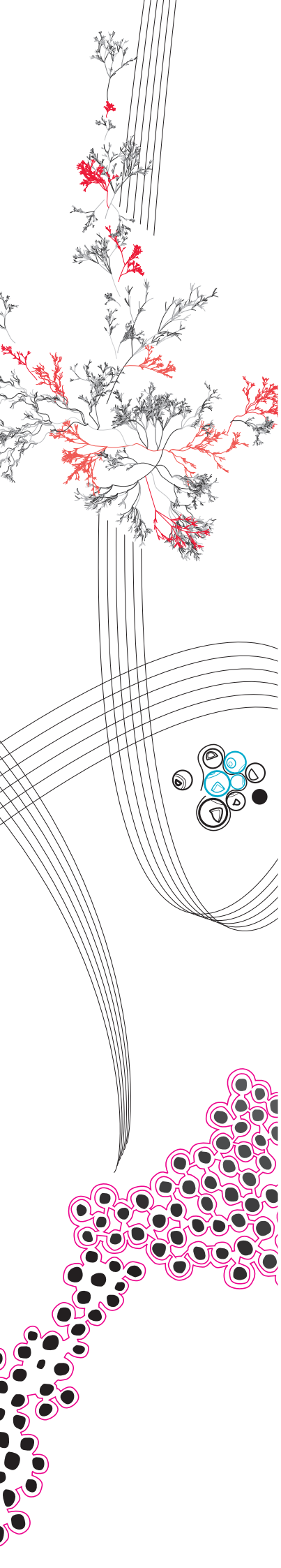


August 18, 2014

MASTER'S THESIS

Hardware design of a cooperative adaptive cruise control system using a functional programming language

T.A.W. (Erwin) Bronkhorst, BSc.

Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)
Chair: Computer Architecture for Embedded Systems (CAES)

Graduation committee:
Dr. ir. J. Kuper
R. Wester, MSc.
Prof.dr. H.J. Zwart
Dr. ir. J. Ploeg

UNIVERSITY OF TWENTE.

Contents

Acronyms **iii**

Abstract **1**

1 Introduction **3**

2 Background **7**
2.1 Cooperative Adaptive Cruise Control 7
2.1.1 Related work 8
2.2 Mathematical model of a CACC system 11
2.3 Hardware design using a functional language 16
2.3.1 Related work 17
2.4 Functional language representation 18
2.4.1 Core model 18
2.4.2 Simulation 21
2.5 Simulation results 22

3 Implementation method **29**
3.1 Modifications to the Haskell code 29
3.1.1 Floating-point to fixed-point 29
3.1.2 Lists to vectors 31
3.2 VHDL simulation 34
3.2.1 VHDL code 34
3.2.2 Simulation results 35

4 Synthesis results **41**
4.1 Synthesis 41
4.2 RTL views 43

5 Conclusion **47**

6 Recommendations **49**
6.1 Modular control systems toolbox 49
6.2 Adjustments in the model 50

6.3 Improvements to CλaSH 50

Bibliography **55**

Appendix A Introduction to Haskell **57**
A.1 General 57
A.2 Basic language components 58
A.2.1 Types 58
A.2.2 Lists 58
A.2.3 Functions 59
A.2.4 Pattern matching 61
A.2.5 Recursion 62
A.2.6 Higher-order functions 63
A.3 Example 63

Appendix B Two-vehicle look-ahead **65**
B.1 Expansion of current model 65
B.2 Two separate input vectors 67

Acronyms

CλaSH CAES Language for Synchronous Hardware

ACC Adaptive Cruise Control

CACC Cooperative Adaptive Cruise Control

CAES Computer Architecture for Embedded Systems

FPGA Field-programmable Gate Array

GHC Glasgow Haskell Compiler

HDL Hardware Description Language

MPC model predictive control

PID Proportional Integral Derivative

RTL Register-transfer Level

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuit

Abstract

In this master's thesis a relatively new hardware design approach is used to model a CACC (Co-operative Adaptive Cruise Control) system. In the traditional hardware design flow, a problem description is made in an imperative language like C and after verification it is rewritten to a hardware description language like VHDL or Verilog. After this step, new verification and simulations are performed and finally, the actual hardware is implemented. In this approach a lot of transformations are involved, which are very time consuming and error-prone.

The new design approach used in this investigation uses a mathematical description given in the functional programming language Haskell to verify the correctness of the model. After that the Haskell code is slightly modified to be compiled into VHDL code using the so called CλaSH (CAES Language for Synchronous Hardware) compiler. This is a compiler developed in the CAES (Computer Architecture for Embedded Systems) group, which translates code that is written in a subset of Haskell into fully synthesisable VHDL code. This enables the possibility to implement the system on an FPGA (Field-programmable Gate Array).

This investigation shows that this design approach is well suited to simulate and model computational systems that control physical behaviour: cyber-physical systems. The Haskell code follows directly from the mathematical description, which makes implementing the problem in a functional language quite easy. Just a few small steps were required to make this Haskell code CλaSH compliant. There was no transformation to another language required, the whole description remained in Haskell. Due to the fully automatic compilation steps, the conversion to VHDL (VHSIC Hardware Description Language) was performed very quick and new simulations showed that no new errors were introduced in this step.

Chapter 1

Introduction

The evolution in technology causes an increasing demand of energy-efficient and at the same time powerful hardware for the calculations in several embedded systems. This demand can not always be satisfied by doing most calculations on general purpose hardware like an ordinary desktop computer, for which the algorithms are written in software. In order to satisfy these demands, it is possible to design hardware which is developed especially for the embedded tasks and which is dedicated to do these calculations. With the increasing complexity of these embedded systems, it becomes harder to go through the design and verification process for the hardware design of these systems. The complexity makes the steps that must be taken in this process very time-consuming and errors can easily arise. In both hardware and software design, a programming language is used to describe the system that is to be designed. While there are a lot of similarities in design approach, there are also some big differences. In hardware design there are some constraints that should be taken into account and which are not common to purely software design, like set up and hold times, input and output buffers and many timing constraints. Therefore there are programming languages developed which focus purely on the design of hardware by giving the designer more control over these things.

A major cause for the challenges in hardware design, is that the traditional hardware description languages, like VHDL and Verilog, lack a high level of abstraction. Unfortunately a high level of abstraction could make the design a lot easier, because the designer can focus on the behavioural part of the system and a lot less on the actual implementation in hardware. A promising method to deal with these problems and to introduce more abstraction to the design flow, is to use a functional programming language. One of the main properties of a functional programming language is that is a high-level language with a high level of abstraction, which is beneficial for productivity. Using this kind of language, the implementation in the programming language stays close to the actual problem description and system behaviour. Another major advantage of using a functional programming language for hardware design, is that there is a direct relation between the functional description of an algorithm and the hardware that implements this algorithm. This relation is not that strong when using an imperative language to describe the system. This is mainly caused by syntactic constructs that are hard to analyse mathematically, like for-loops and assignment-statements.

In a traditional hardware design flow, a model is often made in an imperative programming

language like C. The model is than simulated and verified in for example MATLAB or Simulink. After these steps, the actual hardware design is performed in a hardware description language like VHDL and verified again. With a hardware design approach using a functional programming language, the transition from the functional language to a hardware description language can be more intuitive.

In the CAES group at the University of Twente, a tool is developed that generates fully synthesisable VHDL code from a specification written in the functional programming language Haskell using a subset of this language. This tool is called the CλaSH-compiler. This tool gives the opportunity to describe a system at a high abstraction level by describing its mathematical properties using functional expressions. By describing a problem description in a functional programming language, the simulations on the system can be done without any transformations to a programming language with other semantics. The direct transformation from the description in Haskell to a hardware description in VHDL also eliminates the need for a manual conversion step which can introduce errors in the system.

Since functional programming languages are well suited to describe systems purely with their mathematical equations, this method looks promising to describe a cyber-physical system. These are systems with computational units that control physical behaviour, for example by using sensor information and controlling actuators. The support for higher-order functions in a functional language enables some extra possibilities. One could think of different step sizes for integration methods, depending on the requirements for that specific calculation. In the traditional way of simulating these systems, one global step size is defined. If at one place a high precision in terms of timing is required, than the whole simulation will get that precision. This will lead to long simulation times, or it will require very powerful hardware to do the simulations.

In this research, a simulator of a cyber-physical system will be implemented in Haskell and the result will be compiled using the CλaSH-compiler to investigate if this design method is suitable to describe a cyber-physical system. This leads to the following research question:

- *To what extend is a hardware design method using a functional programming language suitable for the simulation and implementation of cyber-physical systems?*

While looking for the answer to this question, we will investigate how the system can be extended, for example to increase the accuracy and make it more realistic by adding extra parameters. The easier it is to make these modifications, the better the hardware description can be maintained in production environments. This will save time and costs during further development and improvement of a system. Furthermore, there will be investigated if the use of higher-order functions can indeed optimize the simulations so less processing power is required to get the same or a better precision.

The cyber-physical system that will be used in this investigation is a CACC system, an expansion to ACC (Adaptive Cruise Control). With ACC, a vehicle is controlled to keep a desired distance to its predecessor using a controller with the distance and speed as input variables. The expansion CACC also uses the desired acceleration of the preceding vehicle as input variable. This acceleration is received using wireless communication. When using this technology there is more knowledge of the behaviour of a preceding car, enabling the ability to anticipate on the

acceleration or break strength of the predecessor. By using this information as a feed forward in the controller, the following distance can be made shorter without losing string stability [1]. This increases the vehicle capacity of the road. In this investigation, a simulator of a CACC system will be developed to verify the behaviour of the controller.

