]. Key principle of model-based development is graphic modeling of software in contrast to former textual programing. Model-based development of embedded systems provides important benefits over conventional approaches: One does not have to worry about implementing controllers textual in software as it was the case for traditional approaches. In contrary, the controllers are designed using special graphical and mathematical representations. Those graphical models can easily be simulated without testing them in the target platform. In the traditional approach, simulating code was not possible at all, but using these representations, testing behavior is no longer a problem. Additionally, designing and testing controllers can be done by domain experts like control systems engineers and mechanical engineers. They prefer using their domain specific modeling techniques like block diagrams for describing the behavior of filters and controllers. Therefore, the responsibilities are much better distributed and the resulting system is likely to be of higher quality.

The model-based development approach suggested in [BST09] consists of five phases: The *Requirements Analysis* step, the *Functional Design* step, the *Software Architecture* step, the *Software Design* step, and the *Code* step. As a advantage of the development of embedded systems and in contrary to other model-based development approaches, the *Functional Design* step is performed earlier in the process - as the first step after the *Requirements Analysis*. Thus, errors can be identified and corrected in early development stages by simulating the models. Accordingly, the overall development time and costs can significantly be reduced. A further advantage of model-based development of embedded systems is that one has the opportunity to reuse and extend the developed components.

In this thesis, the assistance system that is developed is an adaptive cruise control system. Adaptive cruise control systems are cruise control systems with extended vehicle following functionality. The adaptive cruise control system allows the vehicle to slow down when another vehicle is approaching ahead. In contrary, cruise control systems are not able to influence the vehicle brakes nor do they have sensors to detect vehicles ahead.

All phases of the development process are executed and described in detail to finally be able to automatically generate executable code from the behavioral models. To construct those behavioral models, the software Matlab/Simulink® of the company *The MathWorks* is used. The scientific target platform is a remote-controlled one-to-five scale concept car of the Fraunhofer-Institute for Experimental Software Engineering. It is used to apply software engineering techniques in a real environment.

This work is organized as follows: Chapter 2 gives an introduction of model-based software development for embedded systems. Different design approaches that are used in my design process are introduced. Afterwards, in chapter 3 the target embedded system is described with its interfaces to the environment. The model-based software development process for embedded systems is discussed in detail in chapter 4. This chapter is subdivided into the different development process steps: First, the process starts with the requirements analysis, discussed in chapter 4.1 and continues with the functional design phase in chapter

4.2 where domain expert knowledge is used. After the software architecture step in chapter 4.3 and the software design step in chapter 4.4, the system code is finally generated in chapter 4.5. Chapter 5 gives an evaluation on the work by analyzing the whole process. At last, chapter 6 summarises this work and provides an outlook on future work.

2. Related Work

The goal of this diploma thesis is the model-based development of an embedded adaptive cruise control system. Key principle of model-based development is graphic modeling of software in contrast to former textual programing. All phases of model-based software development use models that follow a strict syntax and may be understood as formal graphic languages. Graphic representations provide higher levels of abstraction - comparable to the transition from assembler to high-level language code.

As these models have a semantic, code may be generated from them. Different modelling languages such as Unified Modeling Language (UML) or Matlab/Simulink® [Mat04] are used in the majority of cases. It is not sufficient to restrict to just one modeling language. UML has its strengths in system architecture modeling, but it is not capable of modeling continuous behavior. Thus, it can not model filters or controllers that in general make intense use of continuous signals. In contrast Matlab/Simulink® supports modeling of continuous systems, but shows weaknesses in modeling architecture and interactions between components.

There are different approaches for model-based development of embedded systems. The model-based development process that is described in [BST09] will be applied in this diploma thesis. Is is especially suitable for design of automotive applications. It consists of the following five phases:

1. *Requirements Analysis*: In the requirements analysis phase, one has to decide what the system has to do and how well it does something. It is a predominantly textual document with additional diagrams, e.g. use case or sequence diagrams. It is important to cover all functional requirements; non-functional requirements do not matter.

2. *Functional Design*: Part of the functional design is to understand how the system may work and how to implement functional parts of the system. In the functional design phase, filters and controllers are developed including their simulation and determining their properties and parameters. The functional design describes a complete data flow

chain from the sensory input to the actuator output. When the functional design phase is finished, the engineers know which values to measure (needed sensors) and which actuators are needed. Domain experts use their modeling tools (e.g. Matlab/Simulink®) for creating the models and are able to test and verify them on powerful computers.

3. In the *Software Architecture* phase of the model-based development process for embedded systems the system is, dependent on the system complexity, structured into different systems. For example the complexity can be distributed on different electronic control units. The system architecture in this case describes the topology of the network connection of the different control units. The system software is distributed on the different units, nodes, or tasks. Static structure diagrams like the UML composite structure diagrams are used for structuring the system. Dynamic interaction diagrams like the UML sequence diagram show how external actors or the environment interacts with the system. Internal interactions are not described in the software architecture phase. In contrast to the functional design phase, where domain experts like control systems engineers or mechanical engineers work on the models, software engineers work on the software architecture models.

4. The *Software Design* phase is structured in the *Structural Refinement* step, the *Behavioral Design* step, and the *Platform Specific Design* step. In the *Software Design* phase, the complex system-models are refined to a platform specific module level. In normal cases, the refinement is performed until one is able to generate at least the main part of the system code out of the models.

   First, in the *Structural Refinement* step, the complex models from the *Software Architecture* step are hierarchically refined into subcomponents allowing the software engineer to apply the divide and conquer paradigm.

   Second, in the *Behavioral Design* step, all behaviors of all subcomponents are modeled in behaviour diagrams like state charts for state based behavior. Block diagrams (e.g. Matlab/Simulink®) are used for modeling continuous behavior. Sometimes hardware-related modules may be directly implemented in code because it might be simpler to write code for them instead of modeling them.

   Finally, the *Platform Specific Design* step extends the architecture models to be able to generate the code. In the *Software Architecture* phase, all models are platform independent but now one needs to define how the models should be integrated into the platform, e.g. bus systems or CAN message identifiers and codings. The software has to be integrated completely into the existing hardware, e.g. IO-ports or analog to digital converters.

5. After the design is finished, the *Code* phase begins. Code can now be automatically generated from the refined models. Sometimes it is necessary to implement platform code or operating system code to get the model code running. For example, the operating system provides drivers for some hardware like CAN-bus, UART, and SD card etc. The platform code passes the models in- and output ports to the operating system (e.g. CAN messages).

Three different model classes are described in [BST09] (compare figure 2.1). Structural models define the static structure of a system and are used for hierarchical step-by-step decomposition of the system into components and subcomponents. They allow the software engineer to apply the divide and conquer paradigm to reduce the problem complexity. Examples of structural models are UML *Composite Structure Diagrams* that present structural levels of decomposition with connections between subcomponents.

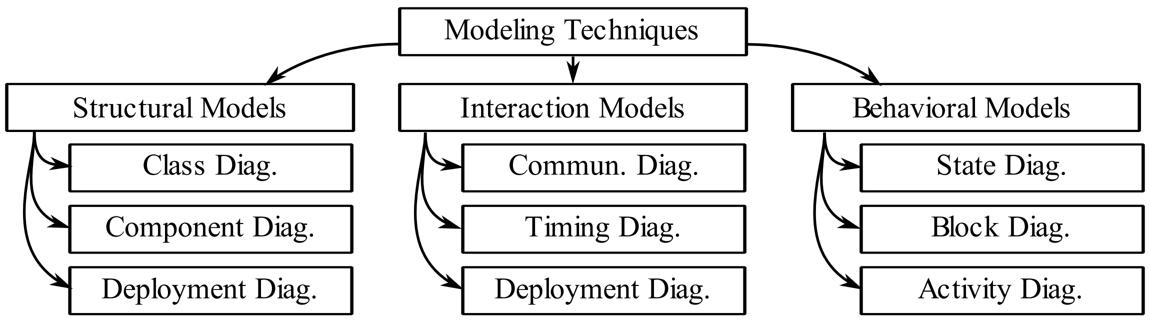


Figure 2.1.: Modeling techniques classification.

Interaction models describe the precise data flow and semantic of the communication between components. Examples for interaction models are sequence diagrams and communication diagrams. Sequence diagrams define the temporal interaction between elements of a system. They also describe interaction to the user or to external systems in the requirements phase. Communication diagrams illustrate the structure of the communication. They have less possibilities than sequence diagrams, but it is easier to show the structure and the communication relations of a system. The temporal sequence is numbered and may quickly get too complex on bigger systems.

Behavioral models describe the internal behavior of components, while interaction models only describe the communication between components. In the described model-based development process, behavioral models are used for code generation purposes and thus need a clear syntax and semantic. They have to specify the behavior of the modules uniquely and in full detail.

- *Activity Diagrams* model algorithms by describing their control flow. They define sequences of actions, operations, conditions, and loops. Activity diagrams are strongly related to code: Code elements are used for defining conditions and actions. They do not provide a good abstraction that may be able to significantly reduce the complexity of the behavioral model. For software engineers it is sometimes even easier to implement the algorithm directly instead of modeling the algorithm in an activity diagram. Furthermore, it is often easier to understand the source code than to understand the model.

- *State Diagrams* are important for modeling the behavior of embedded systems. As embedded systems often have stated-based behavior. This behavior is often referred to as automata: The current state and the automata outputs depend on the previous state and the history of occurred inputs, state diagrams are a good way of modeling embedded system behavior. The code generation can easily be achieved because the models are very formal. State diagrams may also be hierarchical structured which means that one composite state may have sub-states with its own transitions.

- *Block Diagrams* are used for modeling continuous behavior. State charts are not able to model this type of data flow based continuous behavior needed for filters and controllers (control flow versus data flow). Compared to electric circuits, inputs, outputs, base elements, connections, and junctions are connected and thus provide a well-defined functionality. In this diploma thesis, Matlab/Simulink® will be used for modeling continuous behavior with block diagrams. Matlab/Simulink® provides functions for creating subsystems and components out of models for reusing them after they are simulated and tested and seem to work properly.

3. Target System

The target system of the development of the adaptive cruise control system is a 1:5 scale remote-controlled concept car of the Fraunhofer-Institute for Experimental Software Engineering IESE. The car is an open scientific platform that is used for model-based development approaches in education [CC]. A public wiki (`http://conceptcar.iese.de`) exists for the concept car. Additionally, a subversion repository whose location and login information can be found in the wiki is available.

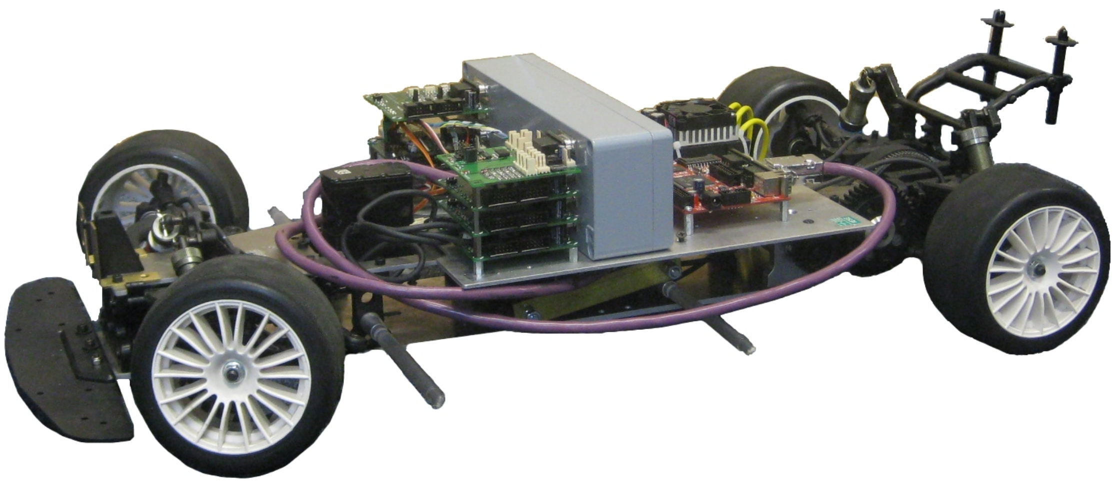


Figure 3.1.: The concept car target platform of the Fraunhofer IESE.

The vehicle is - depending in ground conditions - capable of driving at a speed of up to $50\,\frac{\mathrm{km}}{\mathrm{h}}$. The car is a drive-by-wire system that has *SensorBoards* and an *ActuatorBoard* for interfacing sensors and actuators. Both board-classes use small 8-bit microcontroller and communicate with each user using a bus system. The *SensorBoards* are described in section 3.2. Furthermore, the *ActuatorBoard* is described in section 3.3. Additionally, an embedded controller board that has more computational power (see chapter 3.4) is used for processing the dataflow from *SensorBoard* to *ActuatorBoard* if desired.

The system can be operated in two different modes:

- *Direct mode*: The *ActuatorBoard* uses the CAN messages that are directly generated by the *SensorBoards*, e.g. the remote control receiver. In this mode the concept car behaves like it would have no drive-by-wire functionality: All PWM signals are generated as if they were received at the *SensorBoards*.

- *Processed mode*: Different CAN identifiers are be used for the generation of the PWM signals. This is the normal operation mode. In this mode, the controller (Matlab/Simulink® model) receives the messages of the *SensorBoards*, drives the model and generates new CAN messages dedicated for the *ActuatorBoard*. This mode enables the use of modern vehicle control systems like adaptive cruise control.

The CAN-bus is described in more detail in chapter 3.1. Chapter 3.2 enumerates all sensors that are available on the concept car. Additionally, the actuators are described in chapter 3.3. Finally, chapter 3.4 pictures how models can be executed on the target platform and how these models can interact with the rest of the system.

3.1. CAN-bus

As stated before, the vehicle utilizes a drive-by-wire system (see figure 3.2) that uses controller-area network (CAN) bus for communication.

CAN-bus development has originally started in 1983 at the Robert-Bosch GmbH and describes an asynchronous serial bus system for communication between microcontrollers and other devices [ZS08]. It is commonly used in automotive applications i.e. for engine control units, transmissions, airbags, anti-lock braking, cruise control, audio systems, windows, doors, and even mirror adjustment. The concept car CAN-bus is configured to a transfer rate of one megabit. CAN protocol version 2.0A is used. Thus, CAN identifiers have a size of eleven bits and in total, over two thousand different identifiers are supported [ZS08].

3.2. Sensors

The sensory part of the concept car drive-by-wire system is described in this section. Sensory CAN message generated by sensors of this chapter, serve as input of my Matlab/Simu-

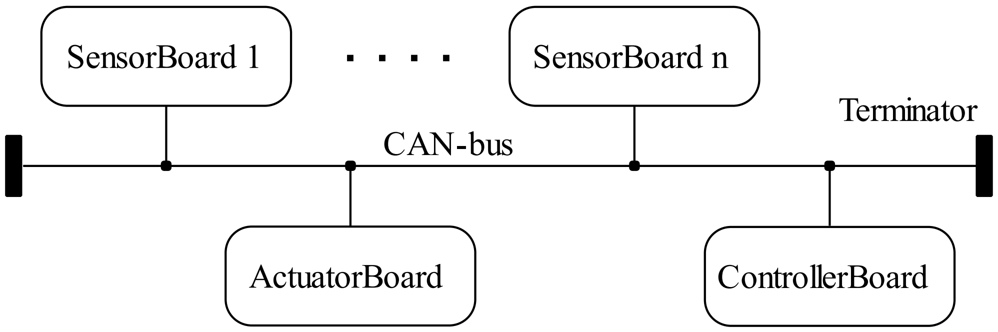


Figure 3.2.: Concept car drive by wire structure.

link® [Mat04] model (see below). The concept car sensory party consists of a remote control receiver (see chapter 3.2.1), wheel speed sensors (see chapter 3.2.2), acceleration sensors (see chapter 3.2.3), rotation sensor (see chapter 3.2.4), and distance sensors (see chapter 3.2.5). There are also other sensors like battery voltage measurement or an emergency receiver that are not taken into consideration in this diploma thesis.

3.2.1. Remote Control Receiver

An essential sensor on the vehicle is the receiver of the remote control. It consists of a standard remote control receiver that outputs a pulse width modulation (PWM) signal with a period of nominal 20ms (the used remote control produces a period of 17ms) and a duty cycle of 5 % to 10 % (see figure 3.3).

An AT90CAN128 microcontroller [Atm08] from the Atmel Corporation samples the PWM signal, converts it to an integer number, and sends it as CAN message over the CAN-bus according to appendix B. The controller has an AVR core with 128 kb of flash memory and four kilobytes of static random access memory [Atm08].

The receiver has two channels:

- First, the channel *throttle* controls the vehicle throttle (see chapter 3.3.1).
- Second, the channel *steering* controls the steering servo (see chapter 3.3.2).

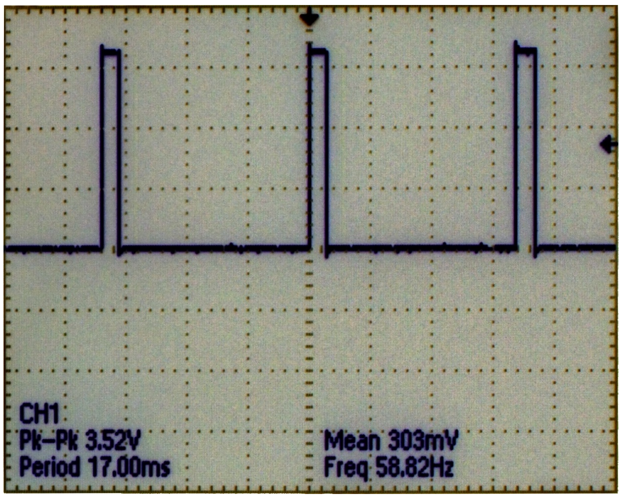


Figure 3.3.: Pulse width modulation signal generated by the remote control receiver.

3.2.2. Wheel Speed Sensors

Each wheel of the vehicle has an attached wheel speed sensor. Black and white code segments attached to the inner side of the wheels are used for determining the angular velocity. The reflective optical sensor *CNY70* (see figure 3.5) has a transistor output [Tem97] that is connected to a pull-up resistor. It receives the signal of the segments. A downstream operational amplifier processes the signal to generate a logic level signal for acquisition on an AT90CAN128 microcontroller [Atm08]. The time between the period of one black to white transitions is measured at a very high resolution (62.5 ns) and the average is sent to the CAN-bus periodically (see appendix B).

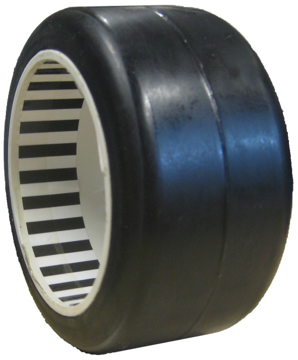


Figure 3.4.: One wheel with its attached black and white encoder strip.

Figure 3.5.: CNY70 reflective optical sensor that is used for rotation speed detection.

3.2.3. Acceleration Sensors

A two-dimensional acceleration sensor of the type *ADIS16006* provides longitudinal and shear acceleration to the Matlab/Simulink® model. The acceleration sensor has the following properties:

The dual-axis accelerometer *ADIS16006* is capable of measuring $-5\,\mathrm{g}$ to $5\,\mathrm{g}$ at a resolution of 1.9 mg at 60 Hz measurement rate [Ana07]. The maximum measurement range is $\pm 8\,\mathrm{g}$. It has a built-in temperature sensor to mask out the temperature drift of the measurement results. Acceleration data is periodically written to the CAN-bus (see appendix B) and used in the Matlab/Simulink® model *calculate_vehicle_speed* (see chapter 4.49) for accurate calculation of the current vehicle speed.

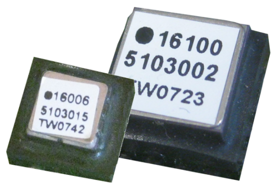


Figure 3.6.: ADIS16100 rotary sensor and ADIS16006 acceleration sensor.

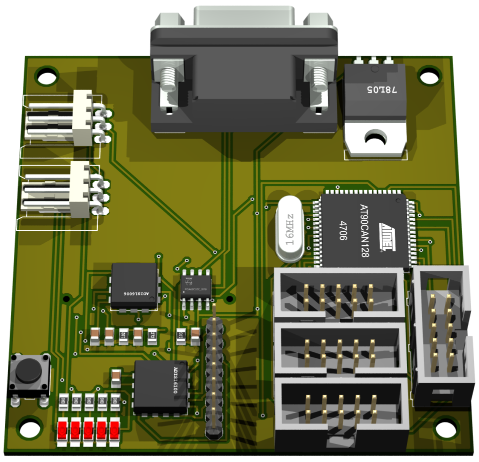


Figure 3.7.: *SensorBoard* that sends acceleration and rotary data to the CAN bus.

3.2.4. Rotation Sensor

The vehicle rotation along the yaw axis is measured using a rotation sensor of the type *ADIS16100*. The dynamic range of the yaw rate sensor *ADIS16100* is $\pm 300\,\frac{\circ}{\mathrm{s}}$ at a resolution of $0.244\,\frac{\circ}{\mathrm{s}}$ [Ana09]. This sensor also provides temperature information for separating out the temperature drift. Both ADIS sensors and a SD card for data logging purposes are connected to a *SPI* (Serial Peripheral Interface) bus on the same *SensorBoard*.

3.2.5. Distance Sensors

Adaptive cruise control systems in the automotive area use radar sensors. For example, LRR (long range radar) systems with $77GHz$ and $10mW$ pulses that achieve a measurement range of up to 140 meters and have a typical opening angle of 4° are used. In our case more convenient ultrasonic sensors of the type *SRF02* from the company *Devantech Ltd* are used. They achieve a measurement range of up to six meters using an ultrasonic wave of 40 kHz [Dav09]. The sensor has an UART and an I^2C interface.

Two of these sensors are used for reducing the probability of sensor break. They are connected to a *SensorBoard* that evaluates both attached sensors and sends the measurement results to the CAN-bus. Two sensors with two different I^2C buses are used to make sure that distance information will still be available even if one sensor or one bus fails.

Figure 3.8.: Two of those *SRF02* distance sensors are used for measuring the distance to the predecessor vehicle.

3.3. Actuators

As the concept car is an x-by-wire system, all actuators are controlled via CAN-bus. A dedicated *ActuatorBoard* receives the CAN messages and generates PWM signals to drive the throttle motor (see chapter 3.3.1) and the steering servo (see chapter 3.3.2). In the near future, a third actuator will be added to the vehicle: hydraulic breaks.

As stated before, the *ActuatorBoard* is capable of working in two different modes. In normal mode the CAN messages of the *SensorBoards* are used for generating the PWM pulses. In contrary, in direct mode the CAN messages of the controller board are evaluated instead.

3.3.1. Throttle

The concept car gains its speed from the brushless motor *1930/9 LK* from *Lehner Motoren Technik*. The motor is air-cooled and drains a maximum of 50 ampere at a voltage of 21 volts and thus consumes 1.05 kW of electrical energy.

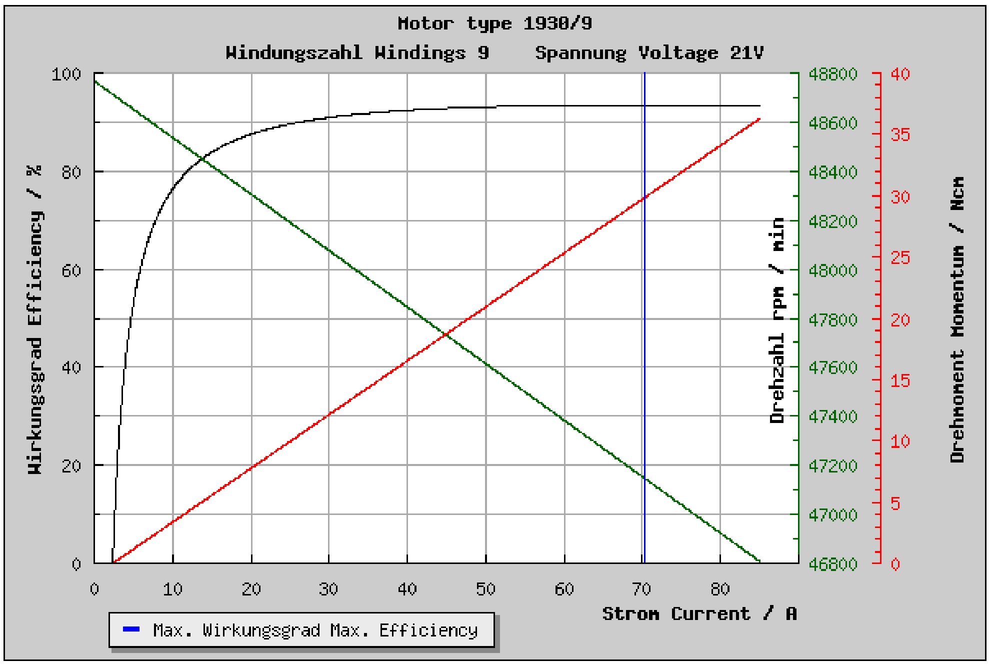


Figure 3.9.: Diagram of the brushless motor *1930/9 LK* from *Lehner Motoren Technik*.

The motor is controlled by the brushless controller *Power JAZZ 63V* from *Kontronik GmbH*:

- The controller is capable of delivering a continuous current of 120 ampere and a peak current of 200 ampere for up to 15 seconds [Kon06].
- It is suitable for a voltage range of 13 volts to 63 volts and can drive motors at a maximum speed of 150 thousand rotations per minute.

The brushless controller gets as input a PWM signal that is - in non-x-by-wire systems - generated by the remote control receiver. In our drive-by-wire case the *ActuatorBoard* generates the PWM signals from the mode-specific (see page 18) CAN messages they receive. The same applies to the steering servo.

3.3.2. Steering Servo

Steering is performed by a steering jumbo-servo that turns both front wheels. The steering servo directly processes the PWM signal that is generated by the *ActuatorBoard*. Steering is not relevant for the adaptive cruise control system and thus the steering system is not examined in detail in this diploma thesis. Other vehicle control systems, for example parking assistance systems, may modify the steering angle but this is not necessary for the ACC system.

3.4. Embedded Controller Board

As shown in figure 3.2, the concept car has an embedded controller board that processes the CAN messages to be able to run Matlab/Simulink® models on the vehicle. This *ControllerBoard* has waste of memory and computational power. Compared to the *SensorBoards* and the *ActuatorBoard* it is considerably faster and all model parts can be executed in one place. First, chapter describes the *ControllerBoard* hardware. Second, chapter explains the software part of this board.

3.4.1. Hardware of the Controller Board

The board is a development board from Olimex called *SAM7-LA2 Development Board for AT91SAM7EA2 ARM7TDMI-S Microcontrollers* (see figure 3.10). The main part of the development board is an Atmel AT91SAM7A2[Atm07] microcontroller that is based on a 32 bit *Acron Risc Machine* 7 (ARM7) architecture from Acron with 16 kb internal SRAM, two built-in UART interfaces, four CAN interfaces, and several other interfaces. The board has the needed hardware to interface UART, CAN, SD-Card, JTAG, and other devices directly. Additionally, it has 4 Mb of external SRAM (two SRAM chips, 2 Mb each) and 1 Mb of external flash memory [Oli08].

3.4.2. Software of the Controller Board

The software part on this *ControllerBoard* consists of a bootloader and of the application software that is to be model-based developed in this diploma thesis.