The developed the simulator consists of two major parts: the vehicle model and the controller. The car model consists of a state variable, which includes the position, speed, acceleration and jerk of the vehicle. The controller uses this state and the state of its predecessor as inputs and calculates a desired acceleration to minimize the distance, speed and acceleration error. When the vehicle is controlled to accelerate at this desired value, the state of the current vehicle is changed, resulting in changing inputs of the controller. When a correct controller is used, an equilibrium is reached in which the errors are (almost) zero.

The structure of this report is as follows. First, a brief introduction on CACC will be given, followed by a full mathematical analysis of a CACC system. When this mathematical background is given, an explanation of the process of designing hardware using a functional programming language is discussed. After that, a description of the implementation is Haskell will be given, ending with the results of the simulation in Haskell.

When the simulations in Haskell are discussed, the conversion to CλaSH compatible code is described. When the conversion is performed, the generated VHDL code is simulated. Finally, the VHDL code is synthesized and the results of this synthesis will be discussed.

Chapter 2

Background

In this chapter, the required background information on CACC is given, followed by the mathematical analysis of the CACC system that will be used. Next, the process of hardware design using a functional language is discussed. After that, an implementation of the CACC system in Haskell is presented.

2.1 Cooperative Adaptive Cruise Control

A big issue on busy roads, are traffic jams. There are campaigns that ask people not to travel during peak hours and to travel together in one car, but they don't have the desired effect. This raises the demand for a technical solution. If the amount of cars can not be reduced, the vehicle throughput of the road must be increased to solve the traffic jams. This can be done by reducing the inter-vehicle distance, but this will immediately lead to unsafe situations caused by the slow response time of human beings. To overcome this slow response time, a technical solution can be introduced, which controls the throttle and brake of the car. An implementation of such a system which is used more and more these days, is ACC. The goal of such a system is to keep a constant distance to its predecessor, or to keep a constant speed if the constant distance could only be achieved if the maximum speed must be exceeded. Using this technique, it is possible to create a vehicle string with all cars driving safely in the platoon while being comfortable for the passenger in the car [2].

ACC systems are developed to increase the comfort of the driver. There is less interference of the driver with the system required, due to the 'adaptive' component in the cruise control. But for a cruise control system that increases the throughput of the road, a shorter following distance must be applied while maintaining safety. This can be achieved if the ACC system is expanded to a CACC [3]. The difference between these systems, is that CACC uses inter-vehicle communication to transmit the current desired acceleration of a vehicle directly to its follower using wireless communication. When this information is used as an extra feed forward signal in the controller of the ACC system, the inter-vehicle distance can be reduced even more [4].

The main challenge of CACC controller design, is to minimize the inter-vehicle distance. There are roughly two ways to look into this distance: in terms of the physical distance (in meters) or in terms of time distance (in seconds). This second method is proven to be a better

spacing policy and it is similar to human behaviour. The time distance between two vehicles is called *headway* and it causes a longer physical inter-vehicle distance at higher speeds. This longer distance can be used as a "buffer" between the vehicles to prevent collisions in case of sudden and fast speed changes of the preceding vehicle. Most spacing policies use a combination of physical distance and time headway. The desired physical distance is than called *standstill distance* and this is added to the time headway to calculate the desired distance while driving at a certain speed.

Keeping the right distance to its predecessor is the main objective of the CACC controller and it is often referred to as *individual vehicle stability*. Besides this one, there is a second control objective: maintaining *string stability*. For a vehicle string to be string stable, there must be a guarantee that fluctuations in the speed of a car are attenuated upstream. This means that following cars should have fewer fluctuations than the preceding car in terms of the signal norm of for example speed [5] [6]. The string stability of a vehicle platoon is very important to prevent the generation of phantom traffic jams. These jams occur when a car breaks too strong, which causes its follower to break even harder to prevent a collision. This behaviour propagates backwards through the traffic, where finally cars will stand still. If a vehicle platoon is string stable, the inter-vehicle time headway is chosen in such a way, that a following car may always break less hard than its predecessor without colliding.

When designing a CACC system, there are two approaches. The first one is designing a centralized system which tells each individual vehicle how to behave. This kind of system gathers the relevant data of all vehicles in a certain area and calculates the best possible behaviour. Such a system requires a lot of information exchange between vehicles and the central system. Therefore, another approach is used which makes use of a decentralized system, where each vehicle gathers the relevant data by itself and calculates its own desired output.

2.1.1 Related work

There is a lot of active research in the subject of CACC systems. When looking into the existing literature in the subject of ACC and CACC, most studies focus on controller design. Some investigations focus on the used control method or algorithm to get better results in terms of stability, others focus on better comfort for the user of the system. A lot of investigations analyse the cruise control systems when several non-idealities are added, like packet loss in the communication. Others try to improve the model by using multi-vehicle look-ahead. String stability and individual vehicle stability is very important in all these investigations.

In 1999, Liang and Peng published a two-level ACC synthesis method [7]. Although the research on string stability started in the late seventies, this publication was one of the first that introduced a synthesis method on string stability. Before that time, design was mostly based on trial and error methods. With this synthesis method, they succeeded in simulating a string stable ACC model with a time headway of 1.4 seconds. The work of this research resulted in the development of a framework for string-stability analysis, which makes it easier to simulate the stability of vehicle strings in MATLAB [8]. Using the simulator, the effectiveness of ACC systems on traffic smoothness was demonstrated. The simulations showed that string stability can be achieved when 25% of the vehicles is equipped with an ACC system set up at the previous used time headway of 1.4 seconds.

Communication delay and packet loss

A very important part of a CACC system, is dealing with delays. A CACC system uses sensors which introduce delays to the system [9]. Furthermore, wireless communication will introduce two negative factors to the system. First, there is a communication delay between the transmission of the signal and the reception of this signal. In 2001, Liu et al. investigated the effects of communication delays on the string stability of a vehicle platoon [10]. The delays in wireless communication are caused by the calculations of the signal processing units and packet losses during the transmission. These packet losses result in absence of the data or delays due to retransmission, depending on the used techniques. The delays in the system are random, which makes it hard to deal with them in a convenient way. Especially in platoons where a vehicle receives information from the platoon leader and its predecessor, these delays have big influence on the string stability of the platoon. Liu et al. propose a solution using synchronization of the information exchange. Using this method, each vehicle in the string uses the received information simultaneously. This makes the delays effectively constant rather than random. Unfortunately, this will require all vehicles to use the same delay which will be larger than the actual delay.

In 2013, Ploeg et al. investigated the packet losses introduced by the communication [11]. Normally, a CACC system degrades to an ACC system in case of packet loss. This will require the time headway to increase but the result will stay string stable. Ploeg et al. present an alternative fall-back method in which the desired acceleration of the preceding vehicle is estimated and used as an input parameter for the CACC controller. Experimental results showed that using this method, the time headway could be kept smaller in case of packet loss, compared to the method where degradation to an ACC system is used.

Two-vehicle look-ahead

Most systems use one-vehicle look-ahead, where only information of the direct predecessor is used. However, there exists research on two-vehicle look-ahead or a system which uses information of the platoon leader to further increase the performance of the system.

In 2004, Sudin and Cook investigated the effect of two-vehicle look-ahead on the string stability and following distance in a platoon [12]. One thing that must be noted, is that this investigation is based on ACC and no information about the desired acceleration of a predecessor is used. In the investigation, two control laws are considered and their results are compared. In the first control law that was used, four errors are defined: the speed and position error to the two directly preceding vehicles in a platoon. The proportional control law minimizes these errors by multiplying each of them with a K factor. The values of this K factor are chosen in such way, that the behaviour of the system is best. The second control law that was investigated introduces an integral term to the defined errors. This resulted in a more smooth response of the vehicles at the same following distance with smaller errors during the experiments. The positive effect of the integral term is even larger in the cases where disturbances like spontaneously breaking vehicles were added to the experiment. However, in both cases the vehicle string was stable and the system avoided collision when a disturbance was added. Unfortunately, there is no comparison made with the performance of a one-vehicle look-ahead system, so no quantitative statement of the effect of two-vehicle look-ahead on the behaviour of the vehicle platoon can be

given.

In the same year, Hallouzi et al. also published a paper which discussed two-vehicle look-ahead in cruise control systems [13]. In this investigation, another control law was used which defines two separate control goals. The fist one tries to minimize the position error to its direct predecessor, the other goal was to minimize the position error to the predecessor of its predecessor. Both separate controllers give a desired acceleration $a_{ref,2}$ and $a_{ref,1}$ respectively as output and based on these two desired acceleration the actual desired acceleration a_{ref} for the vehicle is calculated. If one or both predecessors are breaking, the desired acceleration is defined equal to the hardest breaking vehicle. If none of the vehicles is breaking, the desired acceleration is defined as the mean of $a_{ref,2}$ and $a_{ref,1}$, or the desired acceleration $a_{ref,2}$ calculated from its direct predecessor if that desired acceleration is larger. Some extra intelligence is implemented to enable a smooth transition from one situation to another. The control method is verified with two experiments with three vehicles. In both experiments, the leading vehicle drove three different speeds for a while. The difference between the two experiments was that in one experiment one of the speeds was 0, so the leading vehicle did a stop and go manoeuvre. The results of the experiments were compared to the behaviour of real human drivers and it turned out that the automatic control with two-vehicle look-ahead performed better in terms of reaction time and passenger comfort. Also in this investigation, no comparison with one-vehicle look-ahead control algorithms is made.

A quantitative comparison between two-vehicle and one-vehicle look-ahead is performed in the master's thesis of Kreuzen in 2012 [14]. In this investigation, two different control techniques are used: PID (Proportional Integral Derivative) and MPC (model predictive control)[1]. In this investigation, a two-vehicle look-ahead method using MPC was compared to MPC with one-vehicle look-ahead and with PID. During the investigation it turned out that PID had a smoother and more comfortable response due to less acceleration and jerk, but during one of the experiments it caused a collision. Therefore, the investigation mainly discusses on MPC and the influence of the performance and behaviour of the system using two-vehicle look-ahead with that control method. In this investigation, there are two different look-ahead techniques mentioned. The first one takes information of the two direct predecessors into account, the other one uses information of the direct predecessor and the platoon leader. The conclusions of the investigation teach that the system performs better when the information of two direct predecessors is used, because the behaviour of the second predecessor can be seen as information from the near future. For PID, two-vehicle look-ahead also had a positive influence. The best results were achieved by using information from the platoon leader and the direct predecessor. What must be noted here, is that the collision with the PID controller was of huge influence by deciding which option is better. The overall response using PID was smoother and therefore more comfortable than the response when using MPC, at the cost of response time. Due to the fact that PID responded slower, a collision occurred. In real life, safety is found more important than comfort, so MPC was labelled as the best option.

In a very recent publication of Ploeg et al., there is a two-vehicle look-ahead control strategy

[1]MPC is based on the prediction of the effect on a system when the inputs of that system are changed. For this prediction, a mathematical model to be controlled is used and, if they are available, information about the future values of reference signals.[14]

mentioned [15]. The main focus in this paper is directed to a design method in which string stability requirements can be explicitly included. The one- and two-vehicle look-ahead analysis is used to illustrate the potential of this design method. Simulations of both variants show that two-vehicle look-ahead only performs better than one-vehicle look-ahead in terms of minimal time headway if the communication delay is large. For short communication delays (which barely increases with the physical distance), the minimal time headway was even larger than without two-vehicle look-ahead. The exact cause of this behaviour is not clear, but it seems to be related to the small error in information at low communication delays. It is also plausible that not the optimal control parameters were used. The used parameters were a result of optimisation and is not proven that this was the best solution for this problem. The authors recommend to actively switch between one- and two-vehicle look-ahead communication in a system, depending on the latency in the wireless communication at that specific moment.

Special traffic situations

In practice, there are many more situations than only driving in a single string of vehicles. While most researchers focus on pure platooning of vehicles in one string, there is some research going on that looks into special but very common situations in actual traffic. These special cases are not taken into account in this master's thesis, but it is important to keep these situations in mind. Therefore, some literature on this subject is discussed here.

In 1999, Darbha and Rajagopal investigated the effect of vehicles that want to enter a vehicle platoon, for example at an access road. Before a vehicle could enter a platoon the platoon should increase the inter-vehicle distance at some place, so the extra vehicle can insert the platoon there. Several simulations were performed where vehicles enter and leave the vehicle platoon and the researchers were unable to find a constant time headway spacing policy which leads to a string stable platoon. Some recommendations have been proposed for future work, which makes use of a different spacing policy.

Another example of such a situation is the crossing of two vehicle platoons, which is investigated by Diab et al. in 2012 [17]. An intersection management system is developed in this investigation, which takes over the control of the leading vehicle in a platoon at a crossing. While there was already some research done on intersections of automated platoons, this investigation claims to be the first that treats each platoon as a whole, opposing to older systems which try to control each vehicle individually. By using the control mechanisms of the platoons itself, this system makes sure that the intersection is passed as efficient as possible, without causing accidents or deadlocks. The researchers managed to get a working model and verified its behaviour by using a scale model of actual vehicles.

2.2 Mathematical model of a CACC system

The first step of this investigation is to use a mathematical model for the CACC system. The developed CACC system will use a constant time-headway spacing policy. For this investigation, the model of Ploeg et al. [18] will be used.

Given a string of m vehicles, the desired distance $d_{r,i}$ of a vehicle to its predecessor can be defined as

$$d_{r,i}(t) = r_i + hv_i(t), \quad 2 \leq i \leq m \tag{2.1}$$

In this equation, r_i is the standstill distance to the preceding car and v_i is the velocity of vehicle i. The h is the time headway, which defines a speed dependent following distance. All used variables used in this section, are also presented in Table 2.1.

For the simulation later in this research, the distance d_i between two cars has to be expressed in terms of their absolute positions. The absolute position s_i of a car is defined as the position of the back of the car. With these definitions, the spacing error $e_i(t)$ can be defined as

$$\begin{aligned} e_i(t) &= d_i(t) - d_{r,i}(t) \\ &= (s_{i-1}(t) - s_i(t) - L_i) - (r_i + hv_i(t)) \end{aligned} \tag{2.2}$$

In this equation, L_i is the length of vehicle i.

According to this equation, the car is too far from its predecessor if the spacing error is positive. A schematic overview of the used distances and positions are shown in Figure 2.1.

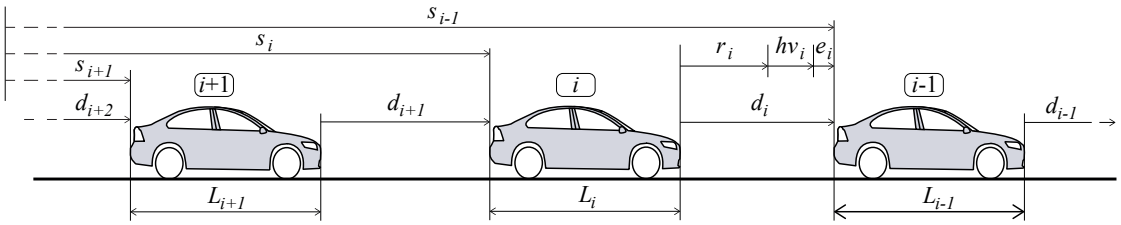


Figure 2.1: Schematic overview of a string of vehicles [19]

For the vehicle model, three quantities are important: the distance d_i to the predecessor, the speed v_i of the vehicle and the acceleration a_i of the vehicle. These quantities will be the basis for the controller design. In the state space representation of a dynamic system, the time derivative of the state of the system at time t is expressed in terms of the state at time t and a control input [20]. To get the state space model of this CACC system, the time derivatives of the previous mentioned quantities should be calculated, resulting in

$$\begin{pmatrix} \dot{d}_i \\ \dot{v}_i \\ \dot{a}_i \end{pmatrix} = \begin{pmatrix} v_{i-1} - v_i \\ a_i \\ -\frac{1}{\tau}a_i + \frac{1}{\tau}u_i \end{pmatrix}, \quad 2 \leq i \leq m \tag{2.3}$$

In this equation, τ is a time constant representing engine dynamics and u_i is the desired acceleration. This is an input for the system: the car that is controlled.

The control law can be designed by formulating the error dynamics. There are three error states $e_{n,i}$ defined, where n is the index of the error state and i is the index of the car, leading to

the definition:

$$\begin{pmatrix} e_{1,i} \\ e_{2,i} \\ e_{3,i} \end{pmatrix} = \begin{pmatrix} e_i \\ \dot{e}_i \\ \ddot{e}_i \end{pmatrix} = \begin{pmatrix} s_{i-1} - s_i - L_i - r_i - hv_i \\ v_{i-1} - v_i - ha_i \\ a_{i-1} - a_i - h\dot{a}_i \end{pmatrix}, \quad 2 \leq i \leq m \tag{2.4}$$

The second and third error state are respectively the first and second order time derivative of the spacing error function. Therefore, they can be expressed in the speed, acceleration and jerk of vehicle i. The second and third element from the error state vector represent the speed difference and acceleration difference between the car and its predecessor.

To construct a state space matrix, the time derivative of each error state $e_{n,i}$ must be expressed in the state variables. From (2.4) it can be extracted that the time derivative $\dot{e}_{1,i}$ equals $e_{2,i}$ and the time derivative $\dot{e}_{2,i}$ equals $e_{3,i}$. The time derivative $\dot{e}_{3,i}$ can be calculated by substituting $\dot{a}_i$ from (2.3) into the expression of the error state. The derivatives of the different error states are

$$\dot{e}_{1,i} = \dot{e}_i = e_{2,i} \tag{2.5}$$

,

$$\dot{e}_{2,i} = \ddot{e}_i = e_{3,i} \tag{2.6}$$

and

$$\begin{aligned} \dot{e}_{3,i} &= \dddot{e}_i \\ &= (\dddot{s}_{i-1} - \dddot{s}_i) - (h\dddot{v}_i) \\ &= (\ddot{a}_{i-1} - \ddot{a}_i) - h\dddot{a}_i \\ &= ((-\frac{1}{\tau}\dot{a}_{i-1} + \frac{1}{\tau}\dot{u}_{i-1}) - (-\frac{1}{\tau}\dot{a}_i + \frac{1}{\tau}\dot{u}_i)) - (h(-\frac{1}{\tau}\ddot{a}_i + \frac{1}{\tau}\ddot{u}_i)) \\ &= -\frac{1}{\tau}(\dot{a}_{i-1} - \dot{a}_i + h\ddot{a}_i) - \frac{1}{\tau}(h\ddot{u}_i + \dot{u}_i) + \frac{1}{\tau}\dot{u}_{i-1} \\ &= -\frac{1}{\tau}\dot{e}_{3,i} - \frac{1}{\tau}\dot{q}_i + \frac{1}{\tau}\dot{u}_{i-1} \end{aligned} \tag{2.7}$$

In (2.7), a new variable q_i is defined. This is the input of the actual controller and is defined as

$$q_i \triangleq h\dot{u}_i + u_i \tag{2.8}$$

This input q_i should stabilize the error dynamics of all states and compensate for the fluctuations in u_{i-1}: the input of the preceding vehicle. While each error state has to reach the value 0 eventually, the control law for q_i is

$$q_i = K \begin{pmatrix} e_{1,i} \\ e_{2,i} \\ e_{3,i} \end{pmatrix} + u_{i-1}, \quad 2 \leq i \leq m \tag{2.9}$$

, with feedback gain factor $K = (k_p \quad k_d \quad k_{dd})$. The values for K have a great influence on the dynamic behaviour of the system. This is primary used to get the desired behaviour (follow the preceding vehicle with a constant time distance) and string stability, but also to get comfortable behaviour. By choosing low values of K the amount of jerk is reduced, resulting in a more comfortable ride.