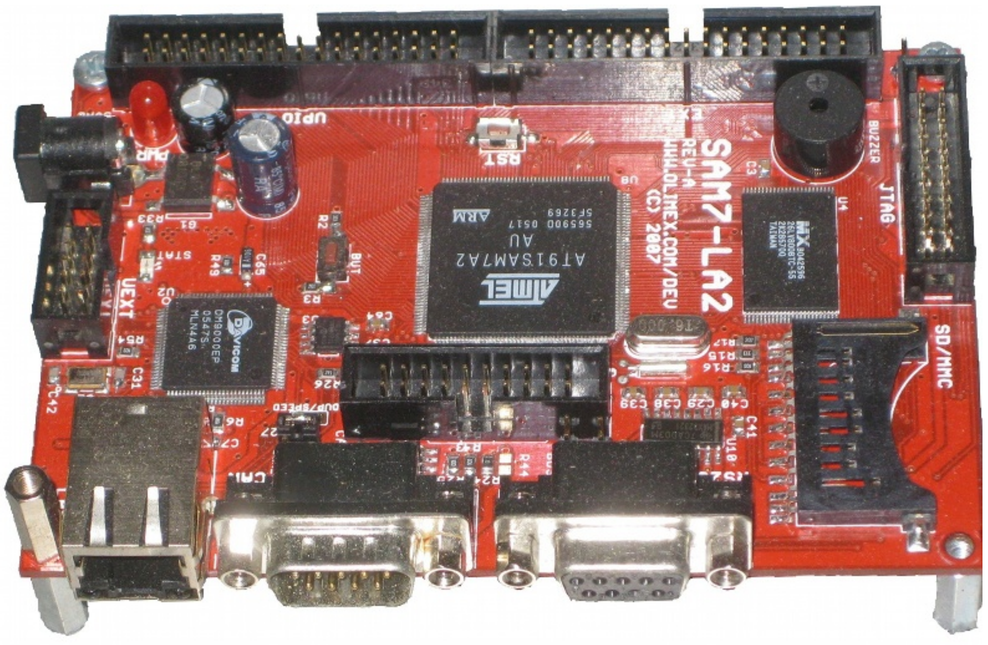


Figure 3.10.: Development board *SAM7-LA2* from *Olimex Ltd.*

The bootloader is located on the external flash memory. It scans for a SD card and checks if it finds an appropriate filesystem with the specific application file on it. If the file is found on the card, it will be copied to the external SRAM and will be executed from there. There is no multi-tasking operating system with syscalls or similar.

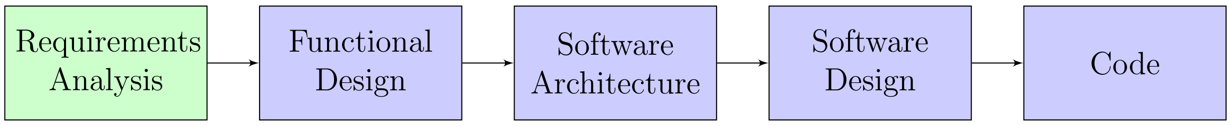
The application code that runs on the *ControllerBoard* is generated out of the Matlab/Simulink® model. It has to be combined with the libraries and drivers of the platform code and operating system code. Combining and compiling the codes is done by the Java program called *SimulinkTarget* (see chapter 4.5.2) and the ARM specific GNU/GCC toolchain [GCC].

4. Model-Based Software Development Process for Embedded Systems

In this chapter, the model-based development process for embedded systems is performed and each step is described in detail: First, the process starts with the *Requirements Analysis* to define what the system should exactly do (see chapter 4.1). It is important to define the border between system and environment. Second, the process continues with the *Functional Design* phase (see chapter4.2) where domain expert knowledge is used to design necessary filters and feedback loop controllers. Third, in the *Software Architecture* phase (see chapter 4.3) software engineers start structuring and refining the system by applying the divide and conquer paradigm. The results of this phase are hierarchic models that define the component of the system. The input of the *Software Architecture* phase are the results of the requirements analysis phase and the models of the functional phase. Fourth, in the *Software Design* phase (see chapter 4.4) the components are further refined until the complexity has reached a reasonable value. The transition of *Software Architecture* phase to *Software Design* phase is smooth. Thus, one can not clearly mark out the boundaries of both phases. Additionally, in the *Software Design* phase, all behaviors of all components have to be defined and platform-specific clues have to be taken into consideration. Finally, after the system is fully modeled, its code is generated (see chapter 4.5).

The advantage of this process is that the last phase, the code generation, is extremely short and one does not have to worry about implementing and simulating controllers in software. In most cases, simulating code would not be possible at all. Additionally, designing and testing controllers can be done by domain experts who are not necessarily able to write code. Therefore, the responsibilities are much better distributed and the resulting system is likely to be of higher quality. Further advantages of model-based development of embedded systems as described in [BST09] are that one has the opportunity to reuse and extend the developed components. The overall of cost- and time-consumption is heavily reduced because errors can be detected and corrected in early design phases.

4.1. Requirements Analysis



The first thing to do for developing an embedded system using the described approach is the *Requirements Analysis*. The analysis is necessary to figure out what the requirements of the system are and what the system should exactly do. But it is not important how the systems completes its task. Sometimes making decisions is essential, but as they reduce the solution space of the system, it should be avoided. For complex systems a requirements management system (like Telelogic DOORS) can be used for the *Requirements Analysis*. In this early phase, it is important to specify the requirements completely, unmistakably, and unambiguously. If the requirements are incomplete, mistakable, or ambiguous and this is realized in a later stage of the development process, the resulting changes could be cost- and time-intensive.

In my case, the requirements are not very complex and thus I will use textual description of the requirements in combination with UML use case diagrams and UML sequence diagrams as described in chapter 2. The use case diagram is the most important diagram type for the requirements analysis phase: On the one hand, it models all possibilities how the user may interact with the system. On the other hand, it defines the boundaries of the system[BJR08].

Figure 4.1 shows the use case diagram of the adaptive cruise control system that has to be developed. One can see from the figure that the adaptive cruise control system has six use cases: *Power on system*, *enabling the controller*, *disabling the controller*, *adjusting the desired speed*, *breaking*, *control speed with headway control*, and *control speed without headway control*. These use cases are described in detail in the following sections.

4.1.1. Powering on the System

The use case *power on system* pictures that the user can power on the system. Initially, the system is not powered and none of the electric circuits is performing any task. By pressing the power switch, the concept car is supplied with energy. After the system is powered,

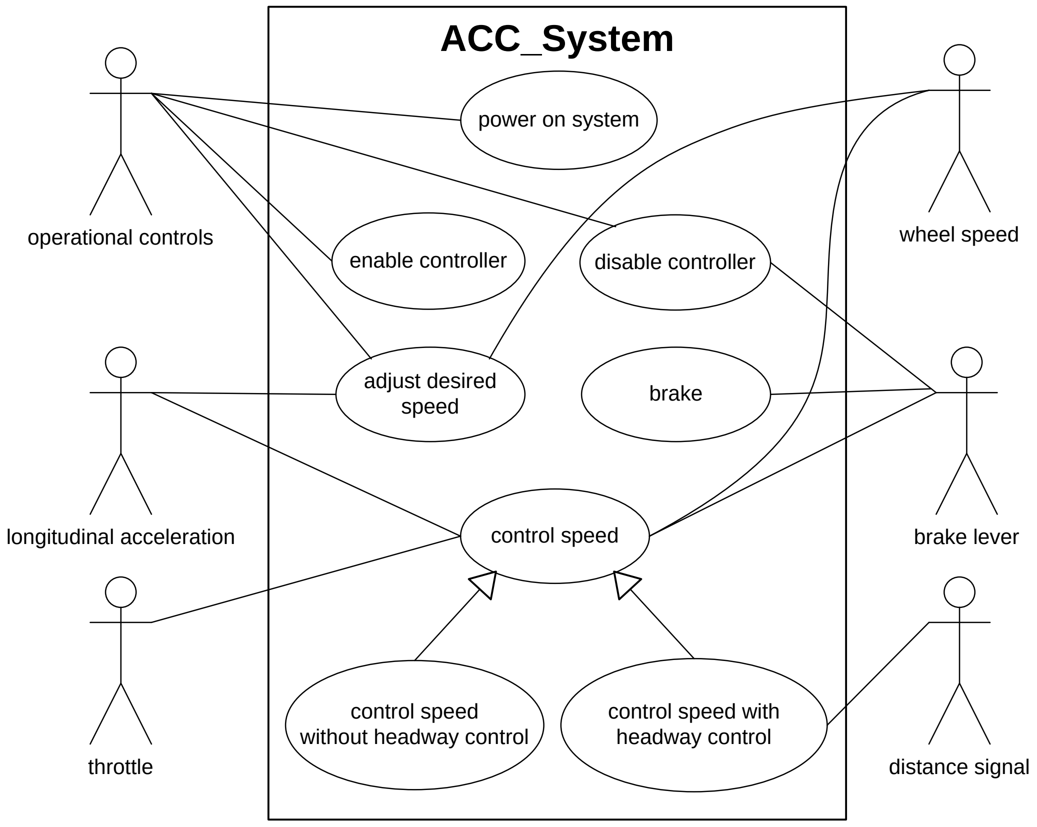


Figure 4.1.: Use case diagram for the adaptive cruise control system.

it is running and capturing the sensory inputs, but it will not control anything unless the controller is enabled.

4.1.2. Enabling the Controller

After the system is powered, the controller is still disabled and does not modify the control messages or the motor reaction. Unless the controller is enabled, the user is able to control the vehicle in normal manner. When the controller is enabled, it gains control over the motor torque and is able to influence the vehicle speed.

4.1.3. Disabling the Controller

On the other side, the user may *disable* the controller again. After the controller is disabled, it will stop interacting with the motor or any other parts of the environment. Thus, normal control over the vehicle is regained. Braking the car will also cause the controller to disable if the controller is enabled.

4.1.4. Adjust Desired Speed

The use case *adjust desired speed* is used to initially setup or later change the desired vehicle cruising speed. The precondition for adjusting the desired speed is that the system is powered and the controller is enabled. Sequence diagram 4.2 describes this procedure in detail. The first step for adjusting the desired speed is the determination of the current vehicle speed. When the driver of the car operates the *set speed* control, the system uses the calculated vehicle speed as a setpoint for the desired vehicle speed. Afterwards, this speed is used for controlling the vehicle throttle.

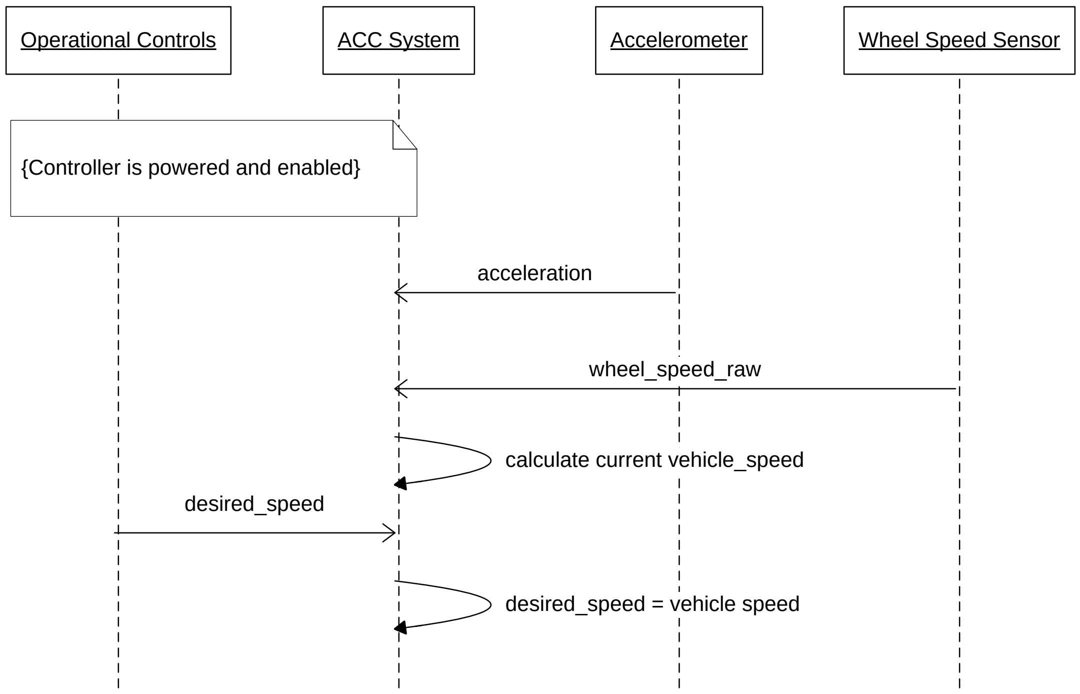


Figure 4.2.: Sequence diagram of the scenario *adjust desired speed*.

4.1.5. Brake

The driver of the vehicle should be able to *brake* at any time. Figure 4.3 shows the sequence diagram of the use case *brake*.

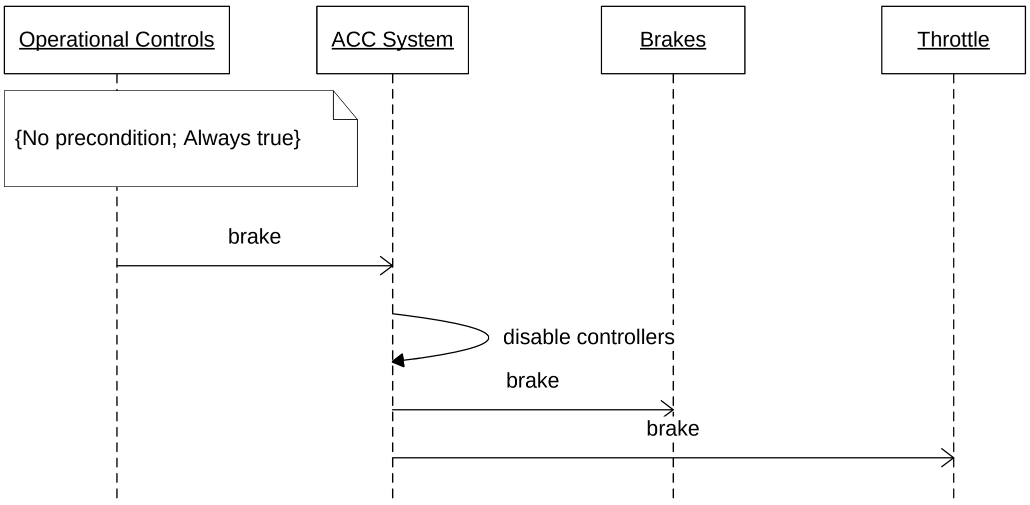


Figure 4.3.: Sequence diagram of the use case *brake*.

The operational control of braking directly influences the motor since (at least at the moment) the motor is used for braking and there are no additional brakes for the front wheels (they are planned as near future extension of the concept car). The system has to make sure that the outputs of the controller, that are directly addressed to the motor controller, do never override the brake signals that are, for the reason mentioned above, also directly addressed to the motor controller for braking purposes.

The system may be in different states of operating modes (cruise control, adaptive cruise control, controller disabled), but when the brake lever is operated, the system has to brake immediately and it automatically has to disable the speed controllers to make sure that the vehicle does not re-accelerate unwanted after braking. After the controller is automatically disabled, the controller may be re-enabled as described in 4.1.2.

4.1.6. Control Speed

The likely most important use case of the system is *controlling speed*. There are two different operational modes for controlling the speed: with and without headway control. Depending on whether there is a predecessor vehicle in front of the concept car and depending on how big the distance between the two vehicles is, one of the sub use cases controls speed with or without headway control will come into operation (see below).

The precondition for controlling speed whether with or without headway control is that the system is powered and the controller is enabled as described in 4.1.2. Furthermore, the desired speed has to be set as described in 4.1.4. In any case a minimum distance of one meter to the predecessor vehicle should be kept. Additionally, the vehicle should never travel faster than 105% of the desired speed.

Control Speed without Headway Control

If all preconditions (4.1.6) in the *control speed without headway control* operation mode are met, the system will behave like a cruise control system. As long as the following additional conditions are met, the system just acts as a simple speed controller:

- There is no predecessor vehicle in front of our vehicle.
- If there is a predecessor vehicle, it will at least be as fast as the desired speed of our vehicle.

Normal cruise control systems just use the vehicles throttle to maintain steady speed and normally they do not interact with the braking system. But in this thesis, as the throttle and the brakes are (at the moment) the same actuator and driven by the same control signal, I also use braking for controlling the vehicle speed as fast as possible towards the desired speed.

Control Speed with Headway Control

If the conditions for speed control without headway control are not met - consequently - if the following conditions are met, the operational mode *control speed with headway control* (see figure 4.5) will be entered:

- There is a predecessor vehicle in front of our vehicle.
- The predecessor vehicle has the same speed or is slower than our car.

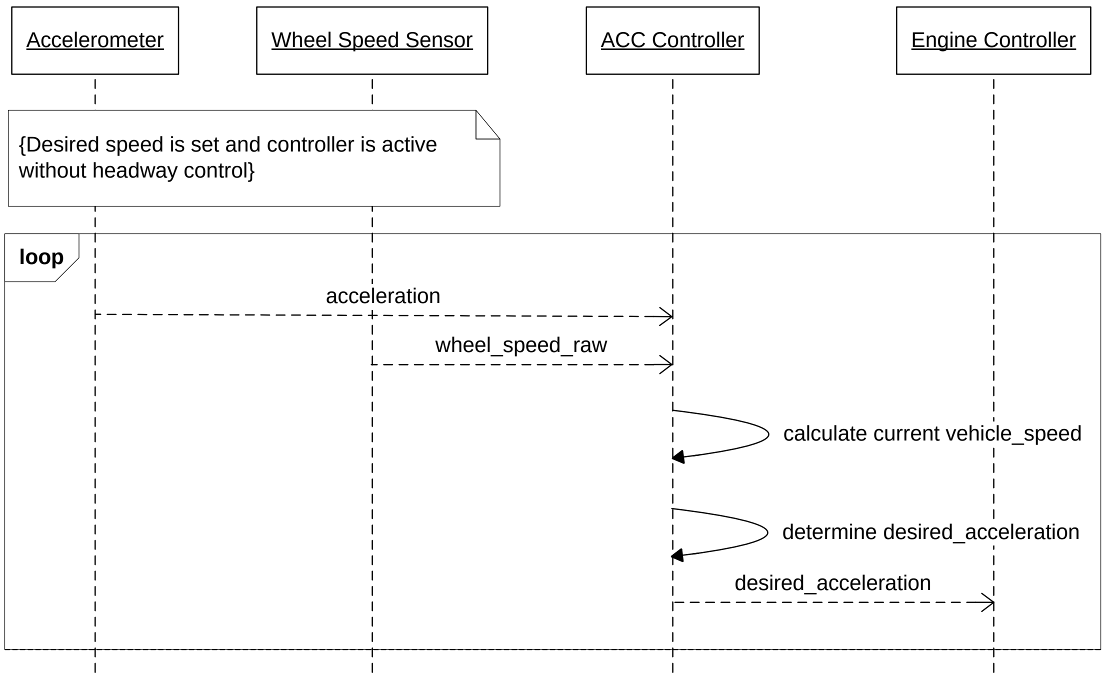

Accelerometer
Wheel Speed Sensor
ACC Controller
Engine Controller
{Desired speed is set and controller is active without headway control}
loop
acceleration
wheel_speed_raw
calculate current vehicle_speed
determine desired_acceleration
desired_acceleration

Figure 4.4.: Sequence diagram of the scenario cruise control (*control speed without headway control*).

The adaptive cruise control system is an extension of the cruise control system. The speed control is extended by vehicle following or spacing control. The spacing between two vehicles should be a time gap between one and two seconds. Normally, braking activity is constrained to a maximum deceleration of $2\frac{m}{s^2}$ to $3\frac{m}{s^2}$ but in this thesis, as the concept car has no occupants and the available distance sensors have a range of maximum six meters, the car may brake with the full deceleration that is available. Thus, there is no need for limiting the braking force to a specific value.

If the predecessor vehicle is lost, the adaptive cruise control system will revert back to conventional cruise control with the driver requested speed.

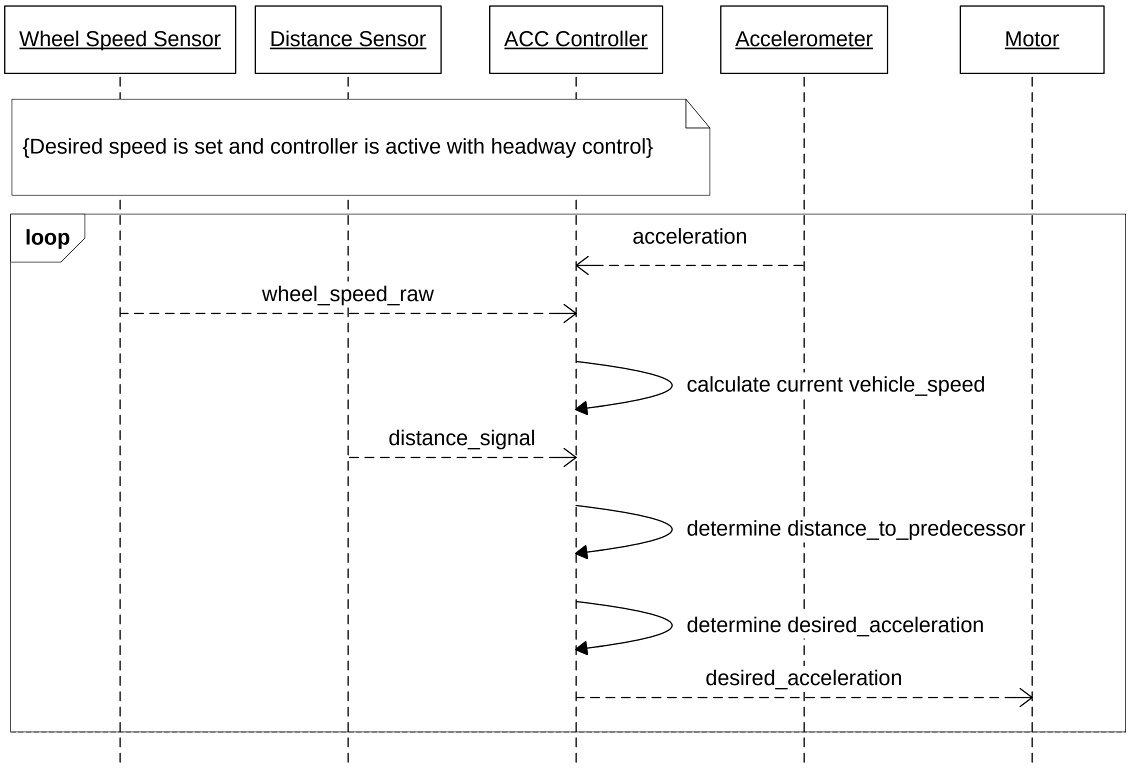
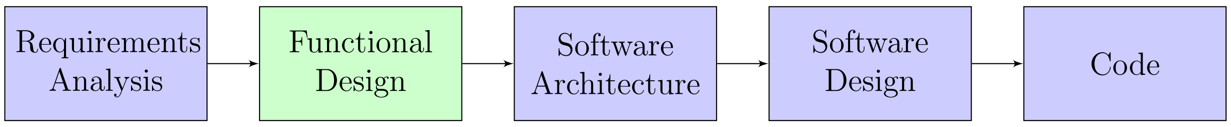

Wheel Speed Sensor
Distance Sensor
ACC Controller
Accelerometer
Motor
{Desired speed is set and controller is active with headway control}
loop
acceleration
wheel_speed_raw
calculate current vehicle_speed
distance_signal
determine distance_to_predecessor
determine desired_acceleration
desired_acceleration

Figure 4.5.: Sequence diagram of the scenario *adaptive cruise control.*

4.2. Functional Design



The techniques used in the *Requirements Analysis* phase are not able of modeling control technology or signal processing tasks. Therefore, in the *Functional Design* phase the functional behavior of the system is focused. The *Functional Design* phase is task of domain experts. Its input are the requirements and the outputs are controllers for all closed loops of the system. Additionally, filters are implemented if preprocessing of sensory data needed.

Block diagrams, for example constructed in Matlab/Simulink® [Mat04], are ideal for modeling continuous behavior. The *Functional Design* phase should find suitable controller parameters and should optimize them by extensive use of simulation.

As an adaptive cruise control system is developed in this thesis, the first thing that is needed in the *Functional Design* phase is a vehicle plant (see chapter 4.2.1). The plant is needed for simulation. Thus, no other controllers can be designed before the plant is known. Second, as the wheel speed sensors generate noisy sensory data a filter is constructed for converting the raw data into a reliable speed value in SI units (see chapter 4.2.2). The whole design uses SI units to avoid errors that are caused by different units of the exchanged data. Additionally, SI units have the advantage that one can easily understand the values that are produced while simulating and debugging the system. In chapter 4.2.5, a speed controller is designed. Before describing the speed controller, chapters 4.2.3 and 4.2.4 describe general background of control loop theory and of PID controllers. The last step of the *Functional Design* phase is the design of the distance controller in chapter 4.2.6.

4.2.1. Derivation of the Vehicle Plant

As the whole model including all controllers has to be simulated, the vehicle plant is needed first. The plant should be as precise as possible and should map the reality as far as possible to be sure that the simulation results match the real behavior. One is never able to model a plant that matches the reality by 100 %, but as the used controllers are very robust, this is no problem. Without any form of plant, no controller can be developed. Thus, finding the plant is the first step for controller design.

Physical Derivation of the Plant

Let x be the distance which the vehicle has covered, the unit of x is meters and the derivation of x with respect to time t is the vehicle speed measured in meters per second:

$$v = \dot{x} = \frac{d}{dt}x$$

The acceleration of the vehicle is the derivation of the speed with respect to time and is measured in $\frac{\mathrm{m}}{\mathrm{s}^2}$:

$$a = \dot{v} = \ddot{x} = \frac{d^2}{dt^2}x$$

To find the plant of the vehicle, one needs to figure out all forces which act upon the vehicle[Kra08]. According to Newton's second law of motion the resulting sum F of all forces is directly related to the acceleration a of the car of the mass m. The unit of m is kilogram:

$$F = m \cdot a = m \cdot \ddot{x} \Rightarrow \ddot{x} = \frac{F}{m}$$

1. The most important force F_M is the force that results of the torque generated by the motor. The drive train of the model car consists of motor, transmission, differential, driving shaft, and wheels. F_M will be positive if the motor controller gets the signal for accelerating and it will be negative if the controller gets the signal for braking.

   $$F_M = \alpha \cdot \tau_{\max}(rpm) \cdot \frac{44}{20} \cdot \frac{77}{20} \cdot \frac{1}{r}$$

   The maximum torque of the motor $\tau_{\max} = 0.297\,\mathrm{Nm}$ is scaled by the coefficient α. This coefficient is the throttle value set by the user and can be considered as normalized torque of the motor (-1 to 1), which is set in the motor controller. The maximum torque $\tau_{\max}(rpm)$ depends on the rotations per minute of the motor axis and is a fixed function that only depends on the used motor. The $\tau_{\max}(rpm)$ function will be deduced later (see page 39).

   The fractions $\frac{44}{20}$ and $\frac{77}{20}$ are two transmissions, first the motor pinion to a bigger gearwheel and second a V-belt. The last factor ($\frac{1}{r}$) converts the torque to a force by dividing by the wheel radius $r = 62$mm.

   After the transmission ratio is known, one is also able to calculate the rotations per minute of the motor:

   $$rpm = \frac{v}{2 \cdot \pi \cdot r} \cdot \frac{44}{20} \cdot \frac{77}{20} \cdot \frac{60\,\mathrm{s}}{1\,\mathrm{min}}$$

   The first fraction calculates the wheel rotations per second from the vehicle speed $v = \dot{x}$ and both following fractions calculate the motor rotations per second by taking the transmission into consideration. The last factor just converts the value to the expected unit. Thus, the force F_M results to:

   $$F_M = \alpha \cdot \tau_{\max}\left(\frac{\dot{x}}{2 \cdot \pi \cdot r} \cdot \frac{44}{20} \cdot \frac{77}{20} \cdot \frac{60\,\mathrm{s}}{1\,\mathrm{min}}\right) \cdot \frac{20}{44} \cdot \frac{20}{77} \cdot \frac{1}{r}$$

2. The second force to examine is the force F_Θ, which results of the slope of the ground. The ground has the slope Θ and according to Newton's second law of motion the force to overcome the ground gradient is:

   $$F_\Theta = m \cdot g \cdot \sin\Theta$$

   Where $g \approx 9.81\frac{m}{s^2}$ is the gravitational acceleration and $\sin\Theta$ is the vertical part of the gravitational acceleration that has to be taken into consideration.

3. Wherever mass rolls on a surface, there exists a rolling resistance or rolling friction that results into a rolling force that is directed against the movement of the vehicle. This force is caused by the deformation of the surface and the objects. The friction force F_R is proportional to the gravitational force of the vehicle [Kra08]:

   $$F_R = c_R \cdot F_N = c_R \cdot \cos\Theta \cdot m \cdot g$$

   The proportional constant is called roll coefficient and depends on the involved materials. Table 4.1 shows a list of rolling coefficients of different materials on several surfaces. In our case a value of approximate 0.3 can be used as it is the value for

c_R	Description
0.0002 to 0.0010	Railroad steel wheel on steel rail
0.0025	Special Michelin marathon tires
0.005	Tram rails standard dirty with straights and curves
0.0055	Typical BMX bicycle tires used for solar cars
0.006 to 0.01	Low-resistance car tires on smooth road
0.010 to 0.015	Ordinary car tires on concrete
0.020	Car on stone plates
0.030 to 0.035	Ordinary car tires on tar or asphalt
0.055 to 0.065	Ordinary car tires on grass, mud, and sand

Table 4.1.: Table of compare-rolling resistance coefficients like described in [Kar00].

   ordinary car tires on tar or asphalt.

4. The air resistance force F_a is proportional to the square of the speed $\dot{x}$:

   $$F_a = \frac{\rho_a}{2} \cdot c_w \cdot A \cdot \dot{x}^2$$

   The proportionality factor is the product of the density of the air $\rho_a \approx 1.2\frac{kg}{m^3}$, the drag coefficient c_w and the projected abutting face A of the vehicle in m^2. According to figure 4.6, the drag coefficient can be expected to have a value of about $c_w \approx 0.1$. The projected abutting face is approximately $A \approx 0.07m^2$ for the concept car. Thus, the coefficient for the air resistance can be calculated and the air resistance force results to: $F_a = 0.0042 \cdot \dot{x}^2$.

After determining all forces, the differential equation for the movement results to:

$$m \cdot a = F_M - F_\Theta - F_R - F_a$$

$$\Rightarrow m \cdot \ddot{x} = \alpha \cdot \frac{\tau_{\max}(rpm)}{r} \cdot \frac{20}{44} \cdot \frac{20}{77} - m \cdot g \cdot \sin\Theta - c_R \cdot \cos\Theta \cdot m \cdot g - \frac{\rho_a}{2} \cdot c_w \cdot A \cdot \dot{x}^2$$

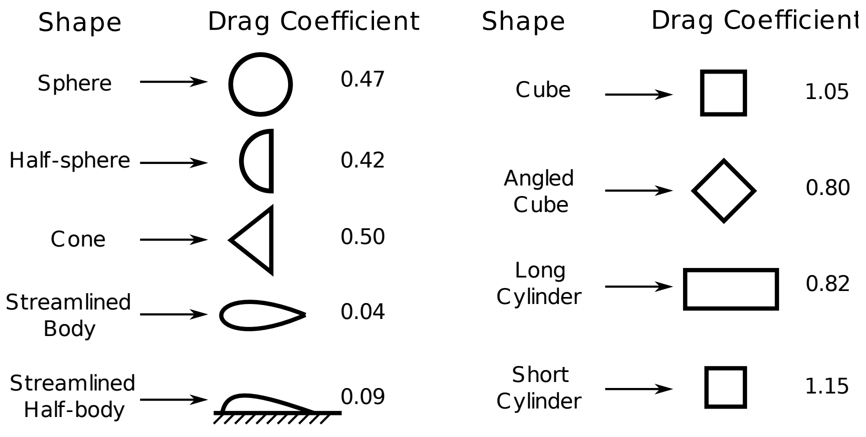


Figure 4.6.: Different drag coefficients.

$$\Rightarrow \ddot{x} = \frac{1}{m} \cdot \left[\alpha \left(\frac{\tau_{\max}(rpm)}{r} \cdot 0.118 \right) - \left(\frac{\rho_a}{2} \cdot c_w \cdot A \right) \cdot \dot{x}^2 \right] - g\left(\sin\Theta + c_R \cdot \cos\Theta\right)$$

Finally, this formula of the vehicle plant can be modeled in Matlab/Simulink$^{®}$ (see figure 4.7). Most of the dataflow connections between Matlab/Simulink$^{®}$ blocks have annotations for easier comprehensibility. Input 1 of the model is the normalized motor torque α (range -1 to 1) that will immediately be converted to the nominal motor torque with unit Nm by multiplying with the maximum torque $\tau_{\max}(rpm)$ at the current motor rotatory speed. The rotations per minute are calculated from the vehicle speed output of the vehicle plant as described on page 36.

The two transmissions of 20:44 and 20:77 result to an additional gain block of value 7.7. After the last gain block of value 16.13 ($\frac{1}{r}$) the edge holds the motor force in the SI unit Newton. Now, the air resistance force will be subtracted, the force will be divided by the vehicle mass $m = 8.3\,\text{kg}$ and finally, the gravitational and roll acceleration will be subtracted. After doing all those steps, the edge holds the vehicle acceleration that is integrated to get the vehicle speed. The speed could be integrated again to gain the distance that our vehicle has covered.

Finally, I have to annotate that of course, the nominal plant of the vehicle that was derived in this section differs from natural plant. There are still small factors that were not taken into consideration like the rotating mass and the rotating friction of the wheels, gearwheels, differential, V-belt, and the motor itself. But as all other factors have a greater impact on the movement of the car, those factors may be neglected and will be eliminated by the robust controllers [Kra08].

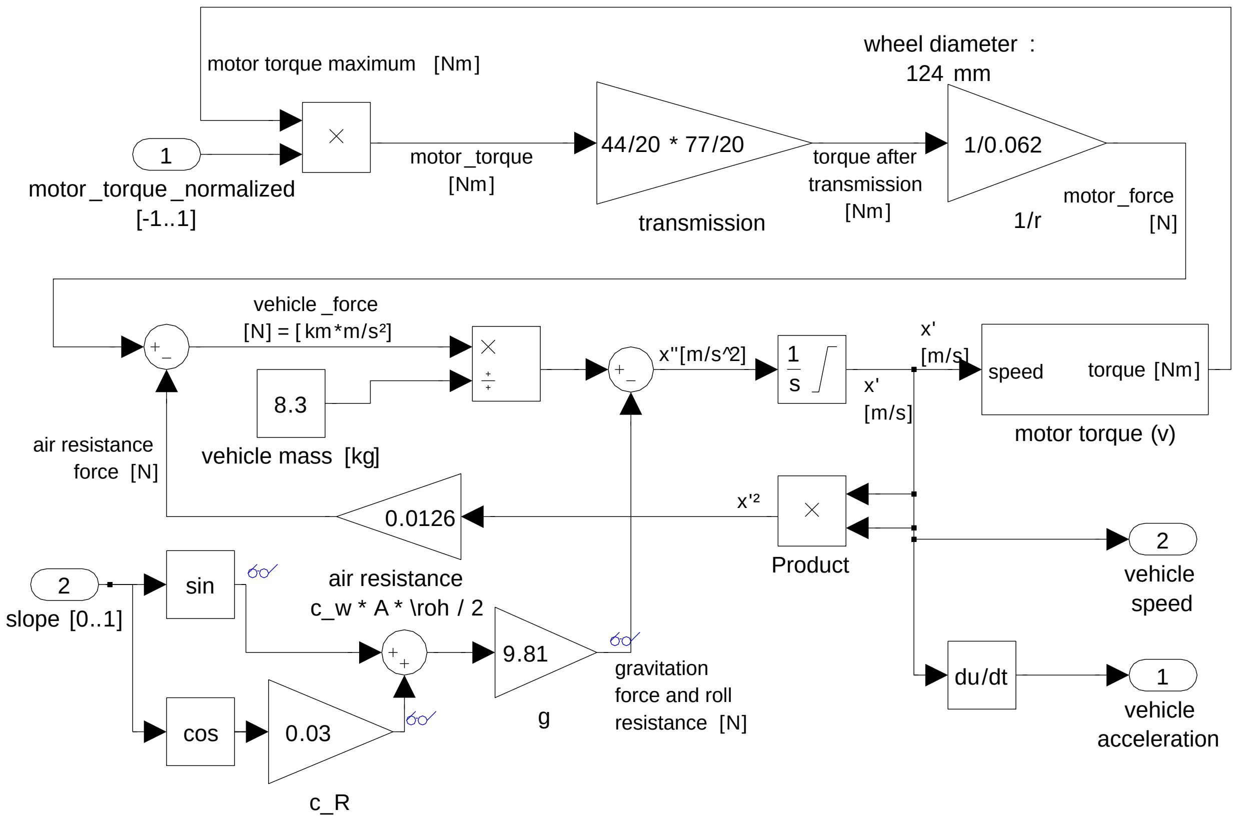


Figure 4.7.: Matlab/Simulink® model of the vehicle plant.

Motor Torque

As described on page 36, the maximum available motor torque is needed for a realistic vehicle plant. Regardless which type of motor is used, all motors have different torques for different rotation speeds. On the concept car, a brushless motor is used as described in chapter 3.3.1.

Because brushless motors use a permanent magnet on the rotor, and user wire windings on the stator, there is no need to use brushes and a commutator to switch the polarity of the voltage on the coil [Hug08]. Instead a controller is needed (see chapter 3.3.1) for alternating the current in the coils to continuously rotate the motor. The rotatory speed of brushless motors is proportional to the frequency of this alternating current. The lack of brushes means that these motors require less maintenance than the brushed direct current motors.

In general brushless motors behave like approximated in figure 4.8. In low rotation areas brushless motors can provide nearly full torque (about 0.3 Nm in this case) and in higher operating areas the maximum torque decreases [Hug08]. When reaching the absolute full maximum rotary speed (48700 $\frac{1}{\text{min}}$ in this case), the motor stops generating torque at all. As stated before, diagram 4.8 just shows the general behavior of brushless motors in direction.

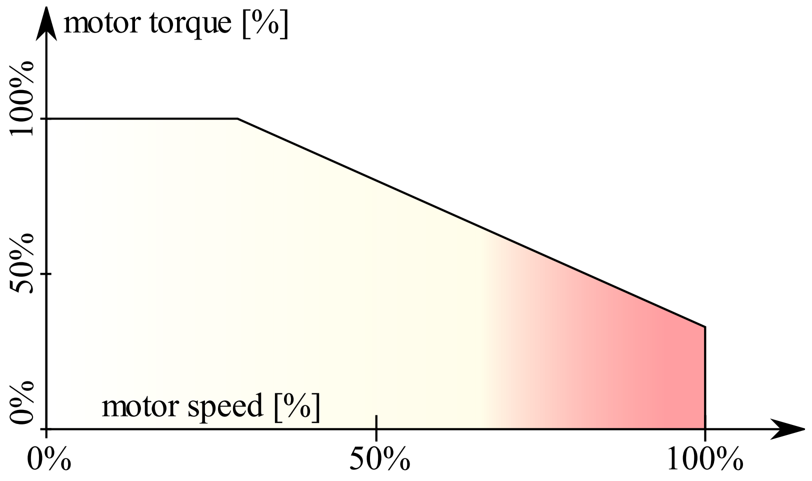


Figure 4.8.: General approximated rotational speed-time-diagram for brushless motors.

The accurate rotary-speed-torque-diagram for the concept car brushless motor cannot be identified. Measuring the complete curve of the uses motor would have consumed too much time. For this reason, the diagram for a really similar motor is used. The curve is shown in figure 4.9. It is the diagram of the brushless motor *Novak 3.5 R* which is also a racing motor with similar torque and rotations per minute. The curve of this diagram will be used for approximating the torque of the deployed brushless motor. Calculating the rotary speed of the motor is done using the vehicle speed value by taking into consideration wheel diameter and transmissions (see page 36).

Finally, figure 4.10 shows the Matlab/Simulink® block diagram for calculating the maximum available motor torque. First, the vehicle speed that is measured in $\frac{\text{m}}{\text{s}}$ is converted in wheel rotary speed in min^{-1} by considering the wheel diameter. Afterwards, the value is multiplied by the transmission ratio to get the motor rotary speed in min^{-1} that is fed into a lookup table. The lookup table represents the brushless motor torque curve from figure 4.9 and returns a normalized value in the range of zero to one. This normalized value is multiplied with the motors maximum torque for finally getting the rotary-speed dependent maximum torque.

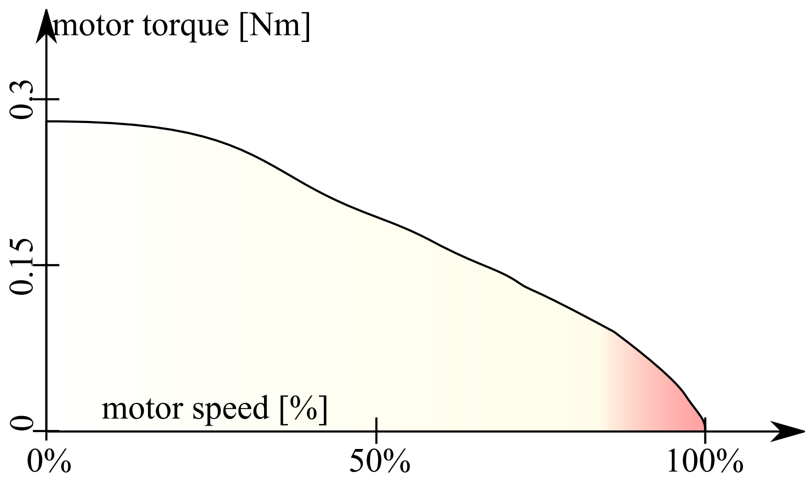


Figure 4.9.: Rotary-speed-torque-diagram for the brushless motor that is used on the concept car.

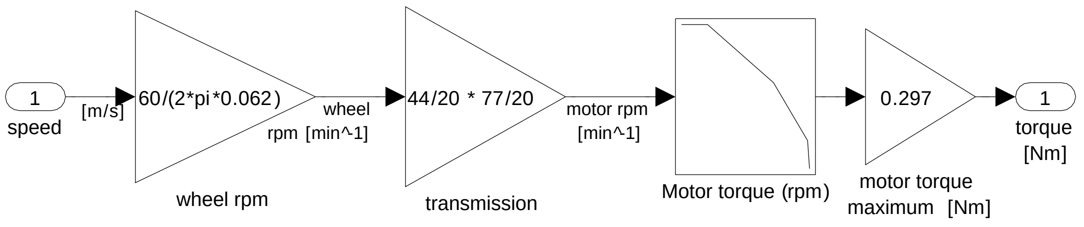


Figure 4.10.: Matlab/Simulink® model that calculates the available motor torque in dependence upon the vehicle speed.

To sum up, the vehicle plant that is used for simulation and for finding controller parameters is fully specified. As stated before, the next step in the *Functional Design* phase is designing the filter for the wheel speed sensors.

4.2.2. Filter for the Wheel Speed Sensors

The signals that are generated by the wheel speed sensors are very noisy and need additional processing for getting reliable values in SI units. Improving the quality of the sensor signals is an important step because the subsequent controllers rely on these values. As described in section 3.2.2 on page 20 the four wheel speed sensors just measure the time between two ticks of the wheel encoder strips. The time is measured very accurate at a resolution of 62.5 ns. The only processing that is performed on the results of the time-difference measuring is building the average over 20 ms which is the measurement interval of the *SensorBoard* that sends the wheel speed CAN messages.

The CAN messages can be captured using a CAN-to-USB interface and can be imported into Matlab/Simulink®. The CAN-to-USB interface was designed and constructed by some fellow students and me while working for the formula student racing team KaRaT (Kaiserslautern Racing Team, `http://fs-kl.de`). Because the interface was implemented by ourselves, it was simple to add the Matlab/Simulink® import functionality. Using the small Matlab/Simulink® model in figure 4.11, the sensory data can be plotted and the *calculate_wheel_speed* component can be tested for getting the best results out of the noisy sensory raw data.

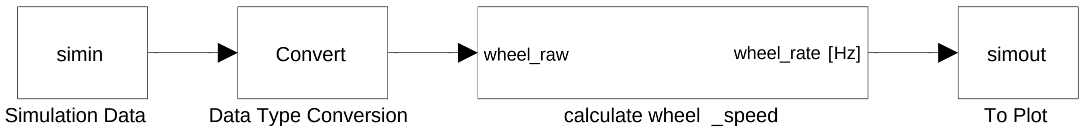


Figure 4.11.: Simulation model for the wheel speed sensors.

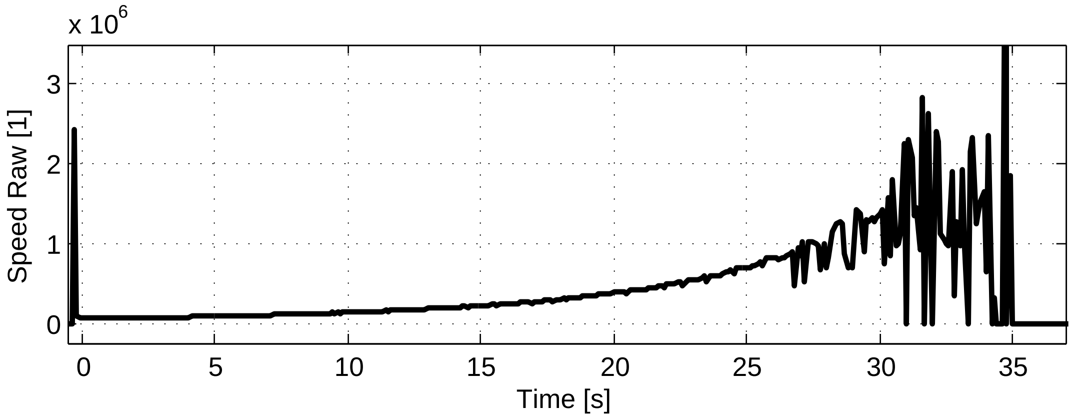


Figure 4.12.: Unprocessed wheel speed raw data.

Figure 4.12 shows the unprocessed wheel speed data for a wheel that spins at a specific speed without touching the ground and rolling off slowly. As one can easily see, the data is very noisy and sometimes there is no data at all in the 20 ms measuring interval for low wheel speeds.

The biggest problem when evaluating the wheel speed sensors is that one can not decide if the wheels are still spinning really slowly or if they are standing. If the wheel spins slow enough, the sensor will not detect at least one black-white transition of the encoder strip

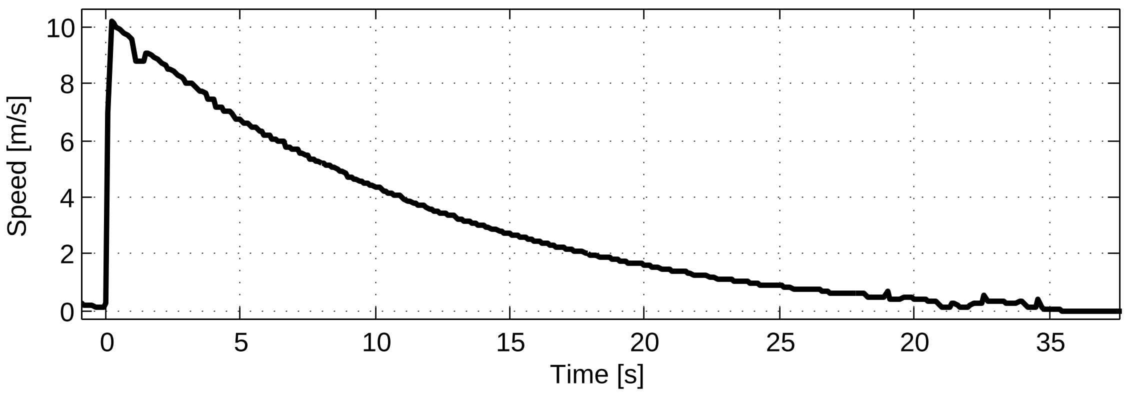


Figure 4.13.: Processed wheel speed data after conversion to rotations per second.

in an appropriate time span. This problem will occur if the black-white transitions takes more than 20 ms which means more than the period of the wheel speed CAN message.

After trying different filters, I figured out that an infinite impulse response filter (see figure 4.14) with some specific switch cases would do a good job converting the raw wheel speed data (time between black-white transitions) to the rotating rate of the wheel (measured in Hertz). Caused by the limited scope of this diploma thesis, there was not more time for further improvements of this filter.

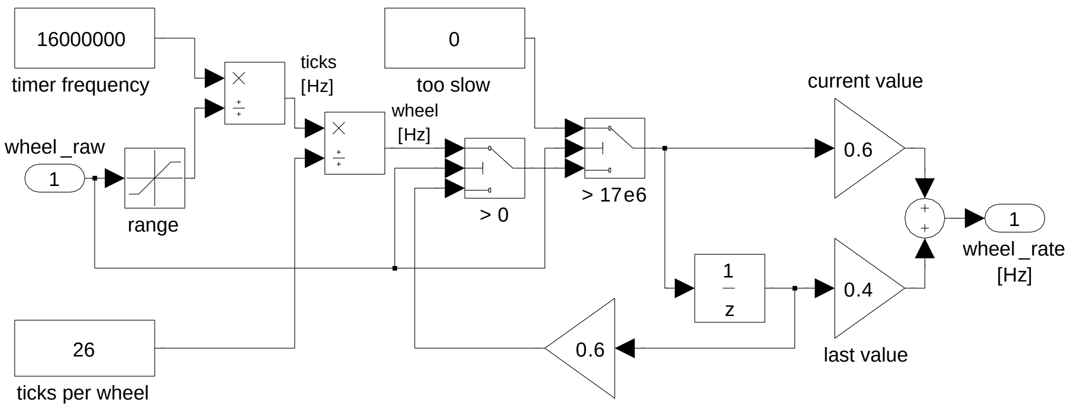


Figure 4.14.: Matlab/Simulink® model (*calculate_wheel_speed*) for processing the raw wheel speed sensory data.

Multiplying the wheel rate with the circumference of the wheel results in the vehicle speed in $\frac{m}{s}$ for this one wheel.

At this point of *Functional Design*, all necessary inputs (sensors) and outputs (actuators) of the controller are available:

- Distance values arrive directly over the CAN-bus and just need to be multiplied by 0.01 for converting the unit of the values from centimeters to meters (3.2.5).
- Wheel speed sensors are provided by the filter of this chapter.
- Longitudinal acceleration is also directly available on the CAN-bus (3.2.3).
- Motor torque can directly be controlled over the CAN-bus. The *ActuatorBoard* (3.3.1) receives the messages and controls the motor.

As all inputs and outputs are available and the plant is known, one can start developing a speed controller and a distance controller right away.

4.2.3. General Control Loop Theory

For controlling properties of a system, feedback loops are used. Figure 4.15 shows a simple standard control system. The input r of the system is called setpoint. The output y is the control variable. The output is measured (y_m) and the difference between input and measured output is calculated. It is called control difference $e = r - y_m$. The controller tries to regulate the control value u to a value that the controller difference e gets minimal $e \to 0$ [Lun08].

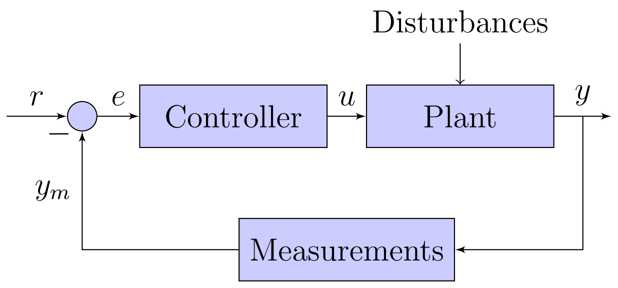


Figure 4.15.: A simple generic control loop.

The plant does not need to be known in detail and the measurements do not have to be ideal ($y_m = y \Leftrightarrow$ transfer function $= 1$) as long as the controller is robust. Furthermore, the influence of disturbances on the plant and on the measurements are minimised by the controller. Disturbances in the case of adaptive cruise control systems or in the automotive area in general may be slope, wind, other friction forces, or the load of the vehicle.

A commonly used controller for many applications is the proportional-integral-derivative (PID) controller that is described in the next section and will also be used in this diploma thesis for controlling speed and distance.

4.2.4. PID Controller

A proportional-integral-derivative controller is a generic control loop feedback mechanism that is widely used in all application domains [Lun06]. This feedback mechanism is used in this diploma thesis because of its leading advantages and the union of advantages of other controllers:

- The most important property of controllers that are used in closed loop feedback is stability. Stable controllers are controllers that do not exceed an output range after a specific point of time. In particular, the stable closed loop does not start oscillating for a long time. The PID controller will be stable with appropriate parameters,
- Taking a PD-controller into consideration, the PID controller inherits an important property: speed. The derivative part of the PID controller ensures that the controller will react quickly on changes of the input difference.
- The integral part of the controller ensures that the position error of the controller for long control times will in any case approximate zero. This is a huge improvement over a P or a PD-controller because both types have position errors.
- All of those properties make the PID controller a very robust and fault-tolerant controller.
- Last but not least, the PID controller's simple structure can easily be modeled in modeling tools like Matlab/Simulink® because just few blocks are used for getting the desired functionality.

Figure 4.16 shows the block diagram of a generic PID controller. The input e is the control error (or control derivation)) and the controller output u is called control value. $e = r - y'$ is the difference of set value and measured value and the output u is the set value that

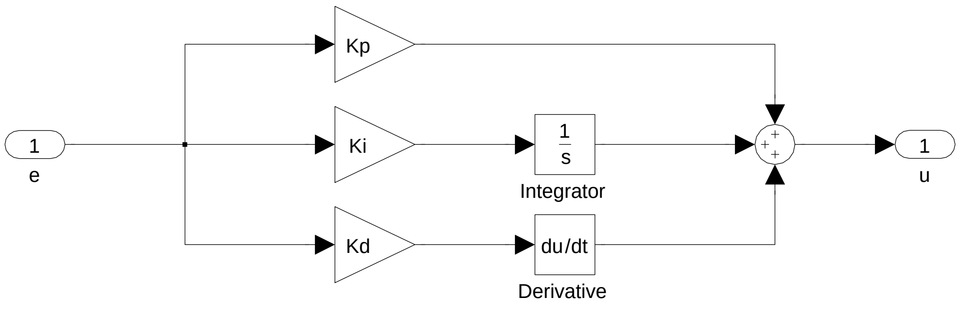


Figure 4.16.: PID controller block diagram.

will be set in the actuator. One can easily see that the controller is separated into three parts:

- First, the proportional part that multiplies the control error with the proportional constant K_p.
- Second, the integral part K_i to prevent the position error.
- And third, the derivative part K_d that increases the reaction speed of the controller.

All parts are added resulting to the control value u. Thus, the PID controller can be represented by the following equation:

$$u(t) = K_p \cdot e(t) + K_i \cdot \int_0^t e(\tau)d\tau + K_d \cdot \frac{de(t)}{dt}$$

Sometimes controllers of this type are represented with a formula that includes time constants:

$$u(t) = K_r \cdot \left(e(t) + \frac{1}{T_i} \cdot \int_0^t e(\tau)d\tau + T_d \cdot \frac{de(t)}{dt} \right)$$

In this thesis, the first representation is used. If time constants are determined, they are instantaneously converted to the PID-coefficients using the following conversion formulas: $K_p = K_R$, $K_i = \frac{K_R}{T_i}$, and $K_d = K_R \cdot T_d$.

To illustrate the impact of different PID controller parameters, figure 4.17 shows a PID controller in a closed loop. Initially, the setpoint is set to zero and after 50 seconds a step to the setpoint five is performed, while at time point 150 seconds, the setpoint is reduced to one.

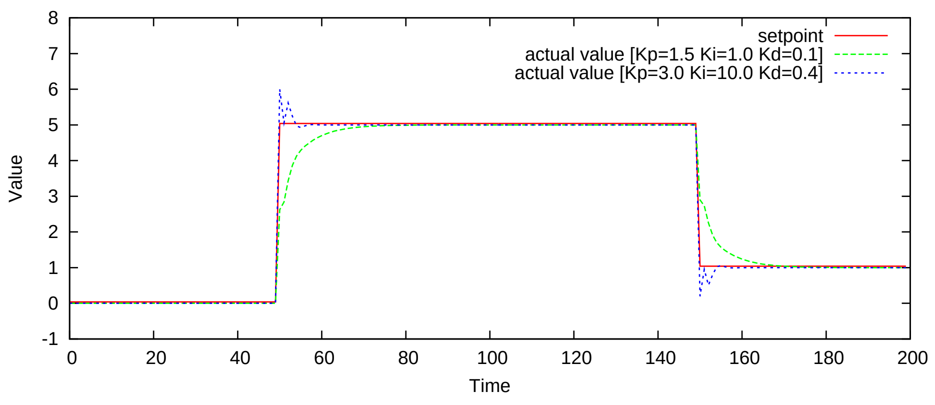


Figure 4.17.: PID controller response for different coefficients.