When substituting the input q_i in (2.7), the expression for the third error state can be constructed as

$$\begin{aligned}\dot{e}_{3,i} &= -\frac{1}{\tau}e_{3,i} - \frac{1}{\tau}(k_p e_{1,i} + k_d e_{2,i} + k_{dd} e_{3,i} + u_{i-1}) + \frac{1}{\tau}u_{i-1} \\ &= -\frac{1}{\tau}k_p e_{1,i} - \frac{1}{\tau}k_d e_{2,i} - \frac{1}{\tau}(1 + k_{dd})e_{3,i} - \frac{1}{\tau}u_{i-1} + \frac{1}{\tau}u_{i-1} \\ &= -\frac{k_p}{\tau}e_{1,i} - \frac{k_d}{\tau}e_{2,i} - \frac{1 + k_{dd}}{\tau}e_{3,i} \end{aligned} \tag{2.10}$$

Now, the time derivative of each error state is expressed by means of the error states and a new state variable u_i, which is the desired acceleration of vehicle i. When (2.9) and (2.8) are combined, an expression of the time derivative of u_i can be made in terms of all state variables. This leads to the expression

$$\dot{u}_i = -\frac{1}{h}u_i + \frac{1}{h}(k_p e_{1,i} + k_d e_{2,i} + k_{dd} e_{3,i}) + \frac{1}{h}u_{i-1} \tag{2.11}$$

With all state variables known and the expressions for their time derivatives, the state space system can be constructed:

$$\begin{pmatrix} \dot{e}_{1,i} \\ \dot{e}_{2,i} \\ \dot{e}_{3,i} \\ \dot{u}_i \end{pmatrix} = \left(\begin{array}{ccc|c} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -\frac{k_p}{\tau} & -\frac{k_d}{\tau} & -\frac{1+k_{dd}}{\tau} & 0 \\ \hline \frac{k_p}{h} & \frac{k_d}{h} & \frac{k_{dd}}{h} & -\frac{1}{h} \end{array}\right) \begin{pmatrix} e_{1,i} \\ e_{2,i} \\ e_{3,i} \\ u_i \end{pmatrix} + \left(\begin{array}{c} 0 \\ 0 \\ 0 \\ \hline \frac{1}{h} \end{array}\right) u_{i-1} \tag{2.12}$$

This system expresses how each state variable changes over time, based on the current state and the input u_{i-1}, which is the desired acceleration of the preceding vehicle. The expression from (2.12) will now be implemented to create the simulator.

In some situations, it is preferred to express the states in terms that have a pure physical meaning. In the state space model of (2.12), the state is expressed in errors of the position, speed and acceleration and there is still a term u_i, which is the desired acceleration and not the actual acceleration. Therefore, the model is changed to contain only the absolute speed, acceleration and jerk. The absolute position is always changing, except when the cars are not moving. Therefore, this state variable will be the distance error. This expression is already shown in (2.2). The time derivative of this expression is shown in (2.3).

The speed is v_i and the acceleration is a_i. The time derivatives of these quantities are respectively a_i and j_i. The latter is the jerk, which will be the last state variable. To calculate the time derivative of this jerk j_i, the expression from (2.3) is used. The expression from (2.8) is used to

substitute $\dot{u}_i$, giving a new expression with variables q_i and u_i. The first one is substituted by (2.9) (with the expressions for the errors given in (2.4)) and the second variable again by (2.3):

$$\begin{aligned}
\dot{\jmath}_i = \ddot{a}_i &= -\frac{1}{\tau}\jmath_i + \frac{1}{\tau}\dot{u}_i \\
&= -\frac{1}{\tau}\jmath_i + \frac{1}{h\tau}(q_i - u_i) \\
&= -\frac{1}{\tau}\jmath_i + \frac{1}{h\tau}(k_p e_{1,i} + k_d e_{2,i} + k_{dd} e_{3,i} - (\tau\dot{a}_i + a_i)) \\
&= -\frac{1}{\tau}\jmath_i - \frac{1}{h\tau}((\tau\dot{a}_i + a_i)) + \\
&\quad \frac{1}{h\tau}(k_p e_i + k_d(v_{i-1} - v_i - h a_i) + k_{dd}(a_{i-1} - a_i - h\dot{a}_i) + (\tau\dot{a}_{i-1} + a_{i-1})) \\
&= -\frac{1}{\tau}\jmath_i - \frac{\tau}{h\tau}\jmath_i + \frac{1}{h\tau}a_i + \\
&\quad \frac{k_p}{h\tau}e_i + \frac{k_d}{h\tau}v_{i-1} - \frac{k_d}{h\tau}v_i - \frac{k_d h}{h\tau}a_i + \frac{k_{dd}}{h\tau}a_{i-1} - \frac{k_{dd}}{h\tau}a_i + \frac{\tau}{h\tau}\jmath_{i-1} + \frac{1}{h\tau}a_{i-1} - \frac{k_{dd}h}{h\tau}\jmath_i \\
&= \frac{k_p}{h\tau}e_i - \frac{k_d}{h\tau}v_i - \frac{1 + k_d h + k_{dd}}{h\tau}a_i - \frac{h + \tau + k_{dd}h}{h\tau}\jmath_i + \frac{k_d}{h\tau}v_{i-1} + \frac{1 + k_{dd}}{h\tau}a_{i-1} + \frac{1}{h}\jmath_{i-1}
\end{aligned} \tag{2.13}$$

With all time derivatives of the state variables known, the state space equation can be represented via a matrix:

$$\begin{pmatrix} \dot{e}_i \\ \dot{v}_i \\ \dot{a}_i \\ \dot{\jmath}_i \end{pmatrix} = \begin{pmatrix} 0 & -1 & -h & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ \frac{k_p}{h\tau} & -\frac{k_d}{h\tau} & -\frac{1+k_d h+k_{dd}}{h\tau} & -\frac{h+\tau+k_{dd}h}{h\tau} \end{pmatrix} \begin{pmatrix} e_i \\ v_i \\ a_i \\ \jmath_i \end{pmatrix} + \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & \frac{k_d}{h\tau} & \frac{1+k_{dd}}{h\tau} & \frac{1}{h} \end{pmatrix} \begin{pmatrix} e_{i-1} \\ v_{i-1} \\ a_{i-1} \\ \jmath_{i-1} \end{pmatrix} \tag{2.14}$$

A final remark that should be made here, is that to be able to do a good analysis on the string stability of the platoon, all state variables should become 0 in the desired state. Therefore, the absolute speed v_i is converted to a speed error v_i'. This speed error is defined as the speed difference with the steady state speed v_{eq}, resulting in the expression

$$v_i' = v_i - v_{eq} \tag{2.15}$$

This change of a variable does not change anything in the values of the state space matrices. This can be shown if the equations of (2.14) are written out, while substituting v_i with $v_i' + v_{eq}$. The negative occurrence of v_{eq} in the v_i part of the equation always cancels out the positive occurrence of v_{eq} in the v_{i-1} part. This holds as long as v_{eq} is equal for all vehicles i, which is the case by definition. When looking into the time derivative of v_i', one will notice that this derivative is the same as the time derivative of v_i as long as v_{eq} is constant. This is important, because the time derivatives are used in the calculations for the second state variable. So under the assumption that v_{eq} is constant and the same for all vehicles, the values in the state space matrices of (2.14) will not change. If v_{eq} is chosen to be 0, the speed error equals the absolute speed of the vehicle, which makes analysis more convenient.

All used variables and quantities used in this section, are summarized in Table 2.1.

Table 2.1: Overview of used variables

Variable	Meaning
a_i	Absolute acceleration of car i
$d_{r,i}$	Desired distance between car i and its predecessor
d_i	Actual distance between car i and its predecessor
e_i	Distance error of car i
$e_{n,i}$	Error state n of car i
h	Time headway (used to calculate speed-dependent distance
$\jmath_i$	Absolute jerk of car i
K	Gain factor of the controller, consisting of parameters k_p,k_d and k_{dd}
L_i	Car length
q_i	Controller input
r_i	Standstill distance between car i and its predecessor
s_i	Absolute position of car i
τ	Time constant representing engine dynamics
u_i	Desired acceleration of car i
v_i	Absolute speed of car i
v_i'	Speed error of car i

Now we have a state space equation with state variables that have a direct physical meaning which can be used in simulations to get a good feeling of the dynamic behaviour of the system.

2.3 Hardware design using a functional language

In the traditional hardware design flow, there are several steps made before the actual hardware is programmed. Most designs start with an algorithmic description, which is written out in a sequential programming language like C or C++. This program is than used to verify the algorithm and system properties with several simulations. When these simulations are performed and the algorithm is considered correct, the program is rewritten in a HDL (Hardware Description Language) like VHDL or Verilog. Unfortunately, this step needs to be performed manually, so new simulations must be performed to verify the correctness. In each step a language with different

semantics is used, so effectively the system description is written twice or even more if domain-specific calculations are required in for example Simulink. This takes a lot of time, because after each translation the code to another language, there are several verifications and simulations that need to be performed to check for new errors that were introduced in the transformation steps.