The graph shows two different PID controller setups:

$$K_1 = \begin{pmatrix} K_p \\ K_i \\ K_d \end{pmatrix} = \begin{pmatrix} 1.5 \\ 1.0 \\ 0.1 \end{pmatrix} \quad \text{and} \quad K_2 = \begin{pmatrix} 3.0 \\ 10.0 \\ 0.4 \end{pmatrix}$$

In the step response of setup K_2 (blue graph), one can easily see that the controller generates overshoots. The control value overshoots by an amount of 20 % which would be too much in case of an adaptive cruise control system. Assuming a setpoint of $100 \frac{\text{km}}{\text{h}}$, the car would speed up to $120 \frac{\text{km}}{\text{h}}$ before slowing down to $100 \frac{\text{km}}{\text{h}}$ again. For ACC systems, the controller with setup K_1 would be more sufficient because it has no overshoot at all, but it has the disadvantage that reaching the desired value consumes more time. Consequently, finding appropriate controller coefficients is an important topic in controller design.

According to [Chr05], for determining the controller coefficients K_p, K_i, and K_d different approaches are known:

- The **manual** method does not require any math, but requires experienced personnel. Several rules exist for how the closed loop reacts on change of the controller, e.g. higher values for K_p imply improved dynamic behavior, but smaller values for K_p reduce the position error. If the K_d part is too high, the system will get instable and if the K_i part is too low, the system will keep its position error. Knowing those rules experienced personnel is able to manually tune the controller.

- In this thesis the **Ziegler–Nichols** method will be used because not much experience is needed and the parameters can be deduced by simulation. There is no need to test parameters in the real system. It is a proven online method and does not require that much experience like the manual method. In brief, this method assumes the controller to be a pure proportional controller. The proportional constant is increased until the systems gets critical stable (i.e. starts oscillating) [Chr05]. Using this critical proportional coefficient and the associated oscillating frequency all parameters can be calculated. This method is described in detail in chapter 4.2.5.

- In real closed loops that are not just simulated sometimes it is dangerous to provoke a critical stable oscillation as used in the Ziegler-Nichols method to determine the parameters. In this case and in the case of systems with bigger signal delays, the **Chien-Hrones-Reswick** method is suitable. The method uses the step response, the delay time T_u, and the compensation time T_g of the system for determining the controller parameters. As this method is not used in this thesis, it is not described in detail.

- **Mathematical** PID loop tuning induces impulses into the system, and then uses the controlled system's frequency response to design the PID loop values. The mathematical tuning method is recommended for loops with long response times (e.g. minutes) because modifying parameters and re-testing the loop will consume too much time.

- According to [Chr05], nowadays industrial applications use PID tuning and loop optimization **software** to ensure consistent results.

Anti-Windup Algorithm

An important part, that no digital controller with integral amount should be missing, is a so-called anti-windup algorithm. When the actuator saturates, which is the case for high control deviations, but the deviation still remains, then the PID controller integral amount will continuously keep integrating the error. This results in high integrated error values and is called integrator windup.

An anti-windup algorithm is used to prevent the integrator windup. Plenty of different approaches exist and four of them are presented in [CB95]:

- **Conditional Integration:** Depending on different conditions such as controller error and controller output the integration of the error is switched on and off.

- The **Limited Integrator** method is the simplest approach: Integration is just performed until a specific value is reached.

- **Tracking Anti-Windup** is the classic approach. The structure is shown in block diagram 4.18. This approach is used in this diploma thesis because the integrator windup limit is handled in combination with the actuator saturation. If the overall controller output exceeds a specific maximum (saturation of the actuator), the exceeding amount (unsaturated minus saturated controller output) will simply be subtracted from the integrator input.

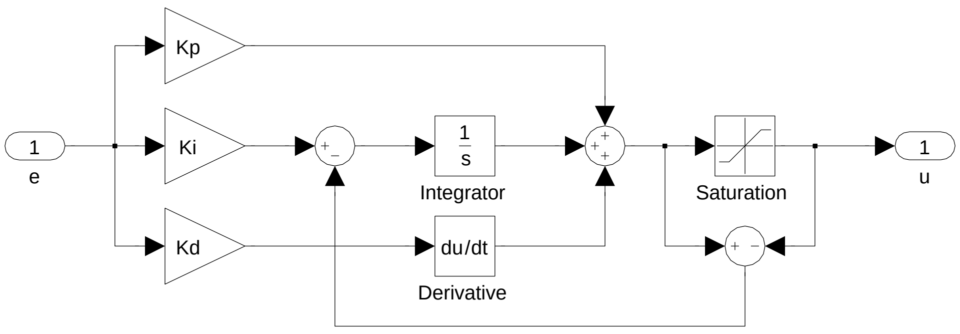


Figure 4.18.: A saturating PID controller with tracking anti-windup algorithm is used in the adaptive control system for controlling speed and distance.

4.2.5. Control Speed

Figure 4.19 shows the Matlab/Simulink® model for controlling the speed of the vehicle and simulating the used PID controller.

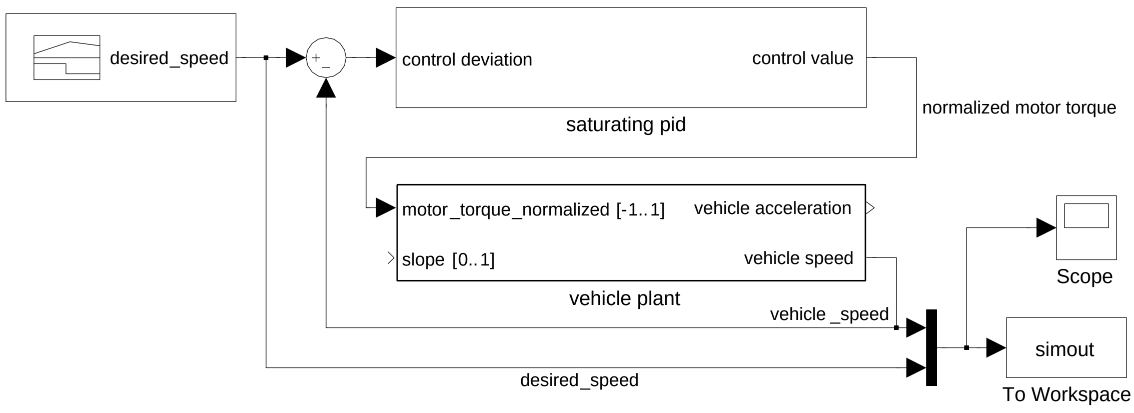


Figure 4.19.: Closed loop for finding the speed controller PID parameters.

The desired speed for simulation is generated by a signal builder component. First, the control difference is calculated as difference of *desired_speed* and *vehicle_speed*. A saturating PID controller (see figure 4.18) is attached to this controller difference. The output of this PID controller directly represents the motor torque and is fed into the plant of the vehicle as shown on page 39. The acceleration output of the vehicle plant is ignored as the vehicle speed output is the only important output that is used for calculating the controller error. This vehicle speed edge is used as feedback of the closed loop and is also displayed on a scope in combination with the edge *desired_speed*. To find appropriate PID controller parameters K_p, K_i, and K_d, the Ziegler-Nichols method is used as described in the next section.

Ziegler-Nichols Method for the Speed Controller

The Ziegler-Nichols method is used for setting up and tuning the PID controller. First, the controller is threated as simple proportional controller. Thus, $K_p \neq 0, K_i = 0$ and $K_d = 0$. Three meters per second is more than ten percent but that is no problem for the Ziegler-Nichols method. Setpoint for the Ziegler-Nichols method is a step-function that has a step width of the order of ten percent of the maximum setpoint range. This range is proposed in [Chr05] but different step widths do not change the resulting values, it might just take longer to find them. In this case, the input is a step from zero to three after one second (see figure 4.20).

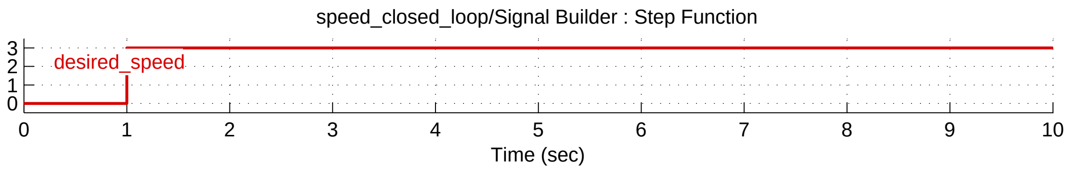


Figure 4.20.: Matlab/Simulink® signal builder block used as input for parameter determination of the Ziegler-Nichols method.

Starting from zero, the proportional coefficient K_p is slowly increased until the closed loop starts oscillating (see figure 4.21 and figure 4.22). This state of the closed loop is called critically stable. The corresponding coefficient is the so-called critical proportional coefficient. In this case, the critical coefficient is $K_{p_{crit}} = 3.5$. Using this value, the closed loop gets critically stable (see figure 4.23). The oscillating period is called critical period. After finding $K_{p_{crit}}$, the critical period T_{crit} is determined from figure 4.23: $T_{crit} = 1.6\,\mathrm{s}$.

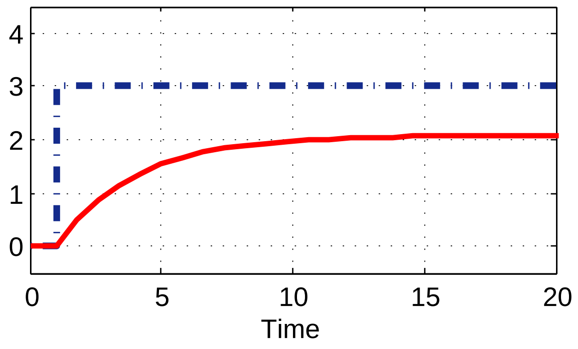


Figure 4.21.: Initial closed loop response for $K_p = 0.1$.

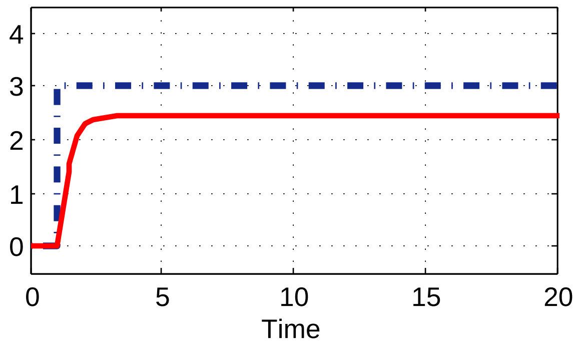


Figure 4.22.: Closed loop response for $K_p = 1$.

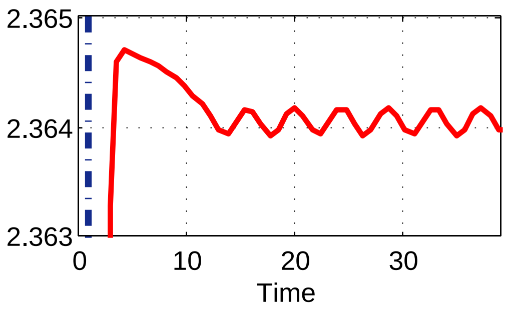


Figure 4.23.: Oscillating response for $K_p = 3.5$.

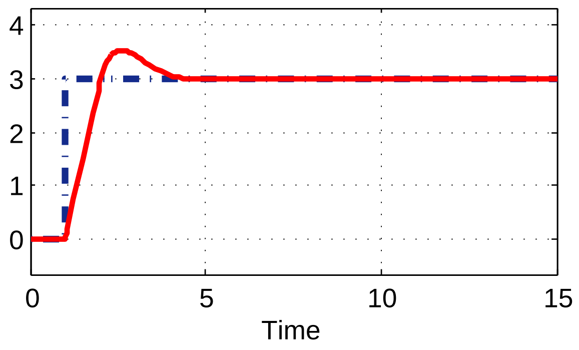


Figure 4.24.: Response for parameters determined with Ziegler-Nichols method.

Using the Ziegler-Nichols method and according to table 4.2, the PID controller parameters should be setup as follows:

$$K_p = 0.6 \cdot K_{p_{crit}} = 0.78 \quad K_i = 2 \cdot \frac{K_p}{T_{crit}} = 0.366 \quad K_d = K_p \cdot T_{crit} \cdot \frac{1}{8} = 0.415$$

After using the PID parameters as described in table 4.2, the controller shows very fast response times (see figure 4.24) but also slightly overshooting behavior. This overshooting behavior is an undesirable behavior for a cruise control system because the controller allows the car to drive faster than the user has requested. Because it is an overshoot of approximate 15 %, a car would travel at 11.5 $\frac{\text{km}}{\text{h}}$ instead of 10$\frac{\text{km}}{\text{h}}$, which is more than allowed by the *Requirements Analysis* phase.

PID parameter tuning is used for getting the most appropriate behavior for an adaptive cruise control system. The K_p part has not been modified. K_i and K_d are manually tuned

Controller	K_p	K_i	K_d
P	$0.5 \cdot K_{p_{crit}}$	-	-
PI	$0.45 \cdot K_{p_{crit}}$	$1.2 \cdot \frac{K_p}{T_{crit}}$	-
PD	$0.55 \cdot K_{p_{crit}}$	-	$K_p \cdot T_{crit} \cdot 0.15$
PID	$0.6 \cdot K_{p_{crit}}$	$2 \cdot \frac{K_p}{T_{crit}}$	$K_p \cdot T_{crit} \cdot \frac{1}{8}$

Table 4.2.: Ziegler-Nichols rules for determining controller parameters [Chr05].

for getting better results: The values are slowly decreased and increased and the control loop results are reviewed. After few iterations, figure 4.25 shows the tuned closed loop response of the speed controller using $K_{p_{new}} = K_p$, $K_{i_{new}} = K_i \cdot 0.7$, and $K_{d_{new}} = 0.042$. One can easily see that on the one hand the overshoot is not that far, but on the other hand is takes longer to reach the final speed.

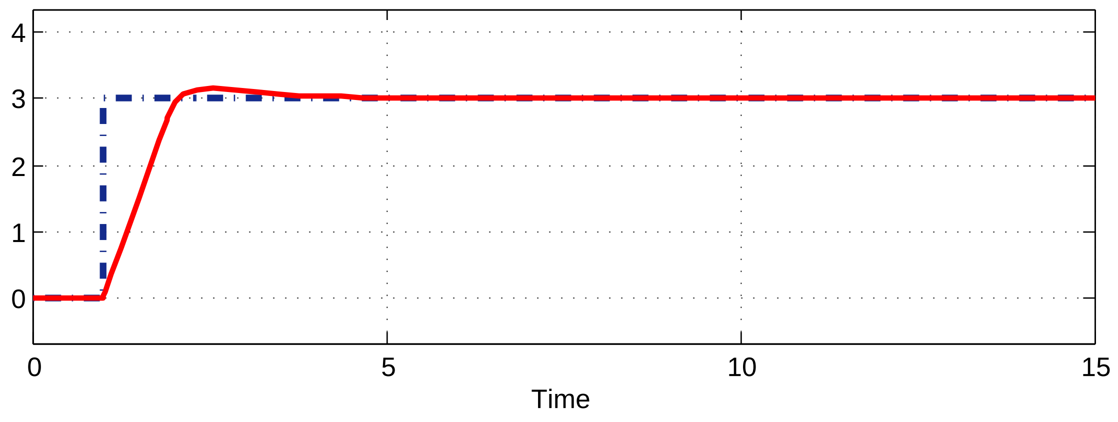


Figure 4.25.: Closed loop step response of the speed controller after parameter tuning.

The controller is verified for different simulation scenarios in chapter 4.4.3 on page 76. Hence, there is no further need to evaluate the controller behavior in this chapter.

4.2.6. Control Distance

The distance controller is used to control the spacing between two vehicles to a specific value. In requirements phase was stated, that the time gap between two vehicles should be one to two seconds. Additionally, the distance should never be less than one meter.

To control the distance between two vehicles, the first thing to do is to calculate the desired distance L_{des}. The distance controller uses the current vehicle speed to figure out the desired distance between the vehicles. Literature states that a time gap of one to two seconds between two vehicles is a common guideline value [Kra08]. Hence, 1.5 seconds will be used in our case. For example, at a speed of 10 $\frac{\text{km}}{\text{h}}$ the desired distance will result to $\Delta x = 1.5\text{s} \cdot 10\frac{\text{km}}{\text{h}} \approx 4.2\text{m}$. Another common value for the distance between two cars is half the value on the speedometer. If the speedometer shows $10\frac{\text{km}}{\text{h}}$, the distance should be around $\frac{10\,\text{m}}{2} = 5\,\text{m}$ which is more than the previously calculated value. Thus, our assumed time gap of 1.5 seconds seems to be a good value.

Accordingly, minimum spacing between two vehicles is calculated from the vehicle speed. After the minimum distance is calculated, the component makes sure that the distance will never be less than one meter even if the cars are not moving and thus the vehicle speed is zero.

$$L_{des} = \min(1\,\text{m}, \dot{x} \cdot 1.5\,\text{s})$$

The resulting value is subtracted from the current distance to the vehicles predecessor to get the spacing error δ that is considered as the control deviation. In simulation, the distance value is calculated from the distance of the route that the two vehicles have covered (see figure 4.26):

$$\delta = x_2 - x_1 - L_{des}$$

Where $x_{1,2}$ are the distances that the two vehicles have covered. The deviation is fed into a controller to get the desired motor torque.

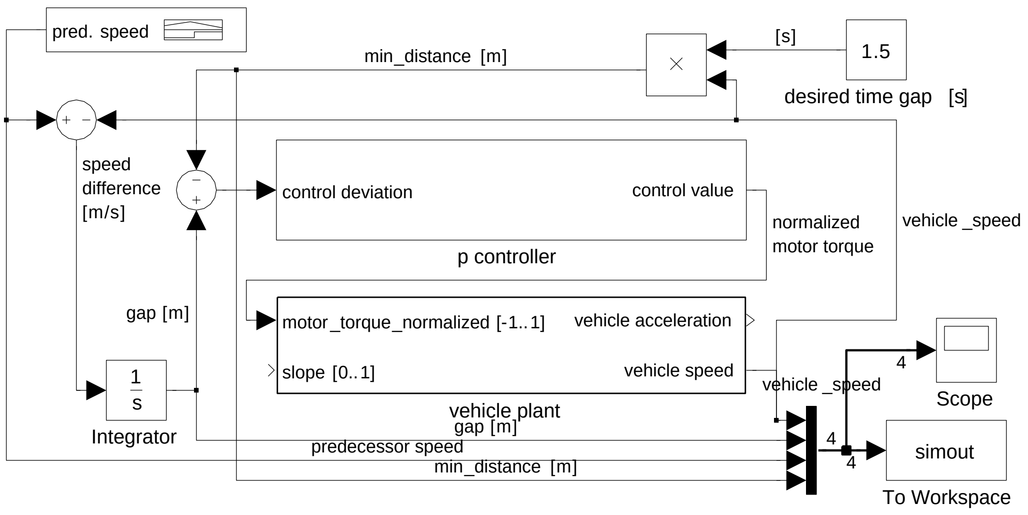


Figure 4.26.: Closed loop for finding the distance controller parameter.

Figure 4.26 shows the closed loop for testing the distance controller. The distance $x_2 - x_1$ that is used in this closed loop is calculated from the two vehicle speeds:

$$x_1 = \int_0^t \dot{x}_1(\tau) d\tau, \quad \text{and} \quad x_2 = \int_0^t \dot{x}_2(\tau) d\tau$$

As one can also see in the figure, the control deviation results to:

$$\delta = \int_0^t \left(\dot{x}_2(\tau) - \dot{x}_1(\tau)\right) d\tau - 1.5\,\mathrm{s} \cdot \dot{x}_1$$

This deviation is fed into the distance controller, which is a proportional controller without integral part because on the one hand the integral part would integrate an error also when the controller is inactive. On the other hand, the closed loop already has an integral part as one can see in the formula above. Thus, the positioning fault of the closed loop will reach zero even if the controller has no integral part.

Ziegler-Nichols Method for the Distance Controller

Again, like for the speed controller, the controller parameter is determined in the *Functional Design* step. Initially, the simulation start condition is configured like follows:

- Initial distance between the two vehicles is set to one meter (spacing between the cars). This is achieved by setting the integrator of figure 4.26 to the specified value one. Bigger values for the initial distance would also work but it would take longer for the successor vehicle to catch up its predecessor.
- The concept car is initially not moving: $\dot{x}_1 = 0\,\frac{\mathrm{m}}{\mathrm{s}}$. The speed setting can be modified in the vehicle plant integrator. Setting the successor vehicle speed to less than the predecessor vehicle speed makes sure that in the simulation the concept car does not overtake its vehicle ahead.
- The predecessor vehicle is cruising at a constant speed of one meter per second. The predecessor vehicle speed is defined in the signal builder.

Figure 4.27 shows the behavior of the closed loop under the described initial conditions for a proportional controller with $K_p = 1$. The dotted cyan line that has a constant value of 1 is the speed of the predecessor vehicle and the green line shows the speed of the concept car. The dotted blue line shows the distance of the two vehicles and the red line shows the desired distance of the two vehicles. Hence, the goal of parameter tuning should be that

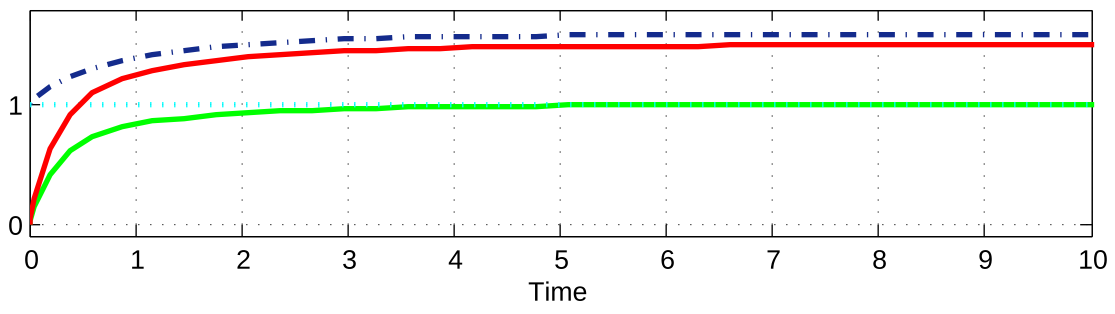


Figure 4.27.: Closed loop using a proportional controller with $K_p = 1$.

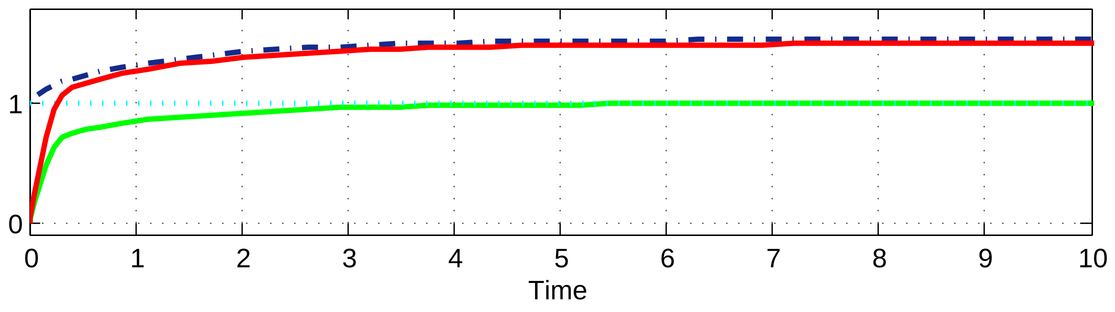


Figure 4.28.: Closed loop starts oscillating for a proportional coefficient of 2.5.

the current distance (dotted blue) is in any case bigger or equal than the current spacing distance.

As described in the previous section, the Ziegler-Nichols method requires an increase in the proportional coefficient of the P controller until the closed loop gets critical stable. For a proportional value of $K_{p_{crit}} = 2.5$ the closed loop starts oscillating periodically (see figure 4.28). Figure 4.29 shows a zoomed view of the oscillations for determining of the critical period $T_{crit} = 0.625\,\mathrm{s}$.

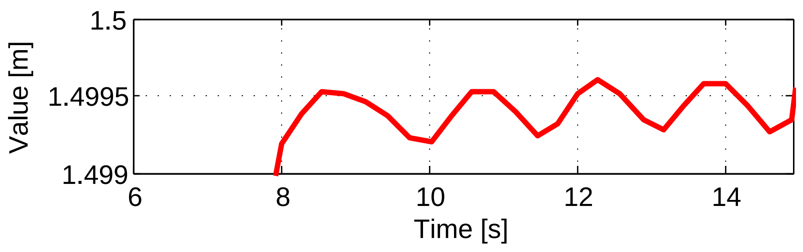


Figure 4.29.: Zoomed view on the oscillation for determining the critical period.

According to table 4.2, to use a proportional controller the parameter has to be set to half of the value of the critical gain: $K_p = 0.5 \cdot K_{p_{crit}} = 1.25$. Figure 4.30 shows the behavior of the closed loop for the estimated proportional gain. Additionally, the speed of the predecessor vehicle is reduced by 50 percent after four seconds and is reseted to the original value after eight seconds.

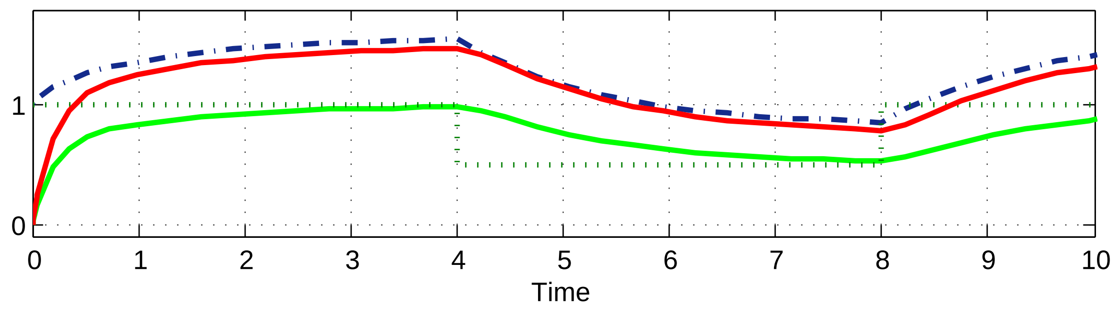
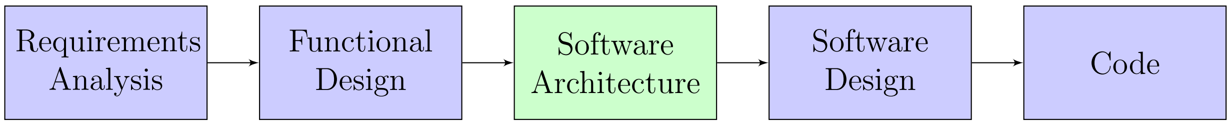


Figure 4.30.: Final closed loop response of the distance controller.

Finding the distance controller parameter was the last thing to do in the *Functional Design* step of the model-based embedded software development process. Now, all necessary inputs and outputs of the system are sufficiently clear and all filters and controllers that will be used are specified, simulated, and optimized. The results of this step are models for all necessary controllers. The next step in the development process is the *Software Architecture* step that depends on the results of this phase.

4.3. Software Architecture



The functional behavior is now completely modeled by domain experts and now the task of software engineers starts. Thus, in the *Software Architecture* phase, the actual software development starts. The input of the *Software Architecture* phase are the results of the requirements analysis phase and also the functional models of the domain experts of the *Functional Design* phase. Output of the phase is a refined model of the system that is consistent and free of errors.

Task of the *Software Architecture* phase is the structuring of the models. It is an important phase in the model-based software development process for embedded systems: On the one hand, the system is refined for making it better understandable and on the other hand, the *Software Architecture* phase helps finding errors in the models. To achieve those two points, different diagram types that ensure different views on the system are used.

The *Software Architecture* phase is the first phase that not only handles functional properties, but also non-functional properties of the system. Non-functional properties imply a number of design decisions that include performance, memory consumption, reliability, reusability, maintainability, extensibility, power consumption, and production cost. Many of those non-functional properties influence each other and especially the production cost depends on decisions of all other properties. Production cost is an important property for embedded systems as embedded systems often have high quantities and thus a small increase of production cost of one part will be a huge increase of the overall production cost.

The *Software Architecture* phase is separated in the static structure part and the dynamic structure part. Both parts are described in this chapter: The static structure (see chapter 4.3.1) helps to understand how the system is divided into components. The dynamic structure (see chapter 4.3.2) defines how these components communicate with each other.

4.3.1. Static Structure

Decomposition of the System

The first thing to do in the *Software Architecture* phase is the static decomposition of the system (see figure 4.31). The system is structured into three layers: the adaptive cruise control application software, the platform software, and the operating system.

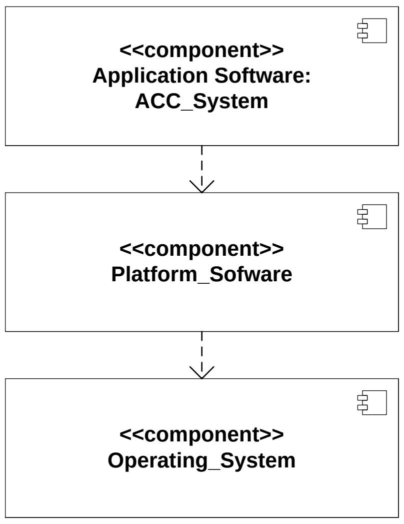


Figure 4.31.: Structure of the complete adaptive cruise control embedded system.

1. The *application software* consists of the Matlab/Simulink® models of the adaptive cruise control system. It contains all controllers, filters, state machines, and especially all modules that were developed and tested in the *Functional Design* phase.

2. The *platform software* handles the models in- and output ports by passing them to the CAN-bus. Between two different types of semantics of the CAN messages can be distinguished and all of them are differently handled by the platform software:

    - Some messages need marshalling on the values on the CAN-bus for getting the actual values in SI units. For example, the acceleration sensors have a measurement range of $-8g$ to $8g$ (where g is the gravitational acceleration) and as they are 12 bit sensors, they produce values in the range of -2048 to 2047. Hence, the values on the CAN-bus have to be processed for getting the actual acceleration a in $\frac{\mathrm{m}}{\mathrm{s}^2}$:

$$a = \frac{8}{2048} \cdot x \cdot 9.81 \frac{\mathrm{m}}{\mathrm{s}^2}$$

This processing is performed by the platform software before passing the value to the application software models.

- Other CAN messages are the messages that need intense preprocessing which is done in the application software and is not part of the platform software. The platform software in this case just passes the unprocessed values and the application software runs filters or state machines for getting useful values in SI units. One example for this type of messages are the wheel speed sensors that are described in detail on page 20 with their filtering that is described on page 41.

3. The lowest layer of the adaptive cruise control system is the *operating system*. The operating system provides drivers for the abstraction of the used hardware. Currently, the concept car uses the following hardware for running the controllers:

   - The CAN-bus may be the most important piece of hardware on the controller board and is described in chapter three.

   - SD card interface for storing the software on. The SD card is also used for debugging purposes. A file allocation table (FAT) driver allows the use of filesystems and thus easier access to the software binary and the log files on other systems like personal computers.

   - An universal asynchronous receiver/transmitter (USART) is used for debugging purposes.

   - Other low-level drivers allow the control of on-chip peripherals like I/O-ports, interrupt controllers, and timers.

   The operating system also takes care of the scheduling of the model components. At the moment no multi-tasking operating system is used and all blocks and components of the Matlab/Simulink® model are executed sequentially with a fixed time interval (see chapter 4.4.4 on page 82).

Component Decomposition

After the embedded system has been structured in several layers, the system itself, the *ACC_System* - will be segmented into its components as shown in the component decomposition diagram 4.32. The decomposition is an UML class diagram that segments the *ACC_System* into the following sub-components:

- The *UserInterface* receives and evaluates the commands of the user. Enabling and disabling the controller and setting the desired speed is detected by the user interface as well as disabling the controller when receiving a brake signal.
- *DataAcquisition* is the component that receives all sensory input and calculates the distance to the predecessor, the wheel speeds, and the vehicle speed and forwards the results to the *ACC_Controller*.
- The *ACC_Controller* is the main component of the adaptive cruise control system. It receives all control signals of the user interface and all pre-calculated values of the data acquisition component and finally calculates the motor torque that should be applied to the vehicle motor.

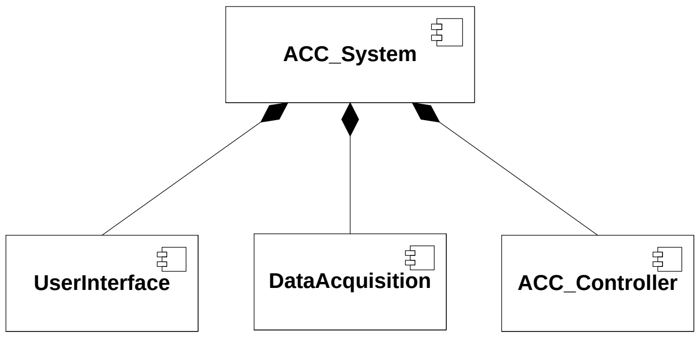
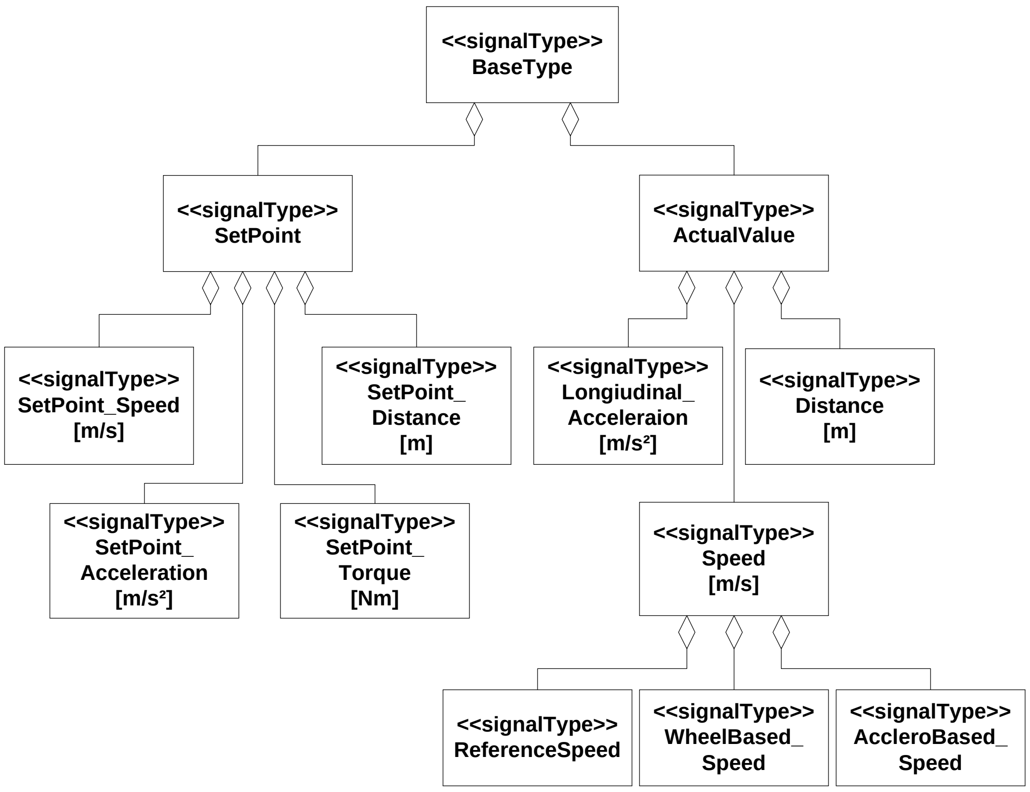


Figure 4.32.: Decomposition of the ACC System.

Type System

With the help of a type system (see figure 4.33) many errors can be avoided. For example, if a component expects an input to be of the type $\frac{m}{s}$ but it actually is the type $\frac{\text{km}}{\text{h}}$, the type system will point out this complicated mistake. As stated before, all values of the models in this thesis will be converted to SI units to avoid errors and to improve comprehensibility of the simulation results. The types of all model inputs and outputs have to be determined



<<signalType>> BaseType
<<signalType>> SetPoint
<<signalType>> ActualValue
<<signalType>> SetPoint_Speed [m/s]
<<signalType>> SetPoint_ Distance [m]
<<signalType>> Longiudinal_ Acceleraion [m/s²]
<<signalType>> Distance [m]
<<signalType>> SetPoint_ Acceleration [m/s²]
<<signalType>> SetPoint_ Torque [Nm]
<<signalType>> Speed [m/s]
<<signalType>> ReferenceSpeed
<<signalType>> WheelBased_ Speed
<<signalType>> AccleroBased_ Speed

Figure 4.33.: Type system of the adaptive cruise control system.

and one has to make sure that the type of an output fits the types of all connected inputs. If the type differs, a type conversion has to be used like demonstrated in figure 4.11 on page 42.

All types derive from the base type called *BaseType*. A distinction is drawn between values that shall be used as control value (called *SetPoint*) and values that are measured (called *ActualValue*). The set point values are distinguished into the following four types:

- *SetPoint_Speed* is used for all data flow paths that are input to a speed controller and is measured in $\frac{m}{s}$.
- *SetPoint_Acceleration* is the output of controllers and reflects the desired acceleration of the vehicle. The *SetPoint_Acceleration* values are of the type $\frac{m}{s^2}$.
- The acceleration is converted to a desired torque of the type *SetPoint_Torque* that is measured in Nm.

- Last, there is the need for a type that may be used as input of the distance controller: *SetPoint_Distance* [m].

The type *ActualValue* is divided into three sub-types:

- The *Distance* type uses the unit meters and is used for the dataflow paths that carry the distance to the predecessor vehicle.

- *Speed* is measured in meters per second and distinguishes the *ReferenceSpeed*, the *WheelBased_Speed*, and the *AcceleroBased_Speed*. The reference speed is the actual speed of the concept car that may not be perfectly measured but e.g. with the use of global positioning system (GPS) one may get good approximations. Calculating the vehicle speed based upon the wheel speed sensors results in values of the type *WheelBased_Speed*. *AcceleroBased_Speed* is the speed that results from integrating over the acceleration sensors.

- *Longitudinal_Acceleration* type is used for values that correlate with the longitudinal acceleration of the vehicle. These values are measured in $\frac{m}{s^2}$.

Static Interaction

The composite structure diagram in figure 4.34 shows the static interaction between the sub-components that were discussed earlier in this chapter. The composite structure diagram clarifies the structure of the system by showing the dataflow starting at the system inputs (sensors and user interfaces) and ending at the system outputs (actuators).

4.3.2. Dynamic Structure and Dynamic Interaction

Figure 4.35 shows the refined sequence diagram of the use case control speed with distance. The diagram is similar to figure 4.5 but demonstrates a more technical view and shows the interaction between **all** sub-components of the system. The first sequence diagram mainly shows the interaction between the extern components and the system itself. As well as all components are split into sub-components, all life lines of those components will be split and connected in the appropriate way. This is needed for a better understanding of the model and for recognition of incompleteness of the interactions.

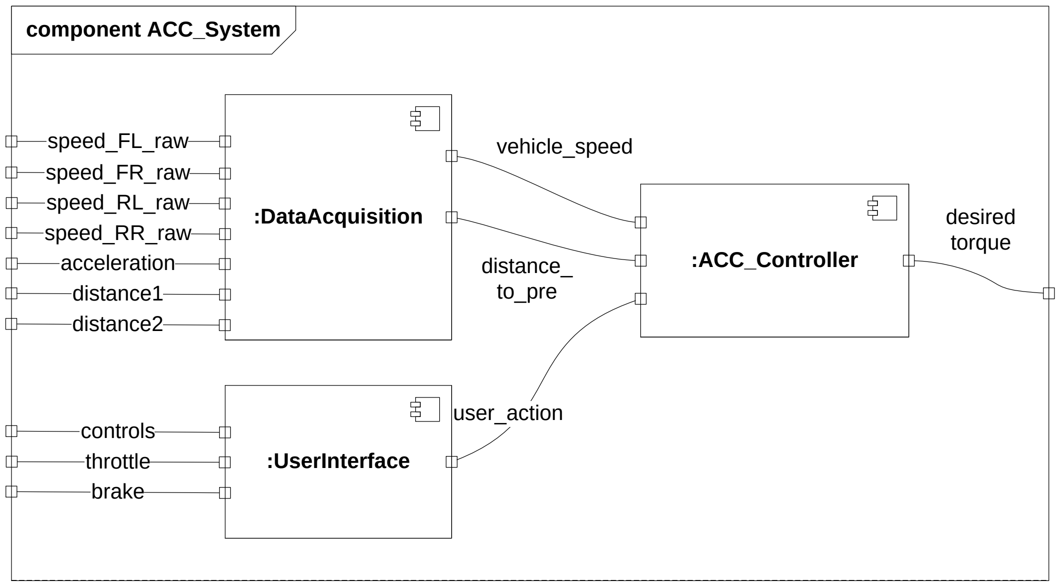


Figure 4.34.: Composite structure diagram of the adaptive cruise control system.

The component *Wheel Speed Sensor* is splitted into four life lines, while the *Distance Sensor* is splitted into the two sensors. Consequently, distance and vehicle speed may be calculated redundantly and sensory errors can be detected. The additional component *Data Acquisition* is added to handle the sensory inputs. In this refined sequence diagram, the complete dataflow from sensory input to actuator output is described.

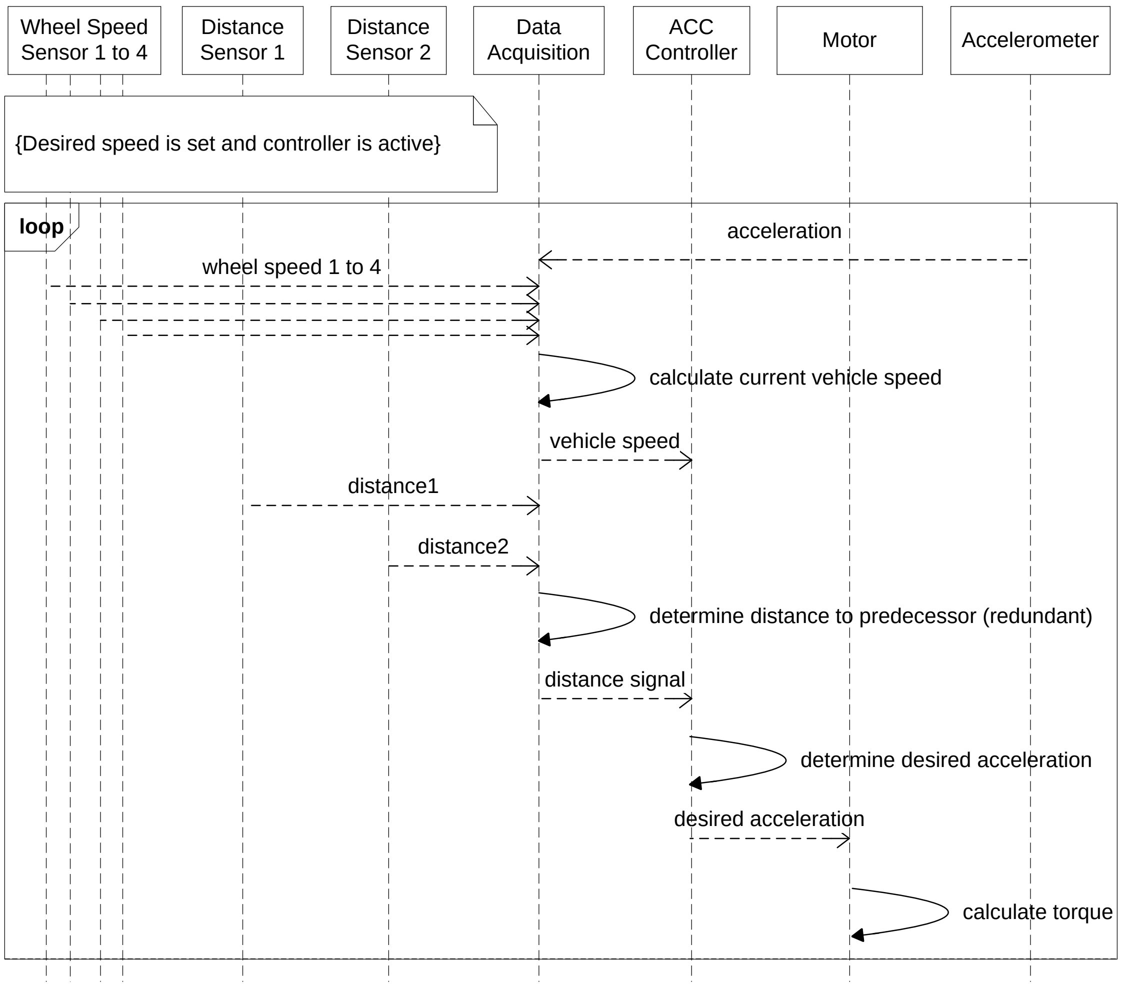
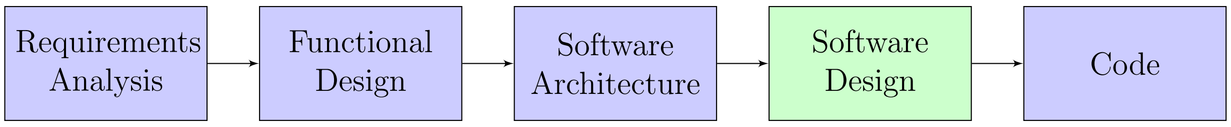


Figure 4.35.: Refined sequence diagram of the scenario *Control speed with distance*.

4.4. Software Design



There is a smooth transition from the *Software Architecture* phase to the *Software Design* phase. Thus, one can not clearly mark out the boundaries of both phases. The *Software*

Design phase takes the results of all previous phases as input. The output of this phase is a complete behavioral model that can be used for the generation of the application code.

After the most important components are structured, one can start the *Software Design* phase. This phase is divided into three steps: In the *Structural Refinement* step, the components are further refined until the complexity reaches a reasonable value (see chapter 4.4.1). Afterwards, in the *Behavioral Design* step, all behaviors of all components are defined (see chapter 4.4.2). Furthermore, these platform independent models are tested in chapter 4.4.3. The *Platform Specific Design* step takes platform-specific clues into consideration (see chapter 4.4.4). After the last step of this phase is finished, the code can automatically be generated.

4.4.1. Structural Refinement

The task of the *Structural Refinement* step is the refinement of all components while they are still too complex. This step is finished when there is no need to refine them anymore and the complexity has reached a value that can easily be understood. This means, that the behavior of the refined sub-components can be easily and flawlessly defined.

Static Decomposition

Figure 4.36 shows the static decomposition of the component *ACC_Controller* that has not been refined in the *Software Architecture* phase. The *ACC_Controller* is divided into

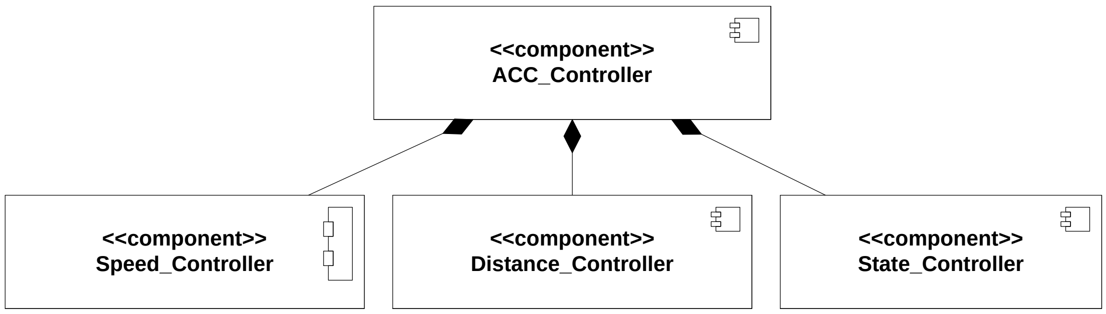


Figure 4.36.: Decomposition of the component *ACC_Controller*.

the three sub-components: *Speed_Controller*, *Distance_Controller*, and *State_Controller* as follows:

- *Speed_Controller*: This component is responsible for controlling the speed of the vehicle by influencing the motor torque in a way that minimizes the control difference (*desired_speed* - *vehicle_speed*).
- The *Distance_Controller* works in a similar way but instead controls the time gap of 1.5 seconds between two vehicles as defined in the *Requirements Analysis* phase (see chapter 4.1). In any case, the distance should not be less than one meter.
- The *State_Controller* determines, depending on the distance between the vehicles and the speed of them, which controller to use.

As three sensory inputs are made available (see the *Requirements Analysis* phase) by the system, the component *DataAcquisition* is also be divided into three sub-components (see figure 4.37) for calculating these values:

- *WheelSpeed_Calculator*: Filter that converts the wheel speed raw values to normal SI unit values.
- *VehicleSpeed_Calculator*: Calculates the vehicle speed from the wheel speeds and the longitudinal acceleration.
- *Distance_Calculator*: Gets two redundant distance values from two ultrasonic sensors and combines them to one distance value.

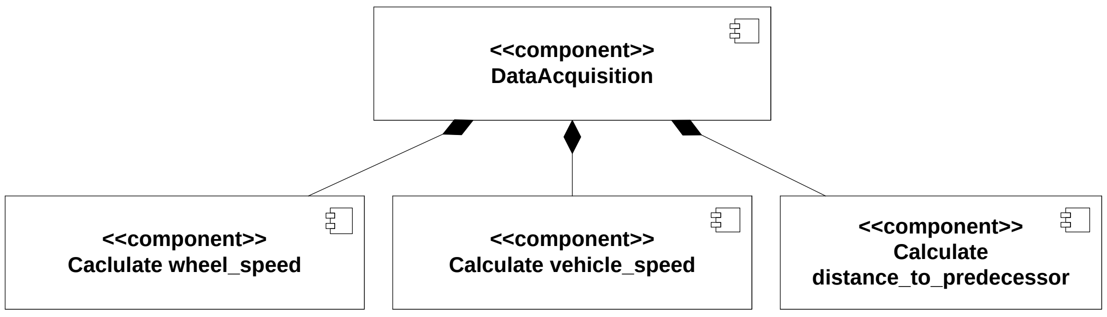


Figure 4.37.: Decomposition of the component *DataAcquisition*.

Static Interaction

The static interaction of the *ACC_Controller* is shown in the composite structure diagram 4.38. One can see that the component gets the user interactions, the vehicle speed, and the distance to the predecessor as input. The output is the desired motor torque that is directly forwarded to the CAN-bus. The vehicle speed is used in all sub-components. The user interaction is just needed in the *Speed_Controller* while the distance to the predecessor vehicle is needed in the *State_Controller* and the *Distance_Controller* but not the *Speed_Controller*. Both controllers - the *Speed_Controller* and the *Distance Controller* - have their own motor torque as output and the *State Controller* decides depending on the actual state which torque to forward to the *ActorBoard* for driving the motor. The *State_Controller* also handles the brake events and disables both torque controllers if the brake lever is pulled or the controller is disabled or not yet enabled.

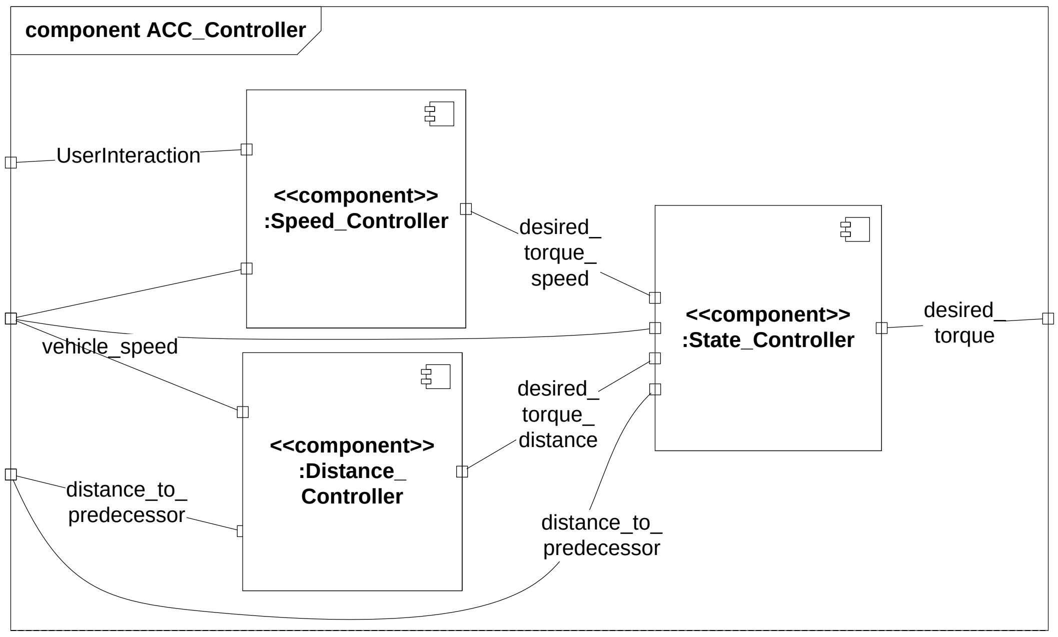


Figure 4.38.: Composite structure diagram of the component *ACC_Controller*.

The composite structure diagram 4.39 shows the static interaction of the component *DataAcquisition*. The component has two main dataflows. On the one hand, the two distance sensor values are forwarded to the *Distance Calculator* and the results are further forwarded to the component boundaries of the data acquisition module. On the other hand, the four wheel speed raw values are processed in the *WheelSpeed Calculator* sub-module and in combination with the longitudinal acceleration, the vehicle speed can be calculated and passed to the *DataAcquisition* module.

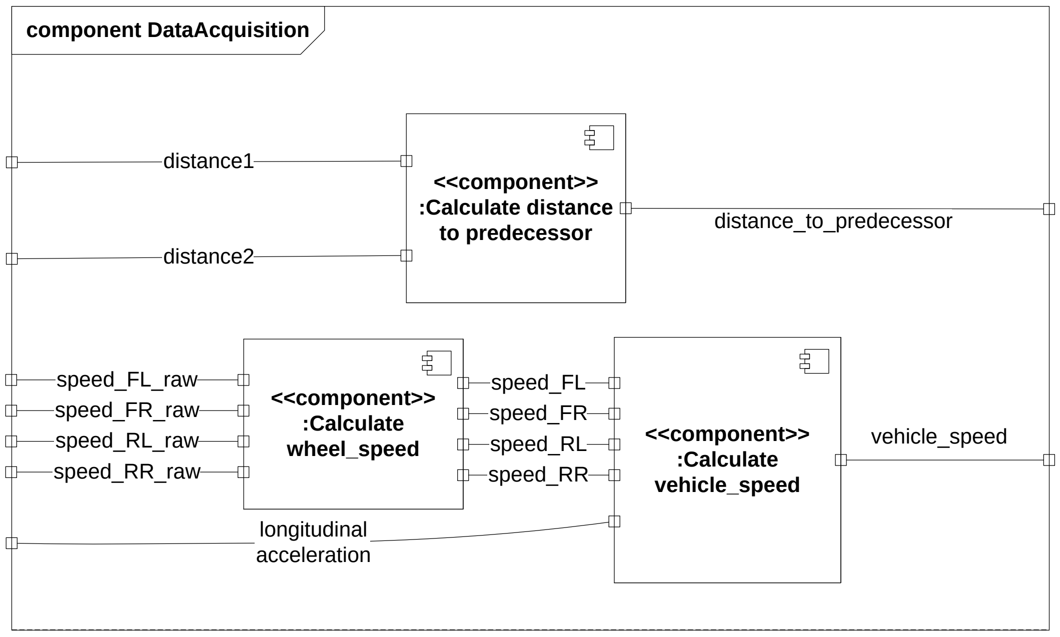

component DataAcquisition
distance1
distance2
<<component>> :Calculate distance to predecessor
distance_to_predecessor
speed_FL_raw
speed_FR_raw
speed_RL_raw
speed_RR_raw
<<component>> :Calculate wheel_speed
speed_FL
speed_FR
speed_RL
speed_RR
<<component>> :Calculate vehicle_speed
vehicle_speed
longitudinal acceleration

Figure 4.39.: Static decomposition of the component *DataAcquisition*.

Dynamic Interaction

In the *Dynamic Interaction* step, in contrast to the *Software Architecture* phase, the life lines are not just splitted into sub-components, but also new sequence diagrams may be used for avoiding really high complexity of the diagrams. Instead of high complexity and confusing diagrams, the interaction diagrams for the sub-components will be analyzed in this thesis.