A more efficient way of designing hardware is enabled with the CλaSH compiler. This compiler is able to generate fully synthesisable VHDL code from a specification written in a subset of Haskell code. A large advantage of this design method, is that the verification and simulation of the designed system can be performed in Haskell. After the transformation to VHDL, the implementation corresponds with the Haskell description and therefore there is no intensive simulation and verification step required. This makes the design process less error prone and the development time much shorter.

Another big advantage from a developer's point of view, is that the hardware description in a functional language like Haskell stays very close to the mathematical description of the algorithm. This makes the hardware description intuitive and relatively easy to debug.

2.3.1 Related work

For already a long time, the idea of designing hardware using a functional programming language exists. In the early eighties, there already were some researchers with the idea of describing hardware in a functional programming language to improve design time and making the process of designing and verification more easy [21]. Some studies already indicated that a functional programming language could be very useful in designing and verifying hardware, for example due to its lazy evaluation and the equational description of systems [22]. Since that time, several projects have started to make hardware design using a functional programming language possible. This means that there are some alternatives to CλaSH, which is used in this investigation. In some of these projects, a completely new language is created. More often, an embedded language is developed which makes use of an existing functional programming language, mostly Haskell. Examples of functional languages that are used to design hardware, are ForSyDe [23], reFL^{ect} [24], Hawk [25], Lava [26] and CλaSH [27]. Since developing a HDL based on a functional programming language is quite a hard and time consuming task and because of the fact that this approach is not yet wide spread, some projects fail to grow out to a solid and full working language. Some other projects evolve into new ones, where the successor deprecates the predecessor. This means that at this moment, not a single "best language" can be nominated. Furthermore, there is a slight difference in application of each language. Some languages are only meant to design and verify hardware, but not to actually implement it. Others provide a way to actually build the hardware, for example CλaSH by compiling the code to synthesisable VHDL code.

One of the more promising projects that grew out to a language what is used on a regular basis, is Lava. Lava is language embedded in Haskell, implemented as a collection of Haskell modules [28]. The language supports polymorphism and higher-order functions [26] and these features can be used to make hardware design more intuitive compared to the classic approach. The written Haskell-code, using the functions and data types from the embedded library, is converted to VHDL after some processing steps. While both Lava and CλaSH, of which the latter will be used in this investigation, convert their own language written in (a subset of) Haskell to

VHDL, there is a fundamental difference between the two. This difference is that CλaSH uses the language Haskell itself to describe the hardware, rather than the embedded language provided by Lava. This enables the use of certain language-specific constructs like case-statements and pattern-matching, which is not possible when using Lava [27].

2.4 Functional language representation

For the developed simulation environment, Haskell is used. In this section, a description is given about how the simulation environment is developed. First, the core simulation is given, which calculates the new error states based on the previous error states. Subsequently the simulator is expanded with some functions, which calculates the actual position, speed and acceleration based on the error states.

2.4.1 Core model

The first step is to define how a car in the vehicle string is represented. The car has to store its own state ($(e_i, v_i', a_i, j_i)^T$) and for programming purposes the index of the car within the vehicle string. For easy analysis of the behaviour of the vehicle, the absolute position and speed of the car are stored. Not all cars have the same length and therefore the length of the car is also stored as a property in the vehicle representation. In Haskell, a new data type `CarType` is constructed by using the record syntax. This type definition is shown in Listing 2.1.

```
data CarType =  Null |
                Car { carid :: Int,
                   position :: Float,
                      speed :: Float,
                  carLength :: Float,
                      state :: [Float]
                 } deriving (Show,Eq)
```

Listing 2.1: The definition of a CarType

Besides the construct `Car`, there is a `Null` without any property. Later on, you will find some functions that require the predecessor of a vehicle as an input parameter. The leading vehicle does not have any predecessor and to deal with that, this Null value is used as the input parameter for the predecessor.

In (2.14), the state change of a vehicle at a certain moment is calculated based on the current state of that vehicle and its predecessor. Define the new state S' at time $t + \Delta t$ as a function of the state S at time t and the state change $\dot{S}$ on the interval $[t, t + \Delta t)$. This state change is a function of the state at time t and the state of the preceding car at time t. In a discrete time system, the time derivative of the state is multiplied by the sample time Δt and added by the current state to get the new state. So if the current state, the sample time and the state change are known, the new state can be calculated using the forward Euler method:

$$S' = \dot{S} \cdot \Delta t + S \tag{2.16}$$

Using this method, the new state of a vehicle can be calculated in Haskell with the function `drive` from Listing 2.2.

```
drive :: CarType -> [Float] -> (CarType,[Float])
drive car predState = (car',state') where
    car' = car {position = pos',
                speed = speed',
                state = state'}
    Car {  state = stateCar,
        position = posCar,
           speed = speedCar} = car

    state' = zipWith (+) stateDiff stateCar where
           stateDiff = map (*dt) dstate
           dstate    = zipWith (+) (mxv a0 stateCar) (mxv a1 predState)

    (_:v':_) = state'
    pos'     = (speed'+speedCar)*0.5*dt + posCar
    speed'   = veq + v'
```

Listing 2.2: Calculation of the new state based on the current state

This `drive` function gets a `Car` and the state of its predecessor as an input, and it returns a tuple of two elements: the new `Car` and the new state of this `Car`. This new state is also present in the returned `Car` itself, but returning it a separate value in the tuple makes it easier in CλaSH to use it as an output.

In the `drive` function, the variables `a0` and `a1` are the matrices from (2.14). This implementation makes use of a function `mxv`, which performs a matrix vector multiplication.

The state of each `Car` consists of four state variables. Two of these variables are the absolute acceleration and jerk, but there are no state variables for the absolute position and absolute speed. In some situations, it is useful to have this information too. As can be seen in Listing 2.1, a `Car` has support for the storage of the absolute position and absolute speed. These values must be calculated separate from the error state. The absolute speed is calculated by adding the steady state speed v_{eq} to the speed error and the absolute position is calculated with time integration of the absolute speed. Due to the discrete time domain, the integral is calculated using the midpoint rule. This is chosen different from the forward Euler method used for the calculation of the new state, because the midpoint rule is more precise than the forward Euler method and all information required for the midpoint rule is known here. All the mentioned calculations are performed in the last three lines of the `drive` function.

To calculate the state change, a matrix-vector multiplication is performed. A matrix vector multiplication is in fact a vector multiplication mapped over all rows in a matrix. This means that each for each row in the matrix the dot product with the given vector is calculated and the result forms the new row, which than consists of one element. In Haskell this is done with the `map` function, which applies the function `.*. ys` to all elements of `xss`. The function `.*. ys` is a dot product with vector `ys`[2]. The matrix `xss` is implemented as a list of rows, so one

[2]If a function requires two arguments and one is given, the result is a function which requires the remaining argument. This is called partial application of the function. This is explained in Appendix A

element of `xss` is one row of the matrix, which is again a list.

To execute a dot product, each nth element of the first vector is multiplied by the nth element of the second vector and the sum of all these products is the result of the dot product. The multiplication is translated to a `zipWith (*)` function in Haskell and the sum of each element translates to a `foldl (+)` function. The resulting implementation of the dot product and matrix vector multiplication is shown in Listing 2.3 [29].

```
mxv :: [[Float]] -> [Float] -> [Float]
mxv xss ys = map (.*. ys) xss

(.*.) :: [Float] -> [Float] -> Float
xs .*. ys = foldl (+) 0 (zipWith (*) xs ys)
```

Listing 2.3: Matrix vector multiplication in Haskell

When a string of vehicles is defined as a list of `Cars`, the string can be simulated for one period Δt with the `drivePlatoon` function, which is shown in Listing 2.4.

```
drivePlatoon :: [CarType] -> (Float,Float) -> ([CarType],[[Float]])
drivePlatoon cars (spdDes,accDes) = (cars',map state cars') where
    cars'     = firstCar' : rest'
    firstCar' = fixFirst.fst $ drive (head cars) [0,
                                                  spdDes - veq,
                                                  accDes,
                                                  0]
    rest'     = map drive' $ zip (tail cars) cars

    drive' (myCar,predState) = fst $ drive myCar (state predState)
    fixFirst car = car {state=(0 : tail (state car))}
```

Listing 2.4: Simulation of a vehicle string for one Δt

The previously mentioned `drive` function requires the current car and the state of its predecessor as an input. Therefore the first `Car` from list of `Cars` is removed with the function `tail` and than `zip`ped with the complete list. The result is a list of tuples which are used as an input for a helper function `drive'`, which calls the function `drive` with the current vehicle and the state of its predecessor and returns the new state of the vehicle (`rest'`). If the `zip` function gets two list of different length (**`tail`** `cars` is one smaller than `cars`), there are tuples created as long as possible. If one of the lists is out of elements, the rest of the elements of the other list are discarded. In this case, the result of the `zip` function is a list of tuples with length equal to **`tail`** `cars`.

By simulating the whole vehicle string, there is a special case for the first vehicle, because it has no predecessor. However, the function `drive` can still be used for this case, when the state variable of its virtual predecessor is constructed. The leading vehicle will use the difference between the steady state speed v_{eq} and the desired speed, which is given as an input for the function `drivePlatoon`, as the value for the second state variable of its (virtual) predecessor. The third state variable is set equal to the desired acceleration, which is also an argument of the

drivePlatoon function. The other state variables are set to 0. By using this state variable as an input for the calculations of the first vehicle, the new state of this vehicle can be calculated. Due to the fact that this is the leading vehicle it can also not have a position error, so the first state variable representing the position error is set to 0 after calculation. With these steps, the new instance of the first Car is placed in front of the new instance of all other Cars (rest').