Sequence diagram 4.40 describes the dynamic interaction of the component *Speed_Controller*. The *Speed_Controller* gets the current vehicle speed and the speed that the user desires. Those two values are subtracted for getting the controller difference. The speed controller contains a PID controller that calculates the control value (motor torque) out of this controller difference. The control value is passed to the state controller as *desired_torque_speed* which selects whether to use the torque of the *Speed_Controller* or of the *Distance_Controller* for motor control.

The sequence diagram of the distance controller component is nearly the same as the diagram of the *Speed_Controller* component. The difference is that the distance controller does not need the desired speed but instead the distance to the predecessor vehicle. It calculates the desired time gap between the two vehicles (see chapter 4.2.6) and subtracts

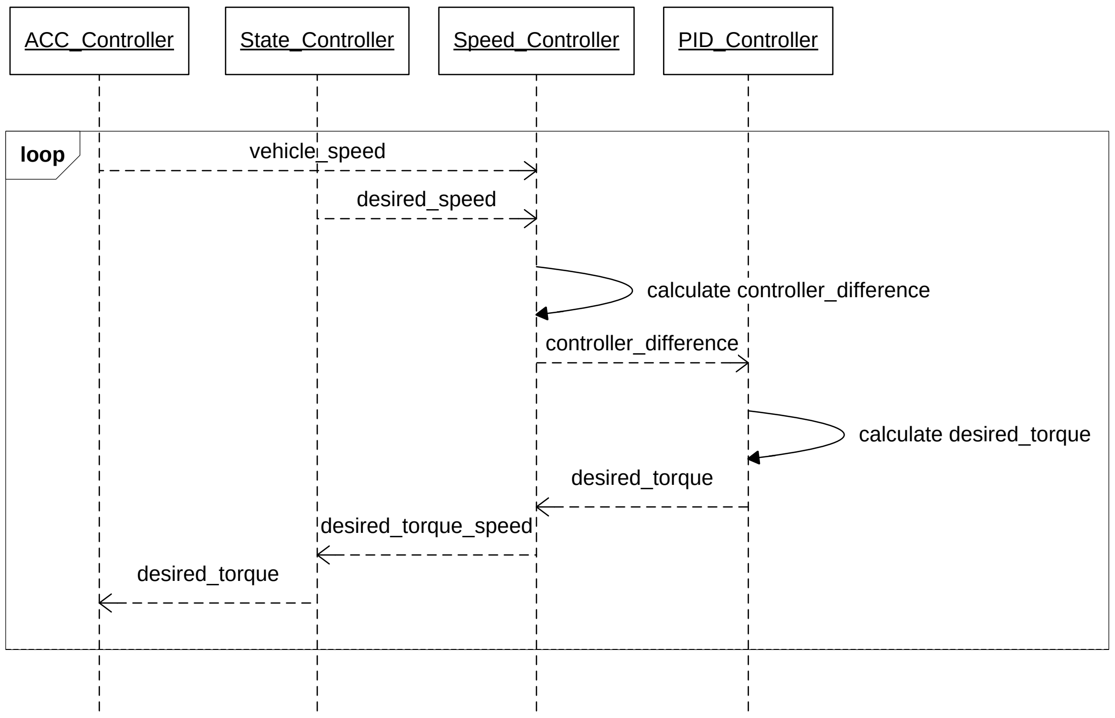


Figure 4.40.: Sequence diagram of the component *Speed_Controller*.

the current distance to the predecessor for getting the controller difference. The rest equals the *Speed_Controller*.

Sequence diagram 4.41 shows the dynamic interaction of the component *DataAcquisition*. The sequence diagram illustrates how the component communicates with its subcomponents for calculating the vehicle speed and the distance to the predecessor. First, all raw values of the four wheel speed sensors are converted to SI units and in combination of the longitudinal acceleration the vehicle speed is calculated. Both distance sensor values are combined to one distance value and this value and the vehicle speed value are forwarded to the *ACC_Controller*.

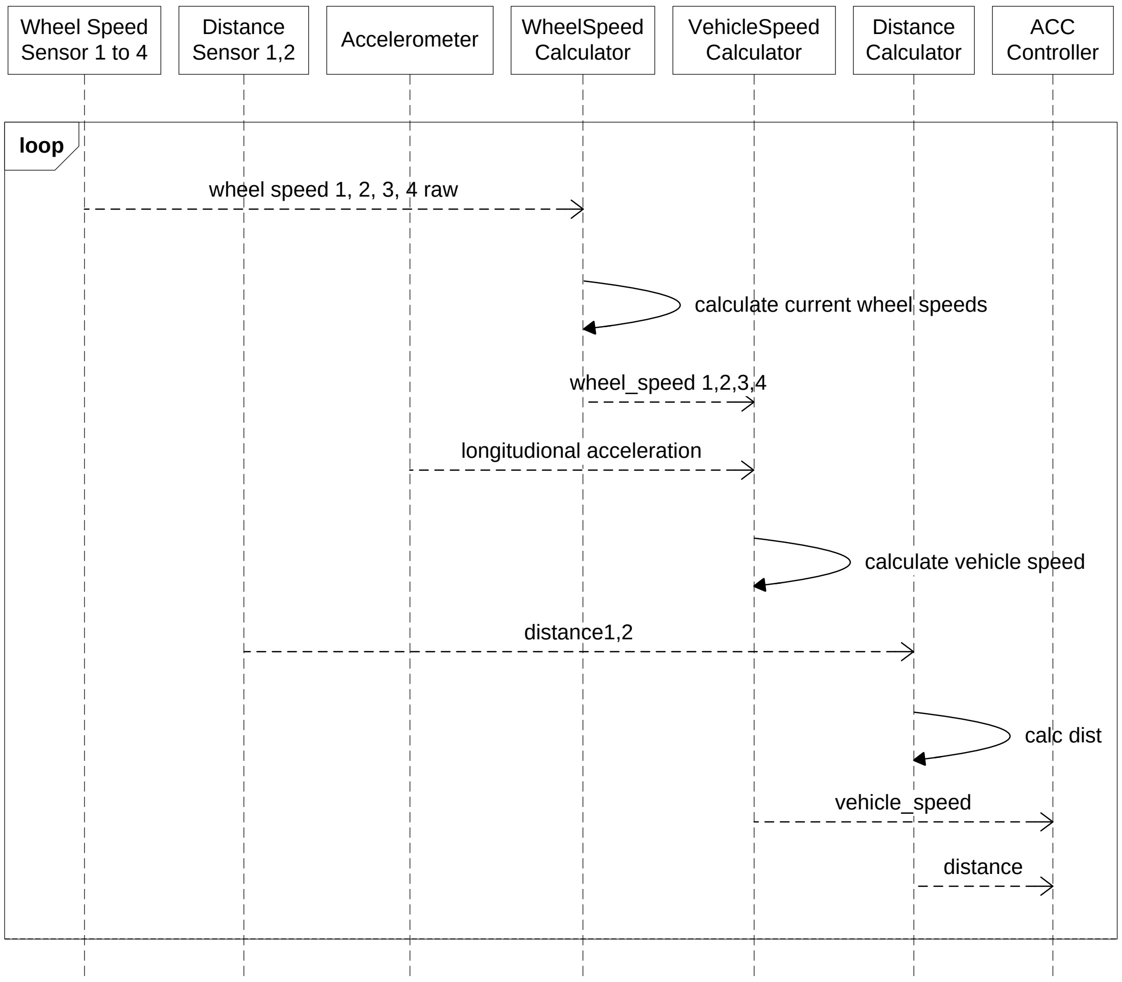


Figure 4.41.: Sequence diagram of the component *DataAcquisition*.

4.4.2. Behavioral Design

After the static structure and the dynamic interaction of all components of the adaptive cruise control system are defined, the behavior is the next thing to specify. All components of *ACC_System*, namely the *ACC_Controller* and the *DataAcquisition* and the system itself have to be fully specified.

ACC_System

The *Behavioral Design* of the *ACC_System* (see figure 4.42) equals the static decomposition (composite structure diagram) of the module. The diagram was strictly converted to a Matlab/Simulink® block diagram to get a defined semantic and to be able to generate code from the model in the later design phases. The distances input of the ACC system (input 2) is a bus that actually holds two values. In the associated UML diagram, both sensory inputs are shown as single inputs because UML is not capable of defining buses. The same applies to the raw wheel speed input that combines all four wheel speed values.

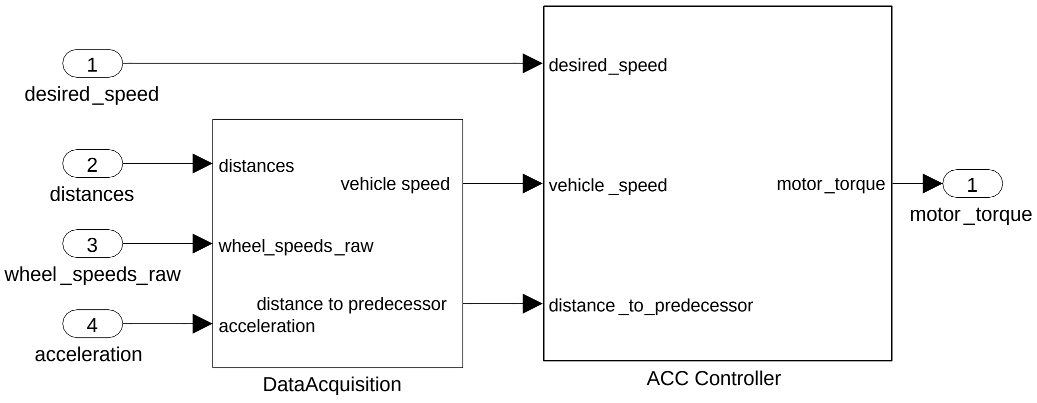


Figure 4.42.: Block diagram of the component *ACC_System*.

ACC_Controller

After the structure of the ACC controller component was decomposed into its sub-components, it is necessary to define its behavior now. Figure 4.43 shows the Matlab/Simulink® model of the component. The block diagram looks similar to the composite structure diagram of this component because it does not need any additional blocks other than connecting its sub-components with each other. The main difference is that other than the composite structure diagram, that was modeled in an UML drawing tool without semantics, now code can be generated of the model using Real-Time Workshop® Embedded Coder™ and the appropriate toolchain.

Figure 4.44 shows the block diagram of the state controller. The task of the controller is to select the torque of either the speed controller or the distance controller. The state controller is simple and effective: It just selects the torque that has the lesser value. This causes exactly the desired behavior:

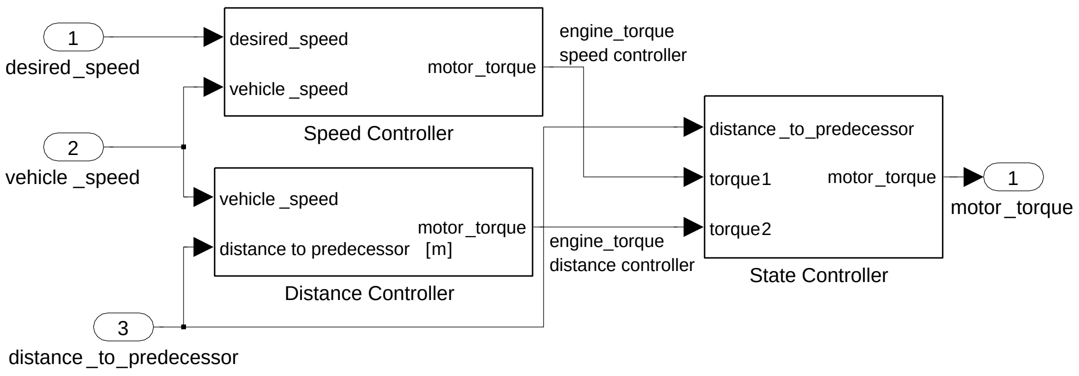


Figure 4.43.: Matlab/Simulink® model of the component *ACC_Controller*.

- The vehicle will never accelerate faster than the *Speed_Controller* suggests. If no obstacle is in front of the concept car, the distance controller torque is set to 100 percent and the *Speed_Controller* suggests a value less or equal than 100 percent and thus is be selected by the state controller. Therefore, the concept car behaves as desired in case of no obstacle in front of the car.

- The vehicle will also never decelerate slower than the distance controller suggests in case of an obstacle. In this case, the *Speed_Controller* desires to drive faster than the distance controller and thus the speed controllers torque is bigger than the distance controllers torque. Hence, the distance controller torque will be selected by the state controller as motor torque. Finally, the concept car behaves as desired in case of an obstacle.

The distance to the predecessor vehicle is not needed at all because this value is already used for calculating the two torque inputs. Thus, the distance value is indirectly taken into consideration. Probably, the state controller could still be optimized but caused by the limited time of this thesis and the well simulation results the controller is used like shown in picture 4.44.

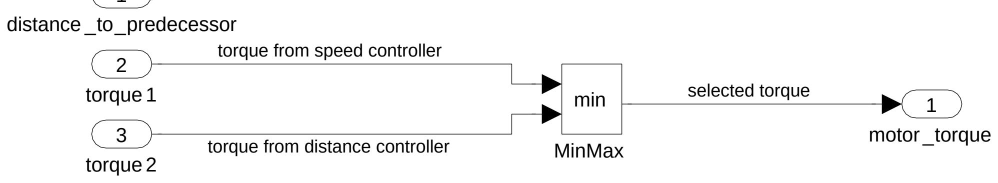


Figure 4.44.: The state controller block diagram.

Figure 4.45 shows the block diagram that defines the behavior of the speed controller component. The component was reused from *Functional Design* phase: It subtracts the

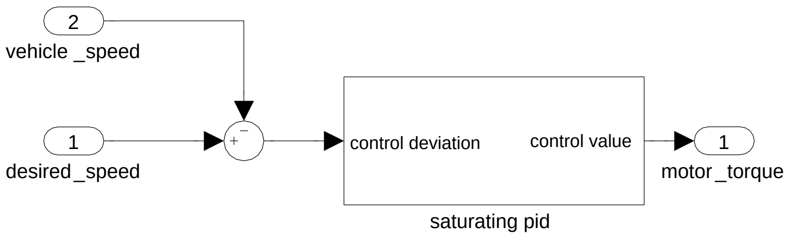


Figure 4.45.: Block diagram of the component *SpeedController*.

vehicle speed and desired speed for getting the controller difference. This difference is used as input for a saturating PID controller (see 4.2.4 on page 48) and the output of the controller is the motor torque.

The block diagram of the distance controller (see figure 4.46) looks similar to the diagram of the speed controller but there are important differences. The distance controller uses

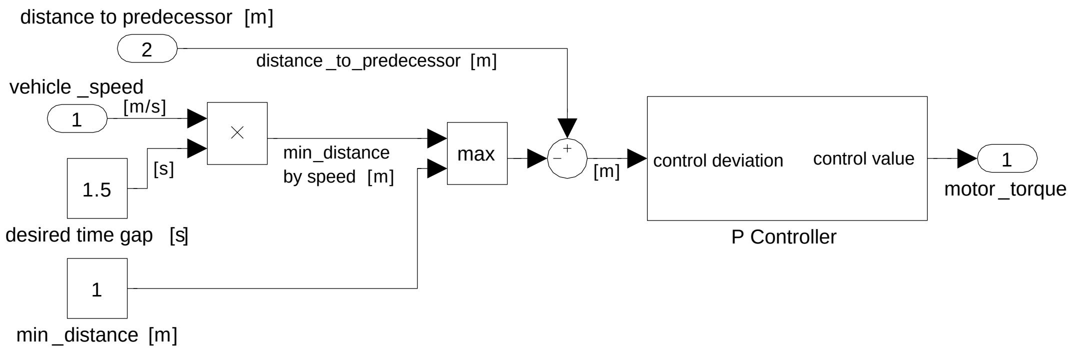


Figure 4.46.: Block diagram of the component *DistanceController*.

the current vehicle speed for figuring out the desired distance between the vehicles as stated in the *Functional Design* phase on page 52. The deviation between desired distance and the actual distance is fed into a controller for getting the desired motor torque that leads to an reduction of the distance error.

DataAcquisition

The *DataAcquisition* module behavior is described by figure 4.47. The component just passes its inputs to its sub-modules that calculate the distance to the predecessor vehicle and the vehicle speed. These sub-modules include calculating the distance to the predecessor vehicle, calculating the wheel based vehicle speed, and calculating the vehicle speed.

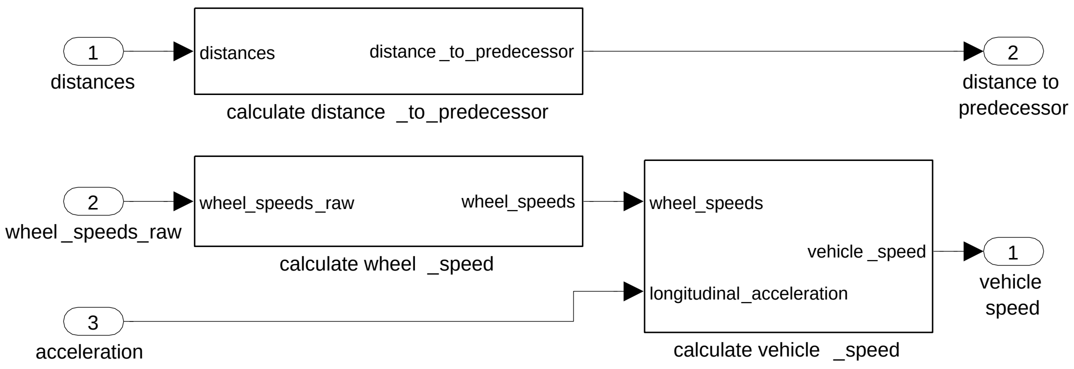


Figure 4.47.: Block diagram of the component *DataAcquisition*.

Figure 4.48 shows the behavior of the component that calculates the distance to the predecessor vehicle: This is simply done by selecting the smaller distance value of both. The distance input is fed with an array of distance values (the concept car has two of them). Those *distance* inputs are measured in the unit centimeters by the distance board. Firstly, the unit is converted to meters for simpler usage later in the model. The following switch-block converts negative input values that signal a distance of more than six meters to a big value to indicate the controller that there is no obstacle in front of the car. The following min-block uses the smallest distance value of all sensors as the *distance_to_predecessor* value.

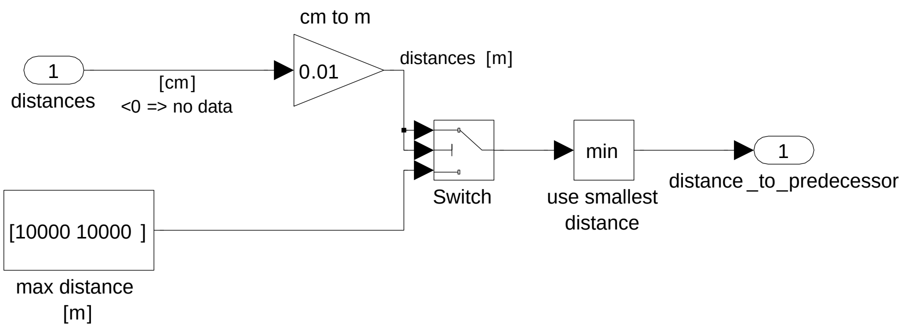


Figure 4.48.: Calculates the distance to predecessor.

Figure 4.49 shows the Matlab/Simulink® model for calculating the vehicle speed from the wheel speeds and the longitudinal vehicle acceleration. First, the number of sensors that deliver values is estimated by checking how many sensors are rotating. Sensors that are standing are not considered in vehicle speed calculation but that is no problem because it would not change the vehicle speed estimation whether they are counted or not. After finding the number of rotating sensors, the sensor average is calculated by adding all sensor values and dividing them by the number of sensors. The quotient is the wheel based vehicle speed. Normally, the longitudinal acceleration could be used to improve the sensor quality of the wheel speed sensors. Additionally, the possible slip of the wheels could be eliminated from the sensor data. Caused by the limited time of this thesis, the longitudinal acceleration is not taken into consideration in my case.

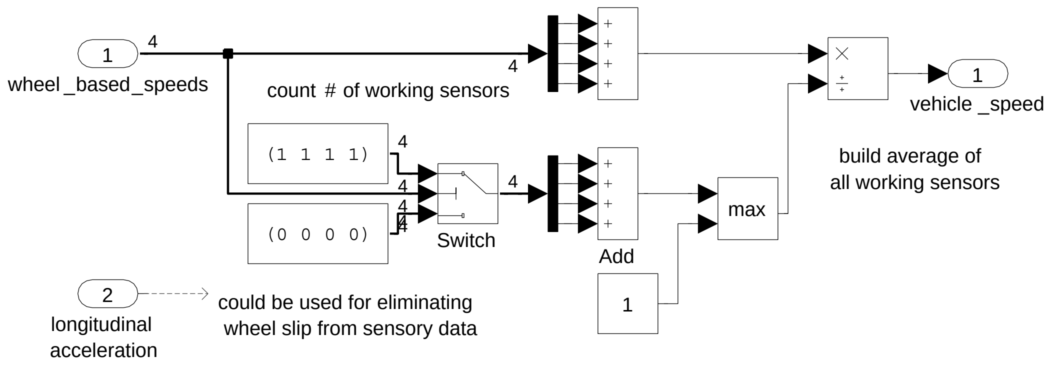

1
wheel _based_speeds
4
count # of working sensors
(1 1 1 1)
(0 0 0 0)
Switch
Add
1
max
1
vehicle _speed
build average of all working sensors
2
longitudinal acceleration
could be used for eliminating wheel slip from sensory data

Figure 4.49.: Matlab/Simulink® model *calculate_vehicle_speed*.

The last sub-component of the data acquisition module is the calculation of the wheel based speeds of the car. This problem has already been solved in the *Functional Design* step by introducing, simulating, and testing the infinite impulse filter on page 41. The filter is simply reused without modifying it. Therefore, the filter has the behavioral block diagram that is shown in figure 4.14 but as the input is a bus that carries four wheel values, the filter is included four times into the model.

4.4.3. Platform Independent System Test

After defining the behavior of each sub-component, modeling the application layer of the adaptive cruise control system is completed. Now, the whole system can be tested by applying simulation test-cases to the system and reviewing how the system reacts. The test cases are deduced from the use cases of the *Requirements Analysis* phase. Running all possible tests of all use cases would exceed the scope of this thesis. Therefore, some important scenarios are selected exemplary (see below). Figure 4.50 shows the top level simulation scenario of the ACC system. The top level simulation model contains a *Signal Builder* that is used to apply different scenarios to the *ACC System*-block. The motor torque output of the ACC system is connected to the environment model and is displayed on a scope in combination with vehicle speed, desired speed, and vehicle distance (the distance that the vehicle has covered). The simulation results are also written to the `simout` structure to be able to analyze and print the results from the Matlab main workspace.

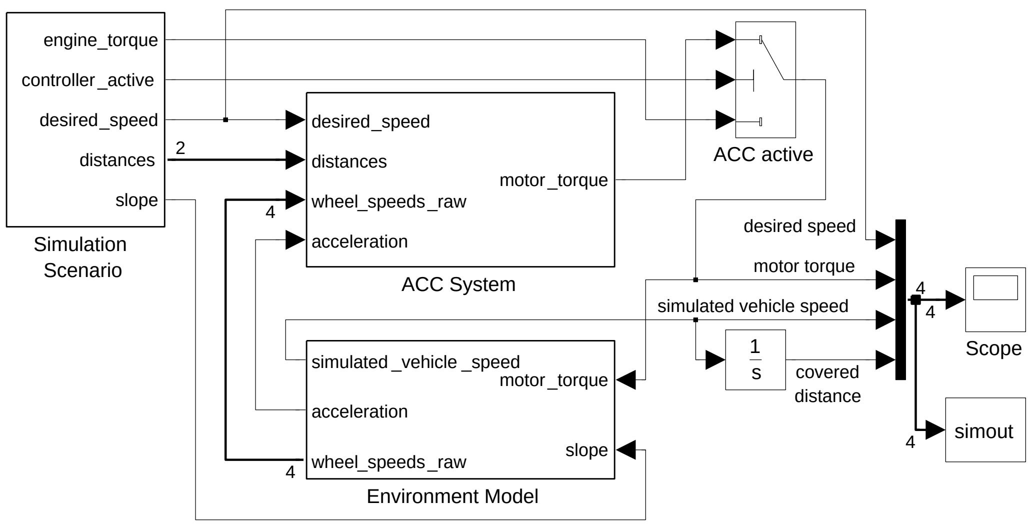


Figure 4.50.: Top level Matlab/Simulink® model for simulation of the ACC system.

Figure 4.51 shows how the outputs of the simulation scenario signal builder are prepared for use together with the ACC system component. The *controller_active* output has to be converted to a boolean. This edge decides if the controller should be enabled at all. With the controller disabled, the environment model and the vehicle plant can be tested and with the controller enabled - after making sure that the vehicle plant behaves like desired - the ACC system controllers can be tested. The signal builder has two outputs for two distance sensors. Those outputs are combined to one edge using a *Mux* block. In contrast, the edges for the motor torque (disabled ACC system), the desired vehicle speed (enabled ACC system), and slope of the ground are connected directly to the system's output.

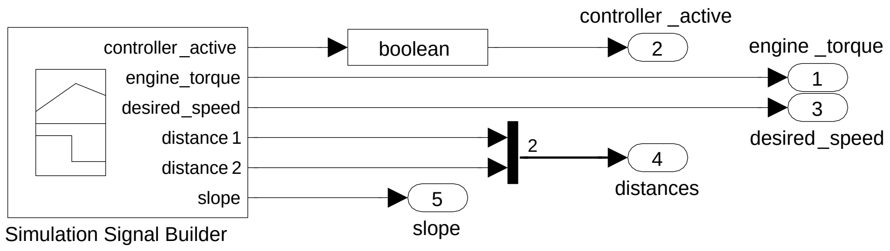


Figure 4.51.: Preparing simulation scenario signals.

As stated before, deducing test cases for all use cases would exceed the scope of this diploma thesis. For this reason, three different simulation scenarios are used exemplary to test the vehicle plant and the controllers:

- First Scenario: Simulating the vehicle plant (see chapter 4.4.3).
- Second Scenario: Simulating the speed controller thus simulating cruise control (see chapter 4.4.3).
- Third Scenario: Simulating the distance controller in combination with the speed controller (see chapter 4.4.3). Thus, the complete adaptive cruise control system is simulated.

Simulating the Vehicle Plant

The scenario that is used for simulation of the vehicle plant is shown in figure 4.52. The scenario does not test a specific use case from *Requirements Analysis* phase but is used to make sure that the plant works which is a precondition for all subsequent tests. The controller is disabled the whole time. Hence, one can analyze the behavior of the vehicle plant.

Initially, the normalized motor torque is negative (breaking) and after a short time the torque is set to 50 percent of the maximum value. Figure 4.53 shows the reaction of the system to the impressed simulation inputs.

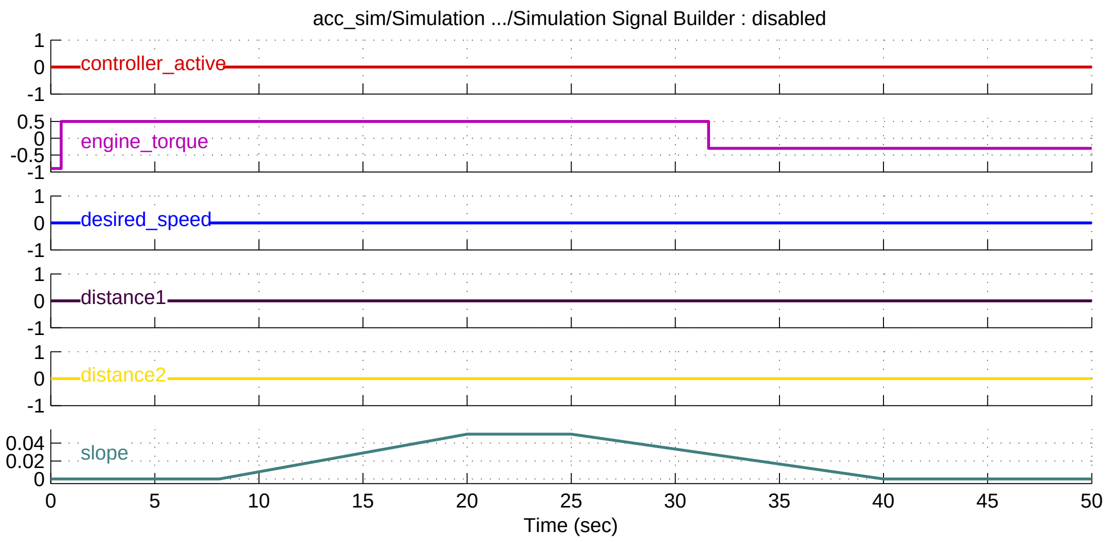


Figure 4.52.: Signal builder for the scenario *controller disabled* i.e. testing the vehicle plant.