To make an endless simulation of a vehicle string, this drivePlatoon function is called repeatedly to get a simulation over time. Each time sample is a list of cars and the whole time is a list of these lists of cars. The simulate function that does this, is shown in Listing 2.5. The parameter n indicates the starting value of the time. In most simulations, this will be set to 0. In the developed simulator, there is a function desiredSpeed n and a function desiredAcc n which return respectively the desired speed and acceleration at time n. By changing these functions, the desired speed and acceleration at any time can be defined. In most situations, the function desiredAcc is implemented as a time derivative of the desiredSpeed function, but it is also possible to implement it as the exact function of the time derivative using the analytical time derivative of the input function.

```
simulate :: Float -> [CarType] -> [[CarType]]
simulate n cars = cars' : simulate (n+dt) cars' where
    cars' = fst (drivePlatoon cars (desiredSpeed n,desiredAcc n))
```

Listing 2.5: Simulation of a vehicle string over time

2.4.2 Simulation

To run an actual simulation of a vehicle string, the following steps are taken. At $t = 0$, the vehicle string is created. To do this, the position, speed and acceleration of each vehicle has to be defined. With these values, the error state of each vehicle can be calculated using the information of the preceding vehicle. The state variables of each vehicle in the string are calculated by using (2.4), except for the jerk. This state variable will always be 0 for each vehicle at the start of a simulation. Due to the fact that the first vehicle has no predecessor, the state variables must be calculated in a different way. The position of this first car is the actual position of the whole vehicle string and therefore, the position error of this vehicle will start at 0 and stay 0 forever. The speed error is calculated with respect to the output of the function desiredSpeed at time 0. In equilibrium, each car will drive at a constant speed and therefore the acceleration will be 0. The acceleration error is thus equal to the actual acceleration of the car. The same holds for the jerk error.

A vehicle can be added to the tail of a vehicle string using the addCar function. The parameters of this function are the length of the new vehicle, the vehicle string and the position, speed and acceleration of the new vehicle at time $t = 0$. With this data, a new vehicle string is created and returned. The Haskell code that implements this function is shown in Listing 2.6.

As shown in Listing 2.6, you can see the state variables being calculated based on the values given by the function arguments and by the state of its predecessor. This makes it easier to add a vehicle to a vehicle string, without calculating the initial state by hand.

```
addCar :: (Float,Float,Float) -> Float -> [CarType] -> [CarType]
addCar (pos,spd,acc) myLength cars = cars ++ [car] where
    Car {carid    = carId,
         position = posLast} = last cars

    state' = [e',v',a',j']
    e'     = posLast - pos - myLength - ri - h*spd
    v'     = spd-veq
    a'     = acc
    j'     = 0
    car    = Car {carid     = carId + 1,
                  position  = pos,
                  speed     = spd,
                  carLength = myLength,
                  state     = state'
                 }
```

Listing 2.6: Adding a car to a vehicle string

2.5 Simulation results

To verify the correct behaviour of the controller, the system is simulated in the interactive Haskell interpreter of GHC (Glasgow Haskell Compiler). The simulation parameters are chosen in such a way, that the correctness and behaviour of the model is verified. Four simulations are performed, with increasing complexity. This allows us to analyse the behaviour of the system step by step and remarkable behaviour is spotted faster.

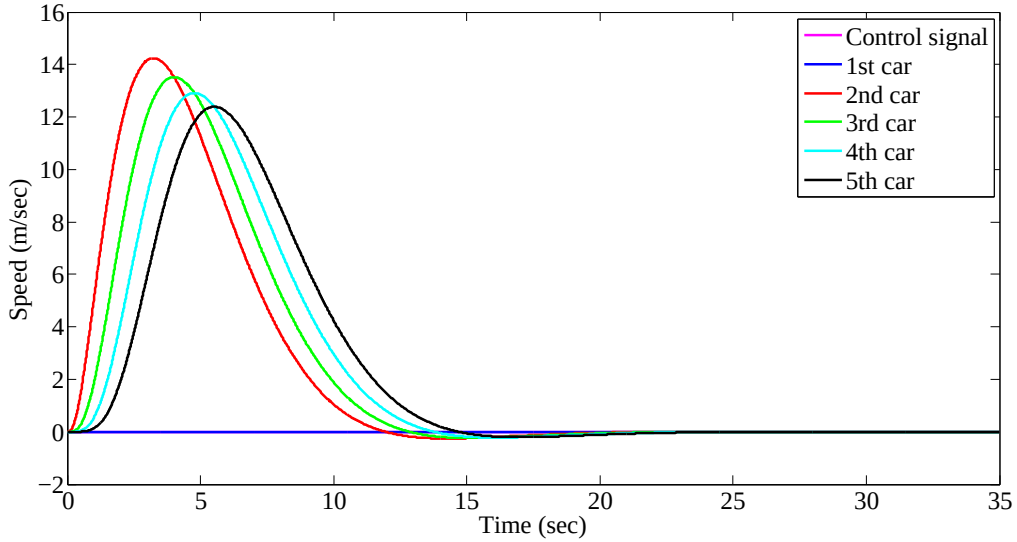
For all simulations, some realistic values for the different constants in the model are used. Most of these values are taken from [18]. The values for the k-constants are chosen to get a realistic speed of response and optimal passenger comfort. The value of h is based on the minimal required time distance for string stability. The values are shown in Table 2.2.

Table 2.2: Overview of used constants in the simulation

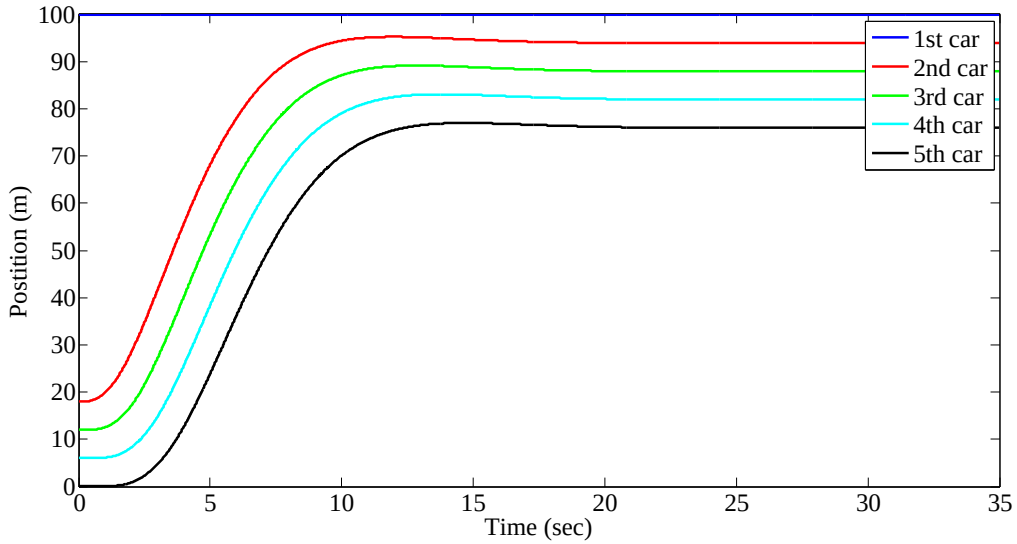
Constant	Value	Unit
r_i	2	m
L_i	4	m
h	0.7	s
τ	0.1	s
k_p	0.2	
k_d	0.7	
k_{dd}	0	
Δt	0.01	s

With these parameters, a simulation is performed with a vehicle string of five cars. All cars start with a speed $v_i = 0$ at time $t = 0$. The starting position of the front car is 100, the starting position of the second car is 18 and each next car is placed 6 meter behind its predecessor (12, 6 and 0). This means that the inter-vehicle distance is 2 meters (equal to the standstill distance), given the vehicle length of 4m. For the first simulation, the control signal with the steady state

speed v_{eq} is kept at 0. This will cause the first vehicle to stand still. The second vehicle is too far behind its predecessor, so it will start driving towards the first vehicle. This will cause all other vehicles to drive as well, until all vehicles stand still at a standstill distance equal to 2 (r_i). The simulation results are shown in Figure 2.2.



(a) Vehicle speeds



(b) Vehicle positions

Figure 2.2: Behaviour of the controlled vehicle string with desired speed 0 as input.

The figures confirm the expected behaviour of the vehicle string: the first vehicle is standing still and the other vehicles are driving towards the leader. There is already a sign of string stability visible in Figure 2.2a. The maximum speed of each vehicle in the vehicle string is

lower than its predecessor, except for the first vehicle of course, because it is standing still.

In Figure 2.2b there is a bit overshoot visible in which the vehicles come too close to their predecessor. This corresponds to in a negative speed: the cars are driving backwards. This seems bad in a real environment, but it is expected that when the cars have a speed larger than 0 at steady state, they will not end up driving in reverse.

To verify the behaviour of the vehicle string in the case where the desired speed is larger than 0, a new simulation is run. In this case, the starting positions remain unchanged and also the starting speeds are kept at 0. However, the function `desiredSpeed` is set to a constant value of 15m/s. The simulation results are shown in Figure 2.3.

As you can see, each vehicle will increase its speed, based on the distance to the preceding car and the speed error compared to the steady state speed v_{eq}. In Figure 2.3b the effect of a time headway as spacing policy is also visible. This can be seen when looking at the distance between the vehicles at the steady state, which is larger than in the steady state of Figure 2.2b, where the speed was 0. It is also visible when looking at $t = 0$ and $t > 20$ in Figure 2.3b, because at $t = 0$ the cars are placed at standstill distance while their speed is 0.