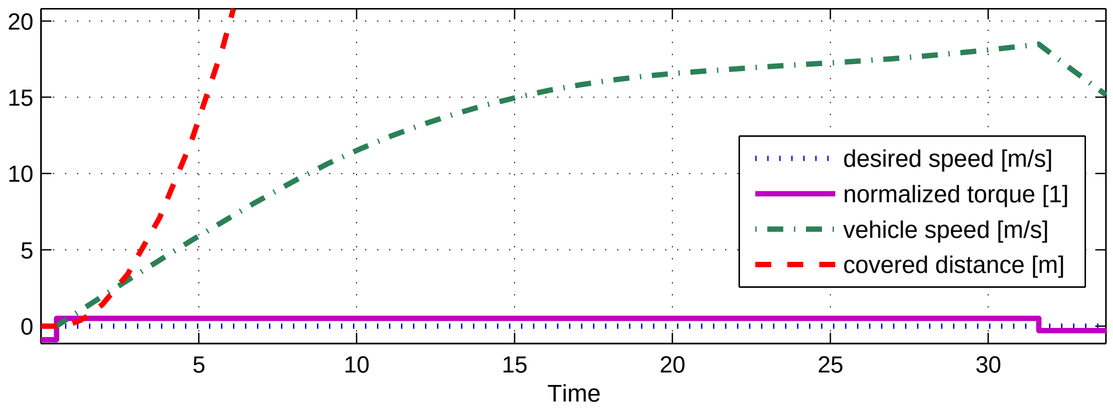


Figure 4.53.: Simulation results for the scenario *controller disabled*.

The vehicle starts accelerating quickly. After eight seconds, the slope value of the ground increases to comparatively high value of five percent. Accordingly, the increase of the speed is reduced. At time 31.6 seconds, the motor torque is set to -0.3 which indicates weak breaking. Therefore, the car stops accelerating and stars decelerating. Consequently, the vehicle plant seems to work as both inputs (motor torque and ground slope) of the environment model were modified and the simulated vehicle behaved like expected.

Simulating Cruise Control

This simulation scenario tests the use case *control speed without headway control* that was defined in the *Requirements Analysis* phase (see chapter 4.1.6). In contrast to the previous simulation, the *controller_active* output of the signal builder for the cruise control scenario is always true (see figure 4.54). Thus, the *motor_torque* edge is not used. On the contrary, the *desired_speed* edge is used now.

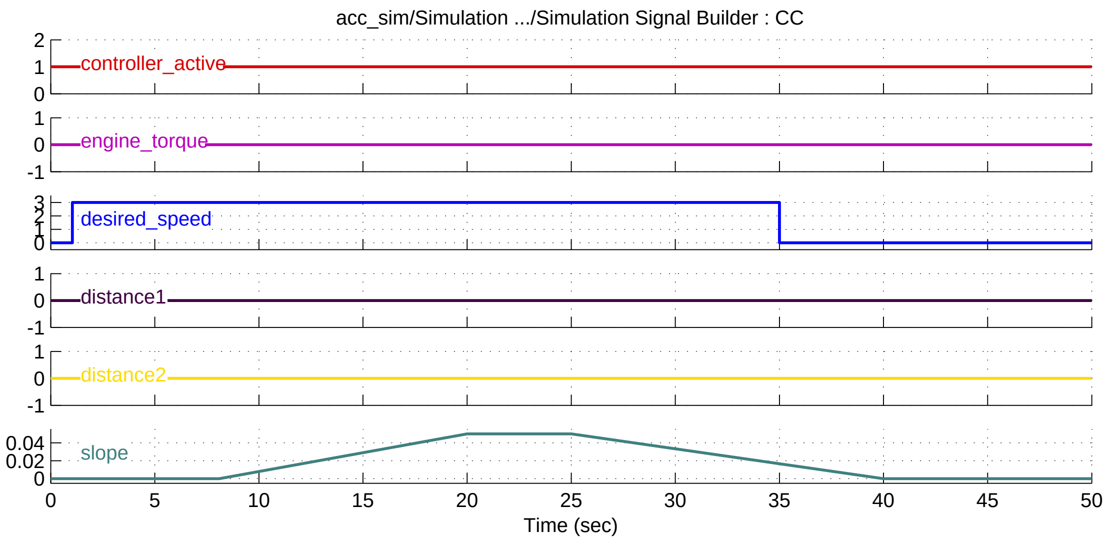


Figure 4.54.: Signal builder for the scenario *cruise control*. Controllers are enabled but there is no obstacle ahead.

After one second, the desired speed steps from $0\,\frac{m}{s}$ to $3\,\frac{m}{s}$. The slope is still configured as explained in the previous scenario. Both distance sensor values are configured to zero which indicates that no obstacle is ahead. Figure 4.55 shows the cruise control simulation results. After the desired speed is set, the controller sets the desired torque to hundred percent. Consequently, the vehicle quickly accelerates towards the desired speed. One second later, the vehicle speed is already more than $2.5\,\frac{m}{s}$ and the torque is reduced by the speed controller to make sure not to overshoot.

In contrast to the previous simulation, the increase of the ground slope results to an increase of the motor torque. Therefore, the vehicle speed does not change essentially and the desired speed is still achieved. This is a huge difference to the original behavior without cruise control and reflects the desired behavior.

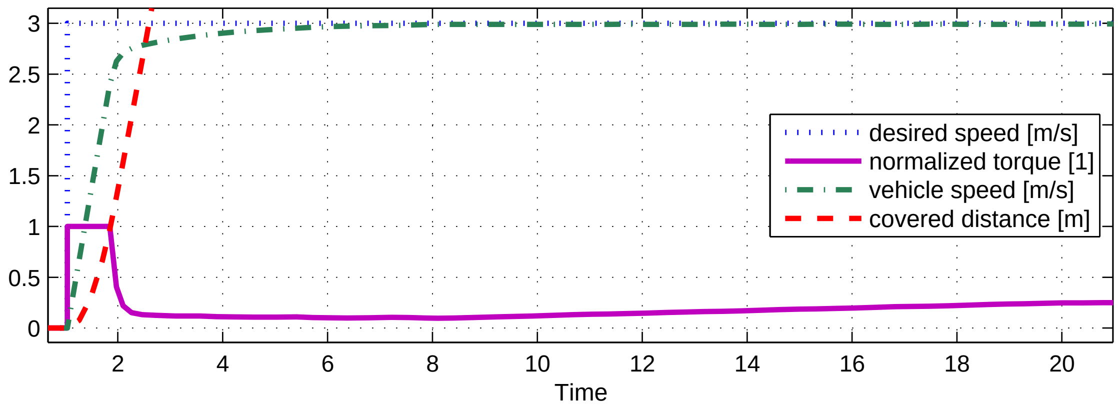


Figure 4.55.: Simulation results for the scenario *cruise control.*

Simulating Adaptive Cruise Control

This simulation scenario is used to tests the use case *control speed with headway control* that was defined in the *Requirements Analysis* phase (see chapter 4.1.6).

The simulation scenario for simulating the adaptive cruise control system looks similar to the scenario of the cruise control simulation. The only difference is that the simulated distance sensors produce values different from zero. A cutting-in predecessor vehicle is simulated by first signalling distance sensor 1 after 6.2 seconds and in the second place distance sensor 2 after 7.3 seconds. As stated before, the maximum range of the distance sensors is six meters. Thus, the sensors start generating values of six meters that decrease as fast as our simulated vehicle catches up its predecessor (see figure 4.56). Both sensors decrease until they reach a spacing distance of three meters between the two vehicles.

Figure 4.57 shows the specific simulation result for the cutting-in predecessor vehicle. Until the predecessor appears, the result is the same as for the cruise control simulation. After the predecessor is visible to the concept car, it brakes until it reaches a safe distance. According to [Kra08], the distance controller was designed to keep a spacing of at least 1.5 times of the vehicle speed. The vehicle speed stabilizes to approximately two meters per second which - in fact - is the appropriate traveling speed for the simulated distance of three meters.

After about 17 seconds, the vehicle ahead gets out of sight, so the ACC system reverts back to cruise control mode and accelerates in the known way until arriving the previously set desired speed of three meters per second.

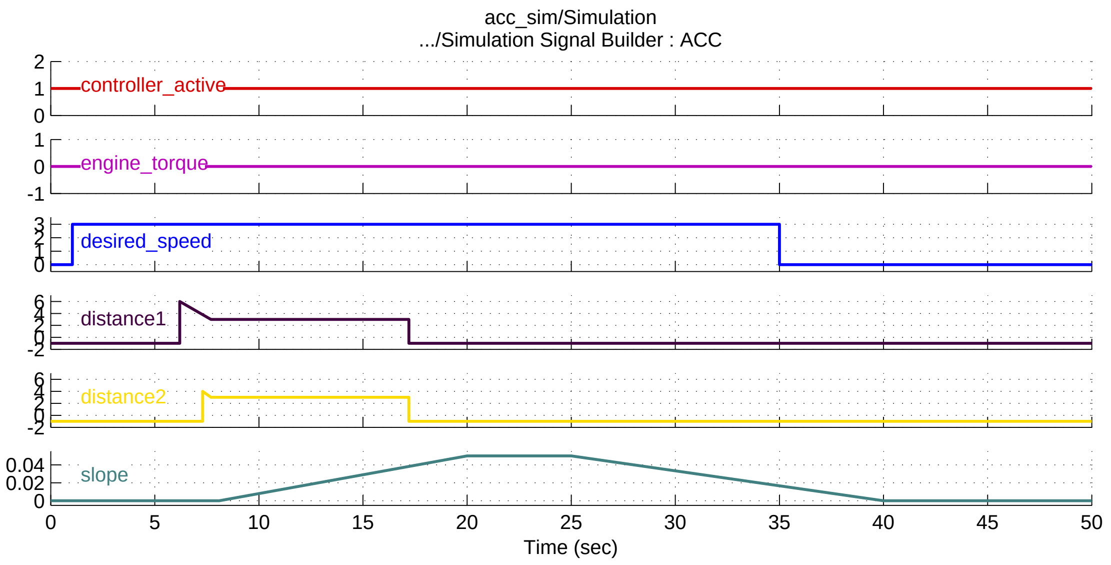


Figure 4.56.: Signal builder for the scenario *adaptive cruise control*. Controllers are enabled and there is an obstacle ahead.

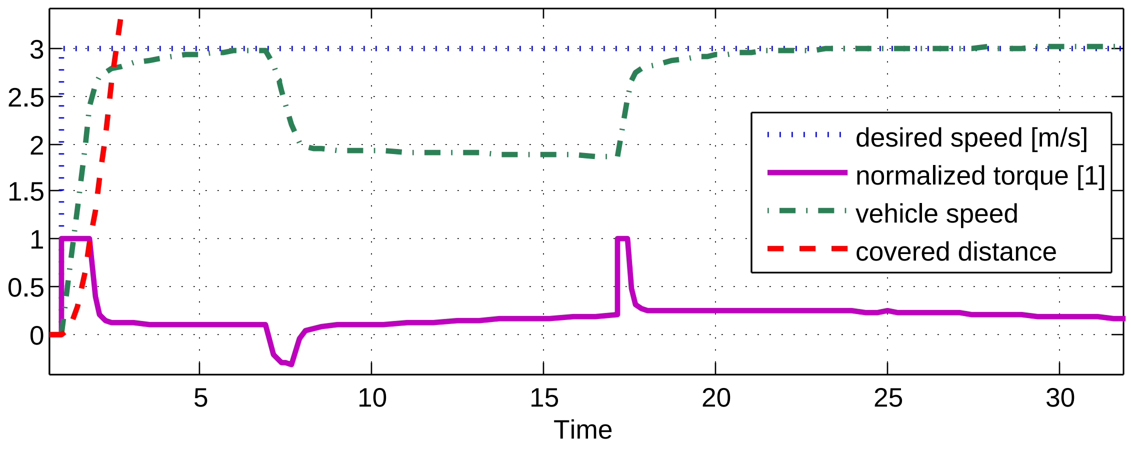


Figure 4.57.: Simulation results for the scenario *adaptive cruise control*.

To sum up the described three simulation scenarios, one can see that in all tested cases the controllers behave like expected. Independent of external influences like slope, the desired speed will be kept. In fact, if a predecessor vehicle is detected, the appropriate distance will be kept to not endanger one of the vehicles. Additionally, switching between cruise control and adaptive cruise control seems to be fully functional.

4.4.4. Platform-Specific Design

After modelling on the application layer is finished, the model is still platform independent. It can be simulated as demonstrated in the previous chapter and code may be generated, but the model is not able to be executed in the target platform. The code that can be generated is just the application code and not the platform code.

It is necessary to extend the structural models by platform-specific aspects as demonstrated in figure 4.58. There are different methods for triggering components:

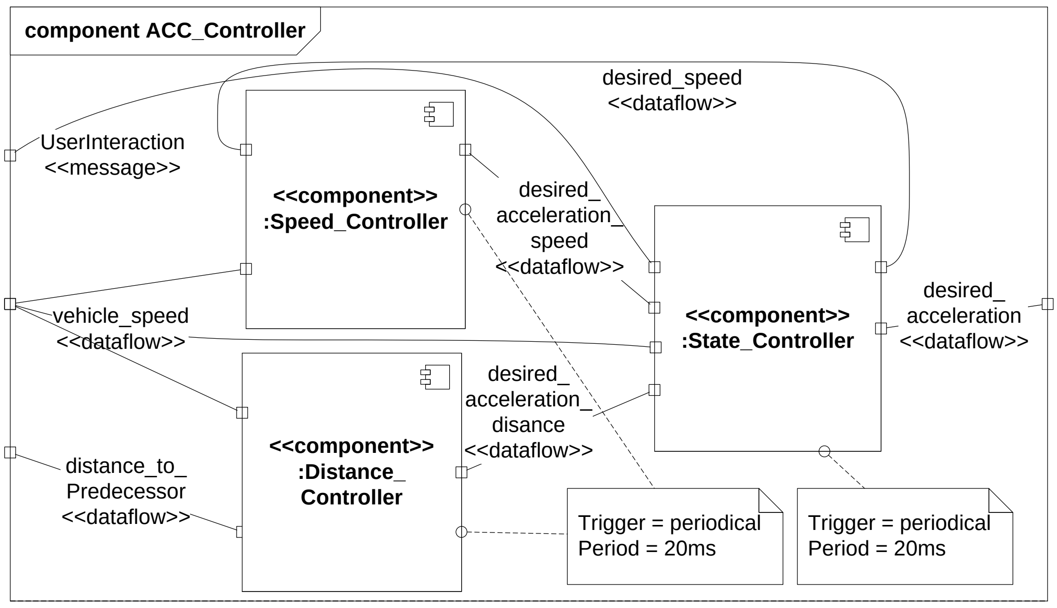

component ACC_Controller
desired_speed <<dataflow>>
UserInteraction <<message>>
<<component>> :Speed_Controller
desired_ acceleration_ speed <<dataflow>>
vehicle_speed <<dataflow>>
<<component>> :State_Controller
desired_ acceleration <<dataflow>>
desired_ acceleration_ disance <<dataflow>>
<<component>> :Distance_ Controller
distance_to_ Predecessor <<dataflow>>
Trigger = periodical Period = 20ms
Trigger = periodical Period = 20ms

Figure 4.58.: Platform specific composite structure diagram of the component *ACC_Controller*.

1. Whenever a message arrives.
2. When the input value changes.
3. Periodically.

In this case, the components will be triggered periodically with a given period of 20 ms because the sensory data of all sensors is received with the same speed. One has to make sure that special hardware restrictions of the used embedded systems are considered. For example, many embedded systems do not have floating-point units and thus using emulated floating-point calculations will slow down the whole system. If fix-point calculations were used instead, the system could run much faster.

In the system, that is developed in the diploma thesis, the embedded hardware (ARM7) does not have a floating-point unit, but single precision floating-point calculations will nevertheless be used because sufficient computational power is available for the desired controller step time. By using floating-point calculations instead of fix-point calculations there will be no special restrictions to the value range of the variables. For example, when using 32 bit fix-point values with 16 bit integer part and 16 bit fractional part, values in the range of 0 to $2^{16} - 2^{-16} \approx 65535.99998$ can be represented at a resolution of $2^{-16} \approx 0.0000152$, which is a resolution of at least four fraction digits. Therefore, single precision floating point numbers are used in the *Platform Specific Design* of this system.

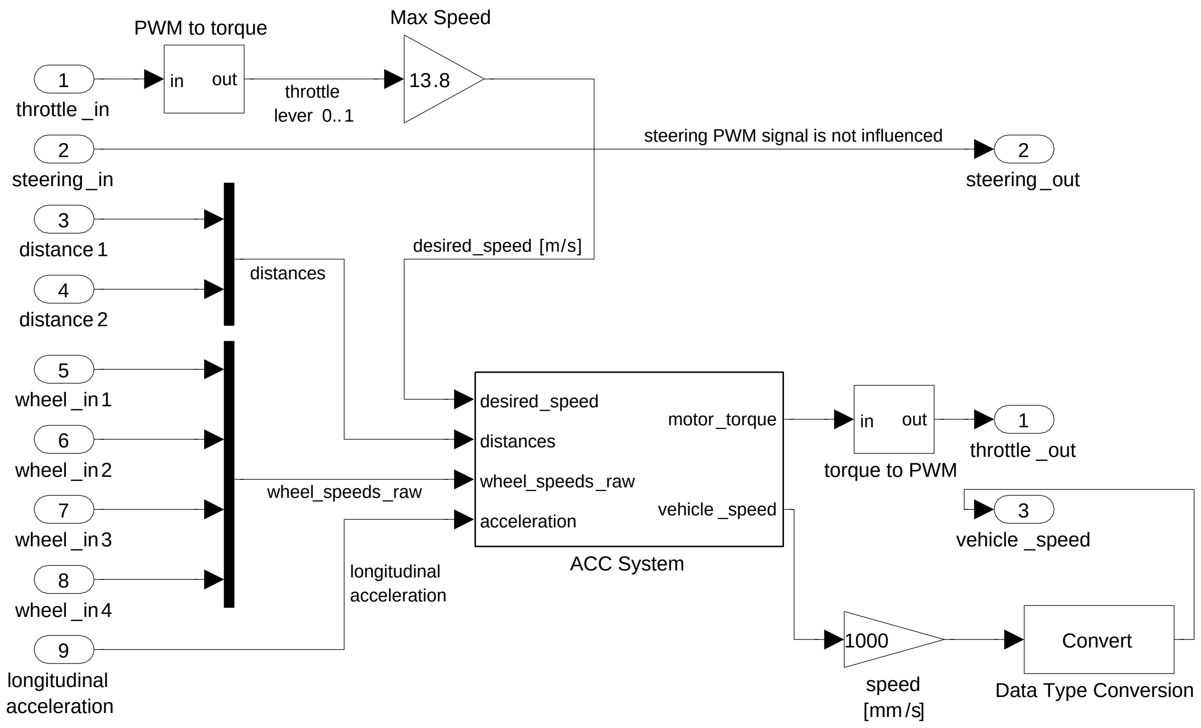


Figure 4.59.: Platform model for using the ACC system on the concept car.

Now, after the adaptive cruise control system has been fully described on platform level, the system has to be integrated into the hardware, the CAN-bus in this case. Each CAN telegram needs its own input and output port. Figure 4.59 shows nine input ports and two output ports. Two distance input ports are combined using a *Mux*-block to one *distances* edge as input for the *ACC_System*. The same is done for the four wheel speed raw edges. The steering signal is directly connected from input port to output port for modifying the CAN identifier for the actuator board. Because the throttle input and output ports are unprocessed, they have to be converted to an useful value range using two simple components:

- Figure 4.60 is used for the conversion of CAN messages containing the PWM values to a throttle value in the range 0 to 1 for acceleration. Negative values represent the users desire to break.

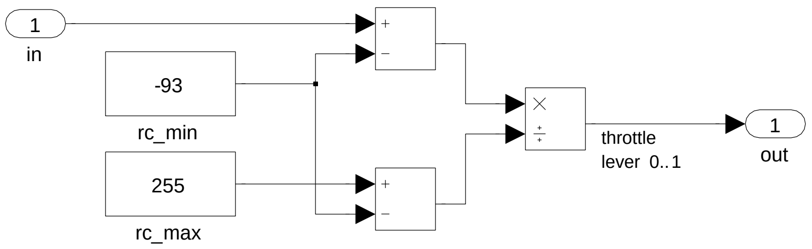


Figure 4.60.: Simple component that converts the incoming PWM CAN messages to normalized throttle.

- In contrast to the previous component, figure 4.61 shows the converter that is used for the inverse operation. Normalized throttle values are converted back to PWM values for using them in the outgoing CAN telegrams.

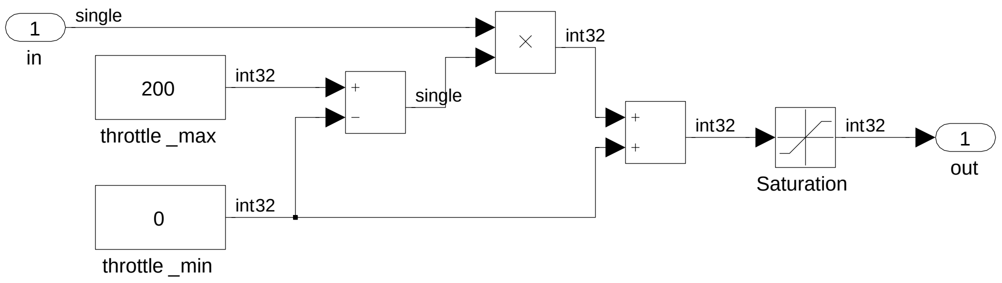
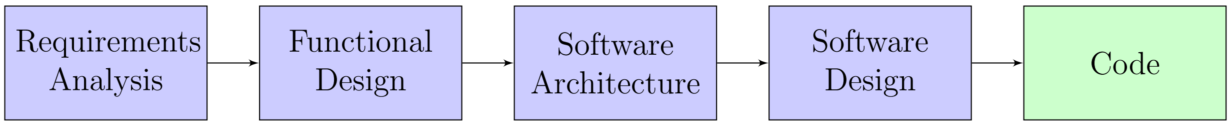


Figure 4.61.: The desired throttle is converted into appropriate CAN messages.

4.5. Code



After all steps that have been taken, the *Code* phase begins. The input for this phase is the platform specific behavioral model of the *Software Design* phase. The output is the finished source code that can be compiled and executed in the target system. A vast benefit of the used model-based design approach is the short *Code* phase. Code generators are used for generating the system's code. As stated before, code generators are used for generating mainly the application code from the models. Nowadays, generators are able to generate efficient and platform independent application code, but they are not able to generate the platform code and operating system code [BST09]. Integration of operating system code, platform code, and application code is mostly done manually because the output of the code generator is mainly a template that still has to be integrated to be able to be executed.

In this thesis, the application code generation is performed by Real-Time Workshop® Embedded Coder™ in chapter 4.5.1. Afterwards, the integration of application-, platform-, and operating system code is done with the support of an additional tool in chapter 4.5.2.

4.5.1. Real-Time Workshop® Embedded Coder™

In this thesis the code generator Real-Time Workshop® Embedded Coder™ is utilized. It is able to generate efficient C code from Matlab/Simulink® and Stateflow® models [Mat].

Before being able to start the generation process, Real-Time Workshop® has to be configured appropriately: See appendix A for configuration details. After configuration is finished correctly, the source code is generated from the model. The continuous dataflow of the behavioral models is converted to source code. Appendix C describes in detail which files are generated and what they are used for.

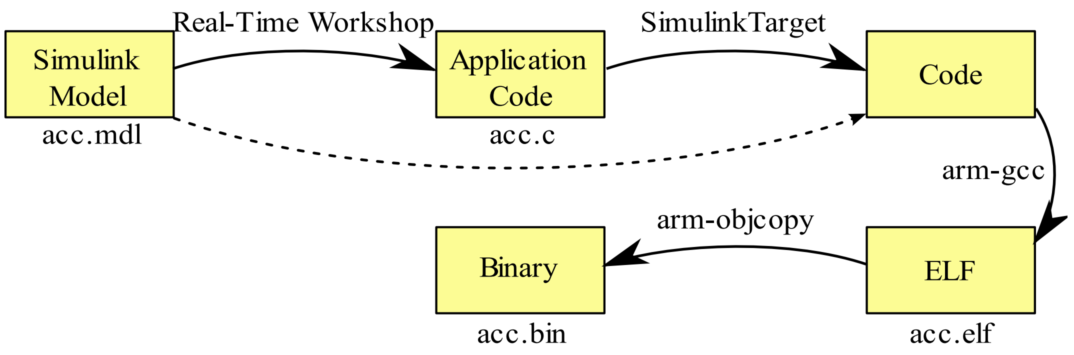


Figure 4.62.: Generating an executable ARM binary from a Matlab/Simulink® model.

4.5.2. Integration and Compilation

After the application code files are generated, they have to be combined with platform code and operating system code. The generated files and the model file are fed into the Java program called *SimulinkTarget* developed by Donald Barkowski of the Fraunhofer IESE. *SimulinkTarget* merges the generated code with the operating system and platform code and compiles a stand-alone binary image (see figure 4.62). As the generated files and their contents do not need to be reviewed (at least as long as the toolchain and the platform are working properly) the detailed information is described in appendices: The source that is generated by *SimulinkTarget* is described in appendix E. Furthermore, appendix F shows the exemplary contents of the main function that is generated by *SimulinkTarget*. To be able to compile ARM binaries out of the generated source code, an ARM toolchain is needed. In this thesis, the free GNU Compiler Collection[GCC] toolchain is used.

After successful compilation, the generated image file has to be copied into the root directory of a SD card and the card has to be plugged into the card reader slot of the Olimex SAM7-LA2 board [Oli08]. The bootloader on the ARM7 board will search for the image file and will copy it to the microcontroller static memory. From where it will be executed.

Finally, the model-based software development process for embedded systems is finished. As one can see, the *Code* phase of the process is short and robust. This is one of the major advantages of the used approach.

Before being able to test the whole system on the target platform, the concept car had to be set up properly. A new revision of *SensorBoards* and *ActuatorBoard* were designed and soldered and their code was reworked. The wheel speed sensors and the distance sensors had to be attached to the car. Severe communication issues of the CAN bus had to be fixed to make sure that no CAN telegrams were dropped and thus, the concept car did not misbehave.

5. Evaluation of the Development Process

Model-based development of embedded systems as described in [BST09] provides great advantages over the traditional textual approach. These enormous benefits were described in this thesis. This chapter gives an evaluation on the described development process and its sub-steps. On the whole, it is informative to see the temporal distribution of all process steps.

As the model-based development of this vehicle control system was my first development in this domain, the effort that was put in each step might not be representative. Chapter 5.1 provides a reflection on the development and analyzes the overall effort for this development. Afterwards, chapter 5.2 pictures the relative distribution of consumed time of all process steps under the precondition that the system will be constructed by experienced personnel with domain knowledge. Finally, an estimation of effort distribution for further developments of vehicle control systems on the same target platform is presented (see chapter 5.3).

5.1. Time Distribution of the Overall Process

Initially, I would like to provide a reflection on the development process. The model-based development of the adaptive cruise control system was my first software development in this domain. Additionally, I experienced some problems while setting up the platform and building the necessary toolchain. Taking these preconditions into consideration, this chapter analyzes the overall time consumption of all process steps. The time that was used to solve the problems is included although these issues are not directly related to the development process itself. Table 5.1 shows the time that was consumed by each step:

Phase	Time (days)
Requirements Analysis	3
Functional Design	25
Software Architecture	5
Software Design	14
Code	36

Table 5.1.: Time consumption of each process step under the given preconditions.