In the previous simulations, the control signal was fixed to one speed. However, the system must be able to change between different desired speeds. To verify the behaviour of the system in these cases, all vehicles are placed at standstill distance of each other, with a starting speed of 0. The last vehicle will have position 0 and each next vehicle will have an absolute position that is 6 more. This results in a starting position of 24 for the leading vehicle. Some different speed steps are used as input, to verify the correctness of the speed dependent following distance. The results of this simulation are shown in Figure 2.4.

With these values, the system still seems to be string stable and the inter-vehicle distance is indeed speed dependent.

With the last simulation that is performed with the written Haskell code, several different cases at once are tested. Therefore, the start conditions of the vehicles in the string are changed. The used start values are shown in Table 2.3.

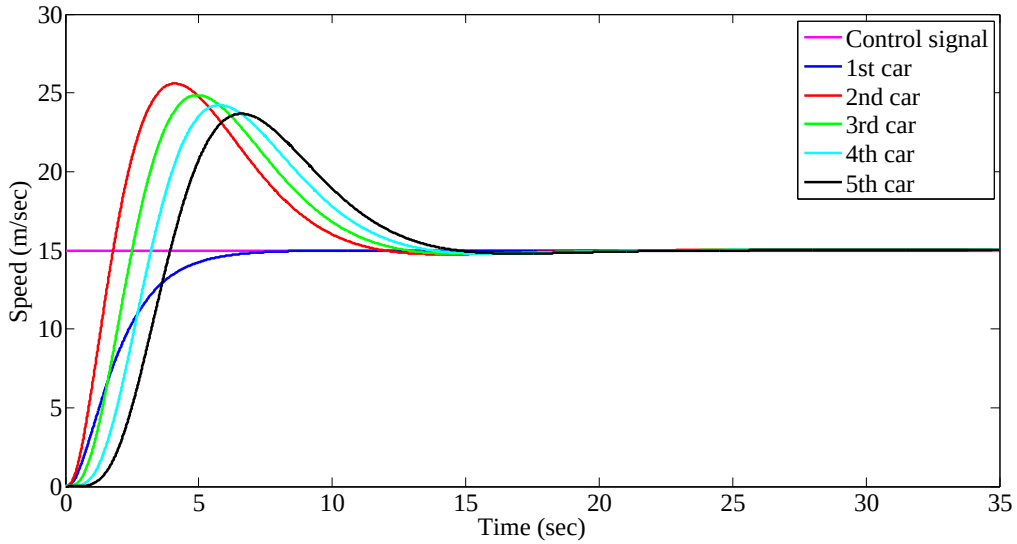
Table 2.3: Start conditions of each vehicle

	Position (m)	Speed (m/s)
Car 1	100	15
Car 2	80	0
Car 3	65	5
Car 4	55	10
Car 5	20	25

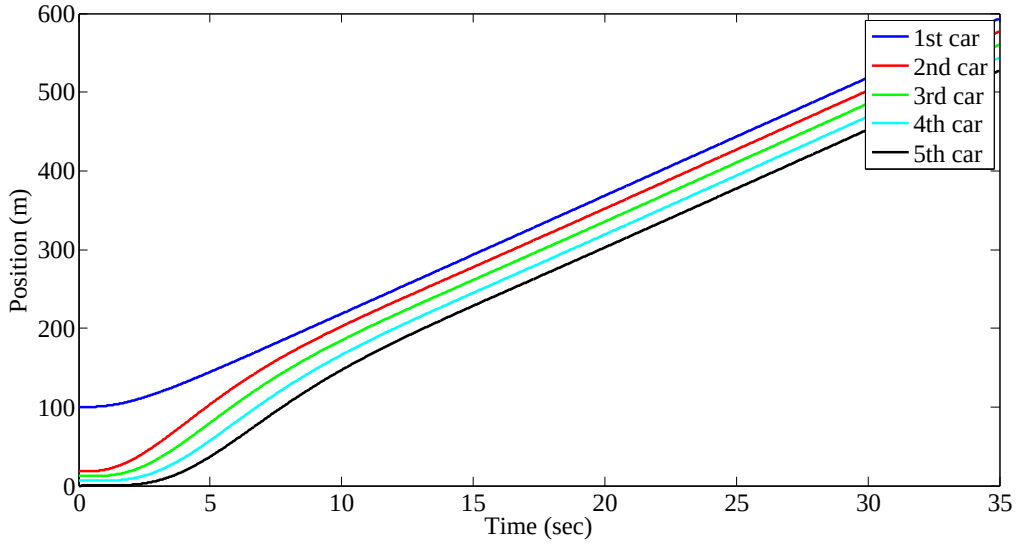
With these starting values, some vehicles need to reduce the distance to their predecessor, some others must increase it.

Furthermore, the control signal dictating the desired speed is changed to the randomly chosen periodic function

$$f(t) = 2(\sin(t) \cdot \cos(1.6t) + 8) - 1 \tag{2.17}$$



(a) Vehicle speeds



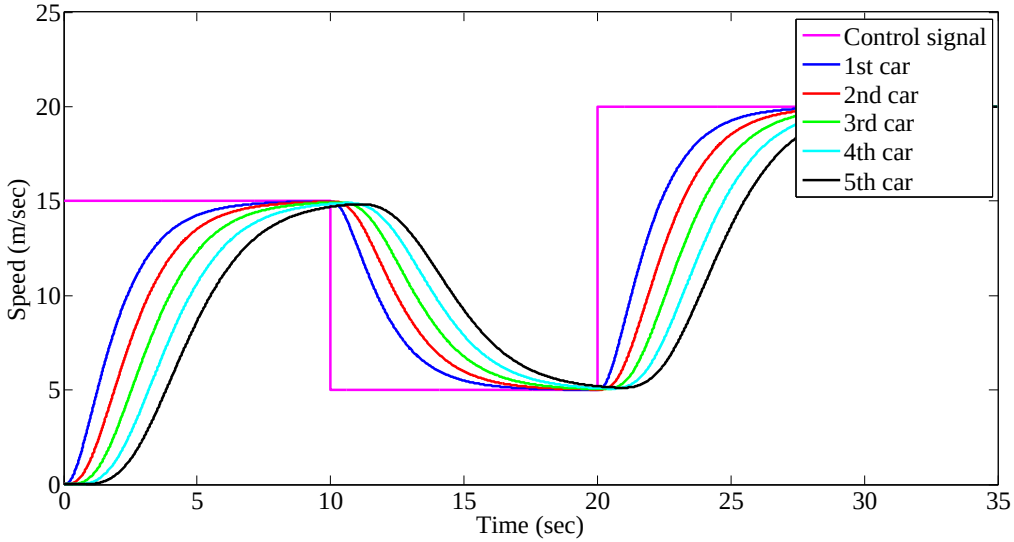
(b) Vehicle positions

Figure 2.3: Behaviour of the controlled vehicle string with desired constant speed as input.

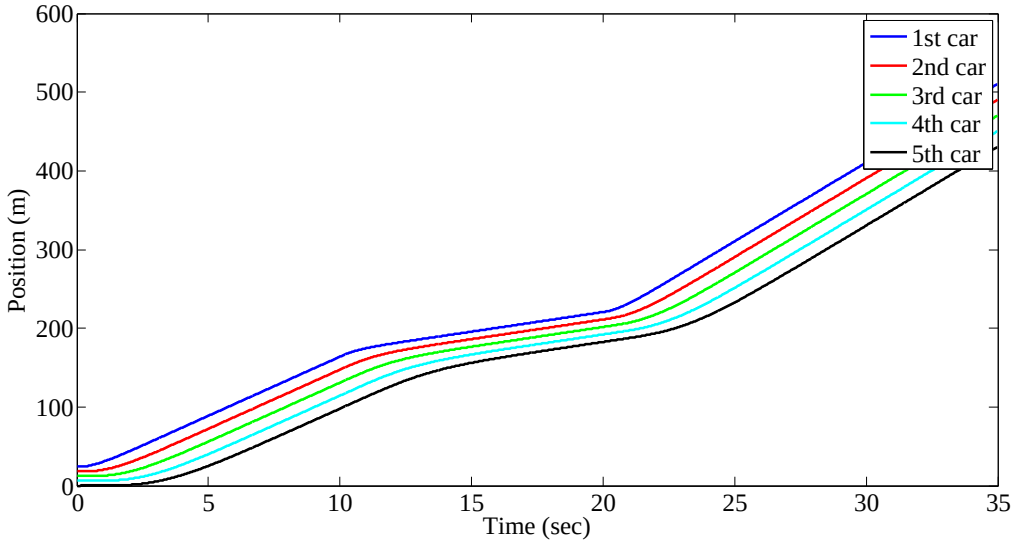
, which consists of a high and a low frequency component. The average of this signal is 15, so the value for v_{eq} stays 15. The simulation results of this experiment are shown in Figure 2.5.

It is visible in the image that each vehicle starts to correct for its incorrect starting values. After about 2.000 time samples (20 seconds) the system is stable and each vehicle follows its predecessor at the correct speed.