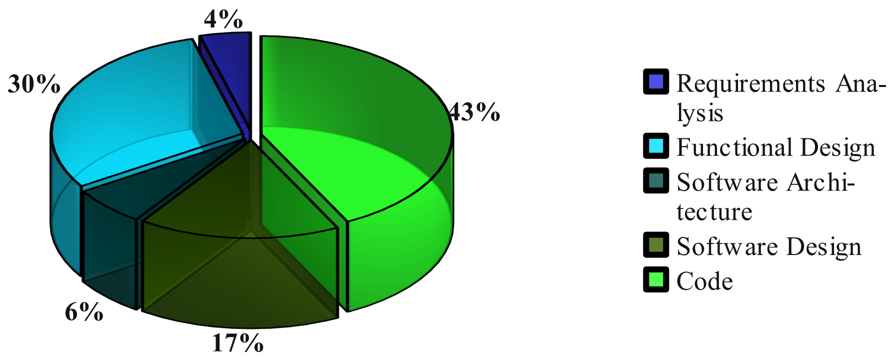


Figure 5.1.: Time distribution of the different process phases.

The *Requirements Analysis* phase took not much time because the adaptive cruise control systems complexity is not that big (see figure 5.1). The *Software Architecture* phase was relatively short for the same reason. The *Function Design* phase took much more time because two feedback loop controllers and one filter were developed. As I was relatively unexperienced in Matlab/Simulink® and just knew the basics, I had to learn the details first. All controllers and filters had to be parameterized, simulated, and tuned. Additionally, the vehicle plant was physically deduced in detail to be able to run realistic simulations and tune parameters in the model. The *Software Design* phase took about 17 % of the whole time. The *Structural Refinement* step and *Platform Specific Design* step consumed less time than the *Behavioral Design* step because in the behavioral design step, the behavior of the whole system was modeled and simulated. In this step, plenty of block diagrams were modeled, partially modified, and reused from the *Functional Design* step.

Last but not least, the *Code* step consumed by far the most time because enormous effort was made to achieve that the platform executes the produced binary image as expected. Initially, the microcontroller that executes the model had problems to receive the CAN telegrams properly. Additionally, the ARM specific toolchain was setup with several issues concerning linking and floating-point emulation. Fixing such issues set me back but finally the model can be executed properly.

Normally, the *Code* phase should be much shorter as this is the main advantage of model-based software development. Therefore, the previous view on the development process is a little bit distorted and normally the results would look slightly different. These differences are described in the next chapter.

5.2. Normal Time Distribution for Domain Experienced Personnel

As described before, this chapter analyzes the time distribution for personnel that is experienced in developing systems in this target domain. All activities that were not directly related to the process are not taken into consideration in this chapter.

Figure 5.2 shows the estimation of the time-distribution for the development process. The first thing that one can see is the vast reduction of the *Code* step of the model-based process. Putting aside all the problems that I encountered while trying to generate the binary images and run them in the target system leads to a really short *Code* phase - as expected for model-based development. When combining platform code with the generated application code is done by a specific tool (*SimulinkTarget* in my case), the code phase may just consist of generating the application code in Matlab/Simulink® and running this tool. The output of the tool is the properly compiled binary. Thus, there is no need to touch code at all and the *Code* phase may be even shorter than in the pie chart 5.2.

Additionally, as I am not a domain expert of vehicle control systems or control theory in general, the *Functional Design* phase should take less time of the overall process than it

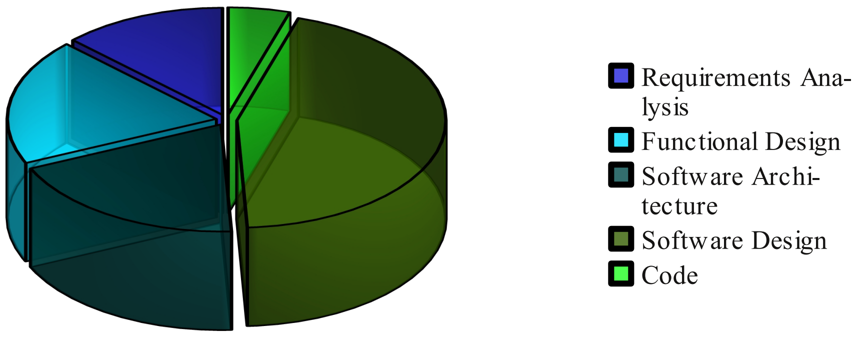


Figure 5.2.: Time distribution estimation under the precondition of experienced personnel and domain experts.

took in my case. I had to figure out some basics like the applied anti-wind-up algorithm. This consumed time that a control systems engineer could have saved.

5.3. Estimated Time Distribution for Further Developments of Vehicle Control Systems on the same Target Platform

After describing the previous distributions, there is one final estimation left over: Figure 5.3 shows the estimation of the time-distribution for the model-based development of a second vehicle control system using the same approach on the same target platform - the concept car of the Fraunhofer IESE. As the precondition for the following considerations is that the development target platform is the Fraunhofer IESE concept car and the control system is unspecified, the resulting numbers are just speculative.

As the code generation process is stable now and the proper execution on the target platform is ensured, the *Code* step should consume - in contrast to my preparatory work - by far the least time of all process steps. The *Code* step is simply reduced to building the code from Real-Time Workshop® Embedded Coder™ and additionally executing *SimulinkTarget* as described in chapter 4.5.2.

Additionally, the absolute time of the *Functional Design* step is reduced for two reasons: On the one hand, the vehicle plant is fully deduced now and thus no more effort has to be put into this part - regardless of which vehicle control system has to be developed. On the other hand, the wheel speed data processing is already working. Most of all driver

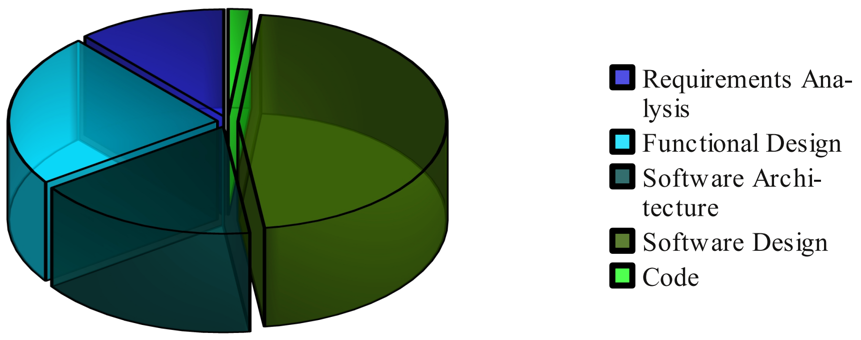


Figure 5.3.: Effort estimation for the model-based development of a second vehicle control system in the Same Domain.

assistance systems need the vehicle speed as input. Thus, the *DataAcquisition* component output that describes the vehicle speed can be connected to all vehicle control systems. The wheel speed raw processing may still be optimized and tuned but for most systems the deduction of a wheel speed filter is no longer required.

Pie chart 5.3 emphasizes one of the advantages of the used approach: The *Code* phase is extremely short and no software engineer nor a control systems engineer needs to worry about how to implement and simulate controllers and filters. This is done in the early *Functional Design* phase for detecting and correcting errors earlier. Therefore, the overall cost- and time-consumption is reduced.

6. Conclusion and Future Work

6.1. Conclusion

The task of this diploma thesis was to develop an adaptive cruise control system using a specific model-based development process for embedded systems [BST09]. The system that was developed runs on a remote-controlled one-to-five-scale concept car. The used approach simplified the development of the vehicle control system by assuring that the controllers could be tested in early design phases. As the *Functional Design* step is processed immediately after the *Requirements Analysis* phase, I was able to detect and correct errors in the vehicle plant and in the controllers in this early design phase.

First, the requirements of the adaptive cruise control systems were analyzed. Afterwards, the vehicle plant was physically deduced to enable detailed simulation of all controllers. This deduced plant was tested using several scenarios. A wheel speed sensor filter was designed to transfer the noisy raw wheel speed data into a reliable vehicle speed value. Two controllers were developed: The speed controller that controls the vehicle to a defined set-speed was modeled and simulated. The distance controller that makes sure that the concept car never comes too close to the vehicle ahead was constructed.

After the controller parameters were tuned, the ACC system architecture was analyzed and all components were decomposed to reduce their complexity. The static decomposition and the dynamic communication, were defined. Afterwards, the behavior of all components was described and the controllers were added to the ACC system. Using the fully designed system, the complete adaptive cruise control functionality was simulated in different scenarios to make sure that the system behaved like expected from the requirements analysis phase. The working system was adapted considering platform specific modifications like timings, input-ports, and output-ports. Finally, code was generated from this modified model by Real-Time Workshop® Embedded Coder™. The generated code was merged with the platform specific code. Using an ARM toolchain, the appropriate binary was successfully generated.

After code generation, different tests were performed with the target platform. All tests were passed and the adaptive cruise control system appears to work properly.

By using the described development process for embedded systems, I was able to develop a whole adaptive cruise control system in a top-down design approach. Although running this waterfall model without using any iterations, the system is fully functioning in the end. This seems to be the benefit of the applied model-based development process [BST09].

As an advantage of model-based software development approaches, the vehicle plant and controllers that were developed in this diploma thesis can be reused and extended in future systems.

6.2. Future Work

Extensions of the developed adaptive cruise control system are conceivable: *Dynamic Set Speed Type* ACC systems use the global positioning system to determine the positions of speed sign from a database to decide the maximum available speed to travel at. *Cooperative ACC Systems* increase reliability because the vehicles may communicate with each other. This leads to faster travel speeds as the needed time gap between two vehicles can be reduced to half a second. One could also imagine ACC systems that look the ground slope up from a database to improve controller qualities.

As the concept car just uses the brushless controller to break at the moment, the deceleration is not optimal. In future, hydraulic front-wheel brakes should be added to the vehicle to make sure that the deceleration is sufficiently large to escape from risky situations. Those hydraulic brakes may individually be controlled. Thus, the concept car would have three actuators attached to the CAN bus that can be used to break. Therefore, the braking models have to be adapted and oversteering could be actively suppressed.

The estimated distribution of time for personnel that is experienced in developing systems in this target domain could be verified. Analyzing a development process with domain experts and comparing the resulting distribution with the values figured out in this thesis could be interesting.

There are plenty driver assistance systems that may also be be developed using this model-based approach for embedded systems. After the preparatory work of this thesis, the development of further systems will be much simpler because the vehicle plant can be reused and controllers and filters can be extended.

Bibliography

[Ana07] Analog Devices, Inc. *ADIS16006 Dual-Axis Accelerometer Data Sheet*, A edition, December 2007.

[Ana09] Analog Devices, Inc. *ADIS16100 Yaw Rate Gyroscope Data Sheet*, D edition, December 2006-2009.

[Atm07] Atmel Corporation. *AT91SAM7A2 Microcontroller Data Sheet*, B edition, March 2007.

[Atm08] Atmel Corporation. *AT90CAN128 Microcontroller Data Sheet*, H edition, August 2008.

[BJR08] Grady Booch, Ivar Jacobson, and Jim Rumbaugh. *OMG Unified Modeling Language Specification.* Object Management Group, Inc., 1.3 edition, August 2008.

[BS05] B. Bouyssounouse and J. Sifakis. *Embedded Systems Design.* Springer Berlin / Heidelberg, 2005.

[BST09] K. Berns, B. Schürmann, and M. Trapp. *Eingebettete Systeme: Systemgrundlagen und Entwicklung eingebetteter Software.* 2009.

[CB95] D.P. Atherton C. Bohn. *An analysis package comparing PID anti-windup strategies.* 1995.

[CC] Concept Car of the Fraunhofer IESE. http://conceptcar.iese.de.

[Chr05] D. Christen. *Praxiswissen der chemischen Verfahrenstechnik, Einstellregeln für Industrielle Regler.* Springer Berlin Heidelberg, 2005.

[Dav09] Davantech Ltd. *SRF02 Distance Sensor Data Sheet*, 2009.

[GCC] GNU Compiler Collection. http://www.gnu.org - Free Software Foundation.

[Hug08] J. Hugh. *Automating Manufacturing Systems with PLCs.* 2008.

[Kar00] N. Karim. *How Tires Work.* 2000.

[Kon06] Kontronik GmbH. *Power JAZZ 63V Manual*, D edition, October 2006.

[Kra08] U. Kramer. *Kraftfahrzeugführung.* Carl Hanser Verlag München, 2008.

[Lun06] J. Lunze. *Regelungstechnik 2: Mehrgrößensysteme, Digitale Regelung.* Springer Berlin Heidelberg, 2006.

[Lun08] J. Lunze. *Regelungstechnik 1: Systemtheoretische Grundlagen, Analyse und Entwurf einschleifiger Regelungen.* Springer, Berlin, 7 edition, August 2008.

[Mat] The MathWorks, Inc. *Real-Time Workshop - Generate C Code from Simulink Models and Matlab Code.*

[Mat04] The MathWorks, Inc. *Simulink. Simulation and Model-Based Design*, 2004.

[Oli08] Olimex Ltd. *SAM7-LA2 Development Board Users Manual*, A edition, July 2008.

[Tem97] Temic Semiconductors. *CNY70 Reflective Optical Sensor Data Sheet*, A2 edition, December 1997.

[Tra05] M. Trapp. *Modeling the Adaptive Behavior of Adaptive Embedded Systems.* PhD thesis, TU Kaiserslautern, 2005.

[ZS08] Werner Zimmermann and Ralf Schmidgall. *Bussysteme in der Fahrzeugtechnik - Protokolle und Standards.* Vieweg+Teubner, 3rd edition, 2008.

List of Tables

4.1. Table of compare-rolling resistance coefficients like described in [Kar00]. . . 37
4.2. Ziegler-Nichols rules for determining controller parameters [Chr05]. 52

5.1. Time consumption of each process step under the given preconditions. . . . 90

D.1. System hierarchy for the ACC model. 111

List of Figures

2.1. Modeling techniques classification. 15

3.1. The concept car target platform of the Fraunhofer IESE. 17
3.2. Concept car drive by wire structure. 19
3.3. Pulse width modulation signal generated by the remote control receiver. . . 20
3.4. One wheel with its attached black and white encoder strip. 20
3.5. CNY70 reflective optical sensor that is used for rotation speed detection. . 20
3.6. ADIS16100 rotary sensor and ADIS16006 acceleration sensor. 21
3.7. *SensorBoard* that sends acceleration and rotary data to the CAN bus. . . . 21
3.8. Two of those *SRF02* distance sensors are used for measuring the distance to the predecessor vehicle. 22
3.9. Diagram of the brushless motor *1930/9 LK* from *Lehner Motoren Technik*. 23
3.10. Development board *SAM7-LA2* from *Olimex Ltd*. 25

4.1. Use case diagram for the adaptive cruise control system. 29
4.2. Sequence diagram of the scenario *adjust desired speed*. 30
4.3. Sequence diagram of the use case *brake*. 31
4.4. Sequence diagram of the scenario cruise control (*control speed without headway control*). 33
4.5. Sequence diagram of the scenario *adaptive cruise control*. 34
4.6. Different drag coefficients. 38
4.7. Matlab/Simulink® model of the vehicle plant. 39
4.8. General approximated rotational speed-time-diagram for brushless motors. 40
4.9. Rotary-speed-torque-diagram for the brushless motor that is used on the concept car. 41
4.10. Matlab/Simulink® model that calculates the available motor torque in dependence upon the vehicle speed. 41
4.11. Simulation model for the wheel speed sensors. 42
4.12. Unprocessed wheel speed raw data. 42
4.13. Processed wheel speed data after conversion to rotations per second. . . . 43
4.14. Matlab/Simulink® model (*calculate_wheel_speed*) for processing the raw wheel speed sensory data. 43

4.15. A simple generic control loop. 44
4.16. PID controller block diagram. 46
4.17. PID controller response for different coefficients. 47
4.18. A saturating PID controller with tracking anti-windup algorithm is used in the adaptive control system for controlling speed and distance. 49
4.19. Closed loop for finding the speed controller PID parameters. 49
4.20. Matlab/Simulink® signal builder block used as input for parameter determination of the Ziegler-Nichols method. 50
4.21. Initial closed loop response for $K_p = 0.1$. 51
4.22. Closed loop response for $K_p = 1$. 51
4.23. Oscillating response for $K_p = 3.5$. 51
4.24. Response for parameters determined with Ziegler-Nichols method. 51
4.25. Closed loop step response of the speed controller after parameter tuning. . 52
4.26. Closed loop for finding the distance controller parameter. 53
4.27. Closed loop using a proportional controller with $K_p = 1$. 55
4.28. Closed loop starts oscillating for a proportional coefficient of 2.5. 55
4.29. Zoomed view on the oscillation for determining the critical period. 55
4.30. Final closed loop response of the distance controller. 56
4.31. Structure of the complete adaptive cruise control embedded system. 58
4.32. Decomposition of the ACC System. 60
4.33. Type system of the adaptive cruise control system. 61
4.34. Composite structure diagram of the adaptive cruise control system. 63
4.35. Refined sequence diagram of the scenario *Control speed with distance*. . . . 64
4.36. Decomposition of the component *ACC_Controller*. 65
4.37. Decomposition of the component *DataAcquisition*. 66
4.38. Composite structure diagram of the component *ACC_Controller*. 67
4.39. Static decomposition of the component *DataAcquisition*. 68
4.40. Sequence diagram of the component *Speed_Controller*. 69
4.41. Sequence diagram of the component *DataAcquisition*. 70
4.42. Block diagram of the component *ACC_System*. 71
4.43. Matlab/Simulink® model of the component *ACC_Controller*. 72
4.44. The state controller block diagram. 72
4.45. Block diagram of the component *SpeedController*. 73
4.46. Block diagram of the component *DistanceController*. 73
4.47. Block diagram of the component *DataAcquisition*. 74
4.48. Calculates the distance to predecessor. 74
4.49. Matlab/Simulink® model *calculate_vehicle_speed*. 75
4.50. Top level Matlab/Simulink® model for simulation of the ACC system. . . . 76
4.51. Preparing simulation scenario signals. 77
4.52. Signal builder for the scenario *controller disabled* i.e. testing the vehicle plant. 78

4.53. Simulation results for the scenario *controller disabled.* 78
4.54. Signal builder for the scenario *cruise control.* Controllers are enabled but there is no obstacle ahead. 79
4.55. Simulation results for the scenario *cruise control.* 80
4.56. Signal builder for the scenario *adaptive cruise control.* Controllers are enabled and there is an obstacle ahead. 81
4.57. Simulation results for the scenario *adaptive cruise control.* 81
4.58. Platform specific composite structure diagram of the component *ACC_Controller.* 82
4.59. Platform model for using the ACC system on the concept car. 83
4.60. Simple component that converts the incoming PWM CAN messages to normalized throttle. 84
4.61. The desired throttle is converted into appropriate CAN messages. 84
4.62. Generating an executable ARM binary from a Matlab/Simulink® model. . 86

5.1. Time distribution of the different process phases. 90
5.2. Time distribution estimation under the precondition of experienced personnel and domain experts. 92
5.3. Effort estimation for the model-based development of a second vehicle control system in the Same Domain. 93

A. Necessary Real-Time Workshop® Configuration Settings

Before being able to start the code generation, Real-Time Workshop®has to be configured appropriately. In the *Model Explorer* section *Configuration* are some configuration options to take care of:

- Configuration section *Solver*: The *ode3 (Bogacki-Shampine)* solver is used as fixed-step type with 20 ms sample time. This sample time will later be used as interval for the generated code that calls the `model_step`-function.

- Section *Hardware Implementation*: The device vendor in this diploma thesis is *ARM compatible* and the device type has to be set to *ARM 7* with a native word size of 32 bit. Byte ordering is set to *little endian* and emulation hardware is set to *none*.

- The *Real-Time Workshop* section has plenty configuration options. Thus, it is structured into different tabs:

  – Tab *General*:

    * System target file: `ert.tlc`

    * Language: C

    * Compiler optimization level: Optimizations on (faster runs)

    * Generate makefile: Disabled

  – Tab *Interface*:

    * Target function library: GNU99 (GNU)

    * Floating-point numbers: Enabled
    * Non-finite numbers: Enabled (used for e.g. integrator saturation)
    * Continuous time: Enabled
    * GRT compatible call interface: Disabled
    * Single output/update function: Enabled
    * Interface: None
- Tab *Templates*: The only important thing in this tab is to deselect the option whether to generate an example main program. The main program will automatically be generated when adding input- and output ports etc.

B. Concept Car CAN Overview

CAN identifiers that are used on the concept car.

ID	Source	Interval	Description
0x025	SensorBoard 1	20ms	PWM signal steering
0x008	SensorBoard 1	20ms	Wheel speed front left
0x00b	SensorBoard 1	20ms	Wheel speed rear right
0x022	SensorBoard 2	20ms	PWM signal throttle
0x009	SensorBoard 2	20ms	Wheel speed front right
0x00a	SensorBoard 2	20ms	Wheel speed rear left
0x010	InertialBoard	20ms	Longitudinal acceleration
0x011	InertialBoard	20ms	Shear acceleration
0x012	InertialBoard	20ms	Rotation speed
0x125	ControllerBoard	20ms	Processed steering signal
0x122	ControllerBoard	20ms	Processed throttle signal
0x400	*all boards*	-	Error code

C. Description of the Generated Files

The following source files are automatically generated from the platform specific Matlab/Simulink® model `acc.mdl` by Real-Time Workshop® Embedded Coder™:

- `acc.c`: This file is the main file. This file contains all behavioral code for the whole model. First, `acc_initialize` initializes the whole model including its working variables, non-finite constants, the solver, states, input-, and output-ports. After the model is properly initialized, inputs may be written to the input ports (e.g. `acc_U.throttle_in`) and the step function `acc_step` may be called subsequently. When the step function returns, the output values may be read from the output structure `acc_Y` (e.g. `acc_Y.throttle_out`).

- `acc.h`: Defines all necessary structures for the adaptive cruise control system. The most important structure is the `Parameters_acc` that is used for giving the data values of file `acc_data.c` the appropriate semantics. The file `acc.h` also describes a list of assignments of system indices (e.g. `<S1>`) to the systems names (e.g. *acc/ACC System*) for tracing the generated code back to the model (see below).

- `acc_data.c`: Contains all parameters of all blocks (e.g. saturation limits of integrators) and constants of the model. Over 30 constants are used in the ACC System.

- `acc_private.h`: Macro definitions for simplified model access.

- `autobuild.h`: Not used.

- `rt_nonfinite.c` and `rt_nonfinite.c`: Support for non-finite numbers like *Inf* (∞), *NaN*, and *-Inf* ($-\infty$).

- `rtwtypes.h`: Contains basic type definitions like boolean types, integer, and floating point types but also complex number types in different formats.

- `acc_types.h`: Type definitions for non-basic types that are used and defined in the ACC system model.
- `rt_defines.h`: Contains common mathematical definitions like constants (π, ln(10), e...) and simple functions (abs, max, min ...).

D. Tracing Code Back to the Model

The generated code includes comments that allow one to trace back to the appropriate location in the model. The basic format is `<system>/block_name`, where `system` is the number of the component. The numbers are uniquely assigned by Matlab/Simulink®. `block_name` is the name of the block inside this system. Table D.1 shows the hierarchic list of all systems.

System ID	Hierarchic system name
`<Root>`	`acc`
`<S1>`	`acc/ACC System`
`<S2>`	`acc/PWM to torque`
`<S3>`	`acc/torque to PWM`
`<S4>`	`acc/ACC System/ACC Controller`
`<S5>`	`acc/ACC System/DataAcquisition`
`<S6>`	`acc/ACC System/ACC Controller/Distance Controller`
`<S7>`	`acc/ACC System/ACC Controller/Speed Controller`
`<S8>`	`acc/ACC System/ACC Controller/State Controller`
`<S9>`	`acc/ACC System/ACC Controller/Distance Controller/controller`
`<S10>`	`acc/ACC System/ACC Controller/Speed Controller/controller`
`<S11>`	`acc/ACC System/ACC Controller/State Controller/too near`
`<S12>`	`acc/ACC System/DataAcquisition/calc distance_to_predecessor`
`<S13>`	`acc/ACC System/DataAcquisition/calc wheel_speed`
`<S14>`	`acc/ACC System/DataAcquisition/calc vehicle_speed`

Table D.1.: System hierarchy for the ACC model.

The Matlab `hilite_system` command may be used to trace the generated code back to the model (e.g. `hilite_system('<S1>/block1')`).

E. Summary of Files Generated by SimulinkTarget

In the *Code* phase, the tool *SimulinkTarget* creates an output directory in the format `CODEGEN_$DATE_$TIME` and produces the following merged file tree into this folder:

```
CODEGEN_09-07-24_02-09-43
+--include
|  +--base: at91sam7a2_adc.h at91sam7a2_can.h sdc.h global.h at91sam7a2_usart.h
|  |        at91sam7a2_exceptions.h at91sam7a2_interrupts.h at91sam7a2_wt.h
|  |        at91sam7a2_pdc.h at91sam7a2_pio.h ssc.h global.h delay.h
|  |        at91sam7a2_pwm.h at91sam7a2_spi.h at91sam7a2_timers.h fat16.h
|  +--usr:  acc.h acc_private.h acc_types.h autobuild.h log.h debug_ports.h
|  |        rt_defines.h rt_nonfinite.h rtlibsrc.h messageIDs.h process.h
|  |        rtw_continuous.h rtw_solver.h rtwtypes.h main.h
|  +--util: util.h
+--lds:     elf32-littlearm-boot.lds elf32-littlearm-usr.lds
+--src
|  +--base: at91sam7a2_adc.c at91sam7a2_can.c delay.c at91sam7a2_exceptions.c
|  |        at91sam7a2_pdc.c at91sam7a2_pio.c fat16.c at91sam7a2_interrupts.c
|  |        at91sam7a2_pwm.c at91sam7a2_spi.c sdc.c at91sam7a2_wt.c
|  |        at91sam7a2_timers.c at91sam7a2_usart.c ssc.c
|  +--boot: crt0_gnu.S main.c
|  +--usr:  acc.c acc_data.c debug_ports.c log.c main.c messageIDs.c
|  |        processOutputs.c rt_nonfinite.c processInputs.c
|  +--util: util.c
+--makefile
```

The two folders called `base` contain low level drivers for analog digital converter, CAN, delay routines, exception handling, multi media card, filesystem (FAT16), PWM module, interrupt handling, general purpose input/output, timers, and USART. The folder `boot` contains the necessary init code for starting up the ARM microcontroller (memory controller, heap, stack etc.) to be able to execute C code. Afterwards, the `main` function is called by this boot code. The files that were generated by Real-Time Workshop® Embedded Coder™ (see list on page C) end up in the directory `usr` and additional files are created by *SimulinkTarget* in this location:

- `processInputs.c` handles incoming CAN messages and evaluates them for loading them into the input data structure.

- `processOutputs.c` is the opposite: It periodically reads the values from the output structure and sends the appropriate CAN telegram.

- `debug_ports.c` and `log.c` can be used for debugging output ports to a file on the SD card filesystem.

- `main.c` is the central file that is called on startup (compare listing F.1). Initially, the main function initializes the used hardware: PIO, USART, interrupts, CAN, LED, logging, and last but not least the timer that will signal an interrupt periodically with the interval that was specified in the solver configuration. After everything has been initialized the main loop periodically processes the CAN inputs, calls the Matlab/Simulink® step function, and processes the outputs in an infinite loop.

F. Overview of the Simulink Main Function

Listing F.1: Main function created by SimulinkTarget.

```c
// main function
int main() {
    // initialize controller hardware
    at91sam7a2_pio_init();
    at91sam7a2_usart_init(NULL);
    at91sam7a2_interrupts_init();
    at91sam7a2_timers_config_as_alarm(INTERVAL, &can_send_msg);
    at91sam7a2_timers_config_freerun();
    util_config_led(LED);
    util_clear_led();

    // initialize CAN module 0 (wired on olimex board)
    AT91SAM7A2_CAN_MODULE *can = at91sam7a2_can_init(0);
    at91sam7a2_can_configure_receiver(can,CAN_RECEIVE_CH,0x101,0xffff,8);

    // initialize simulink solver and parameters and register CAN IDs
    simulink_init();
    at91sam7a2_can_register_ids(messageIDs);

    // infinite control loop
    while (1) {
        // can_send_msg flag is set in ISR
        if (can_send_msg) {
            can_send_msg = 0;
            // process the system inputs
            process_inputs();

            // most important: call the step function of the Simulink solver
            simulink_step(0);

            // send resulting CAN telegrams
            process_outputs(can);
}   }   }
```