One thing that can be noticed, is that Car 3 lowers its speed a little bit at $t = 0$, while its speed is already too low. The reason for this is that the predecessor of this vehicle (Car 2) has



(a) Vehicle speeds

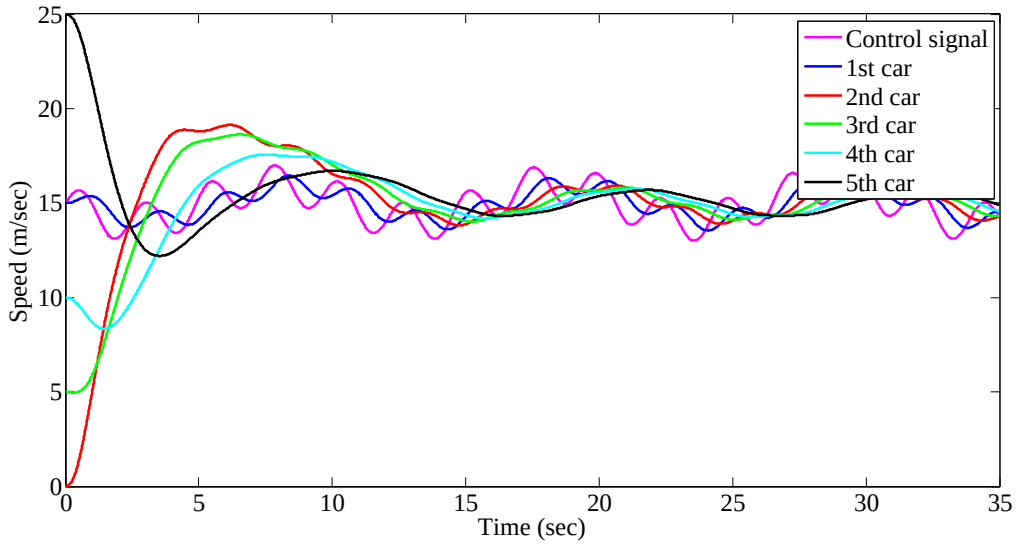


(b) Vehicle positions

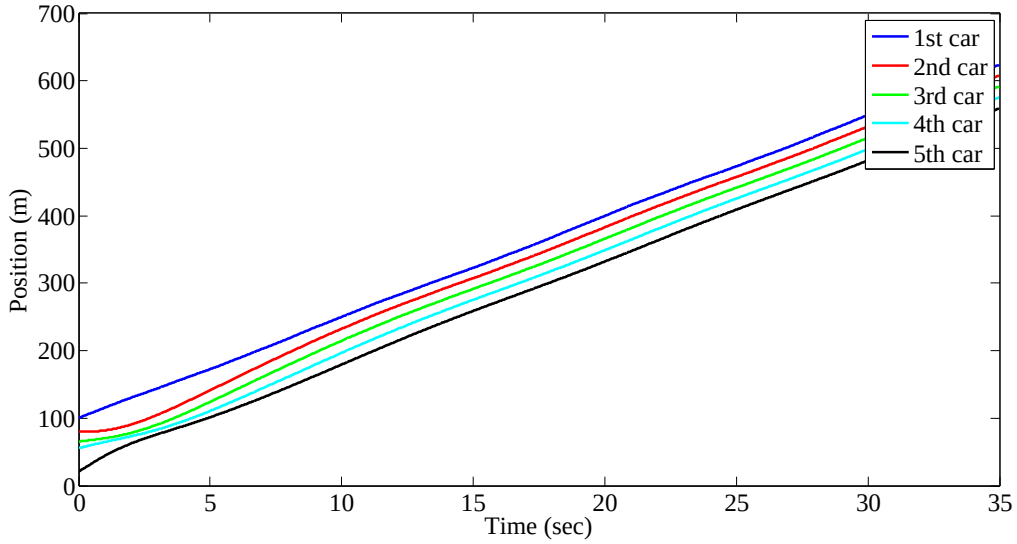
Figure 2.4: Response of the controlled vehicle string with different speed steps.

a lower speed, so Car 3 has to slow down to prevent collision. Also Car 4 slows down a lot and the main reason for that is that its distance to Car 3 is very short. The car slows down to let this distance increase and after a while, the speed is increased to get to the desired speed.

In this simulation, string stability is also present. After the set-up time, each vehicle follows the control speed but with a lower amplitude than its predecessor. This means that fluctuations in the speed are attenuated propagating through the vehicle string, which is a requirement for string stability.



(a) Vehicle speeds



(b) Vehicle positions

Figure 2.5: Response of the controlled vehicle string with periodic function as input.

Chapter 3

Implementation method

The developed functional language representation in Section 2.4 is used as reference for the CλaSH code. However, before the CλaSH compiler is able to generate VHDL from the Haskell code, it should be modified. In the following section, these modifications are discussed. When the CλaSH compatible Haskell code is explained, the compilation to VHDL is discussed, followed by the simulations performed on the VHDL code in QuestaSim. The last part of this chapter focuses on the synthesis of the VHDL code.

3.1 Modifications to the Haskell code

While the CλaSH compiler can compile "normal" Haskell code, some changes must be made before it can be translated to a real hardware description[30]. The reason for this, is that hardware puts some limitations on the design and in some cases the efficiency in terms of FPGA area usage can be increased. In the following subsections, two important changes to the Haskell code are discussed.

3.1.1 Floating-point to fixed-point

In the developed program in Section 2.4, the most used data type is a single precision floating-point number. It is possible to implement floating-point calculations in hardware, but it requires a lot more hardware compared to fixed-point calculations. The design choice for floating-point numbers in the program from Section 2.4 is for the precision of the values. The program uses meters and seconds as units of measurement and the control system requires more precision than that. It is also possible to achieve the required precision by using fixed-point numbers.

For the conversion from floating-point to fixed-point, there are a few things to take care of. The first one, is that there is a choice between signed and unsigned numbers. The state space matrix from (2.14) contains negative numbers. If these numbers are represented as fixed-point numbers, they should be signed. Furthermore, it is very likely that the acceleration of the car will become negative in some situations, it is required to use a signed fixed-point number for these as well.

The second thing to take care of, is that no integer overflow should occur. If too few bits are used for the integer part of the fixed-point number, a large value of the integer part will overflow and become a negative value. This should never happen, because the system will get undesired behaviour which can have a great impact on the safety of the driver. Therefore, the integer part should be large enough to represent the highest possible integer value within the simulator. The same holds for large negative numbers, which might underflow from negative to positive.

The last important design choice is the number of bits used for the fractional part of a fixed-point number. The more bits are used in this part, the more precision can be achieved. When the precision is too low, errors are introduced in the calculations. These errors might influence the calculations later on, because they use the result of the previous calculations. To deal with this effect, it is important to keep the precision at the required level.

For the implementation of the simulator, a single data type for all occurrences of a floating-point number is chosen. This will have implications for the efficiency of the program, because sometimes more precision is required and sometimes a higher value for the maximum integer value. To be able to deal with both situations, the length of the fixed-point number will be higher. However, using one data type makes it easier to do calculations between domains. For example, if the speed is represented in fixed-point number with different dimensions than time, a conversion should be made before the speed can be converted to a distance with integration. To eliminate these conversions, a fixed-point number is used which is large enough for all situations.

In the developed simulator, the position of the car will have the highest value for the integer part of the fixed-point number. A problem is that the absolute position has virtually no limit if the car keeps driving in one direction. While the speed and acceleration have a physical limitation on their value (at some point, a car can not drive or accelerate faster), there is no such limitation for the absolute position. Fortunately, the value for the absolute position is only used to show the behaviour of the car and is not used in the calculations. In these calculations, only the position error is used and this value is for example bounded by the range of the lidar and the range of the wireless transmitter.

The boundaries on the speed, acceleration and jerk of the car are determined by the controller. The control parameters K of the controller are chosen in such a way, that the acceleration and jerk should not exceed the comfort boundaries. In 1977 an investigation is performed on the comfort values for the acceleration and speed of a vehicle [31]. This investigation pointed out that a jerk that exceeds 3 m/s^3 and an acceleration higher than 2 m/s^2 are considered uncomfortable, although sportier drivers will have no problems with larger accelerations. If the controller obeys these boundaries, the integer part must be able to contain these physical quantities. If the controller is allowed to exceed these boundaries, for example in emergency situations, the values will become larger. However, they will never exceed the limitations of the vehicle itself (maximum break and acceleration power). It is expected that when the maximum is considered as twice the comfort boundaries, the actual acceleration and jerk will never exceed this maximum value for the integer part.

Due to the fact that the physical limit for the integer part is very low, the boundary would be the absolute position used in the simulator. In this investigation, there is chosen for a maximum simulation distance of 2km, which translates to 11 bits ($2^{11} = 2048$). Due to the fact that a signed number is used, the integer part of the fixed-point number will consist of 12 bits.

With regard to the fractional part of the fixed-point data one can say that more bits is better. The more bits are used, the more precise the simulation will be. In this investigation, it is chosen to use 32 bits for the fixed-point data type. A word size of 32 bits is very common in computing and in the previous simulations from Section 2.5 a Float, which is 32 bits long, was used. When 12 bits for the integer part are used, there are 20 bits left for the fractional part. It is expected that this will be enough to get a reasonable precision, but this should be verified by running the simulations.

To summarize this analysis, a global data type will be used to represent all values that are floating-point numbers in the regular Haskell code from Section 2.4. A 32 bits unsigned fixed-point number will be used, in which 12 bits are used for the (signed) integer part and 20 bits for the fractional part. For easy modification of this configuration, a new data type `FPnumber` will be defined in Haskell, using the fixed-point data type from the CλaSH-prelude [32]:

```
type FPnumber = SFixed 12 20
```

Listing 3.7: Global fixed-point data type

This data type is used everywhere in the Haskell code, where `Float` is used before. If it points out later in the investigation that the size for the integer or fractional part is wrong, it can easily be changed here.

The correctness of the choice for 12 integer bits and 20 fractional bits is verified by running some simulations with different sizes of fixed-point numbers. The behaviour was clearly incorrect when there were not enough integer bits, because of the overflow. In case of the fractional part, the precision was clearly visible and more bits was always better. There is no clear boundary of what is acceptable or not, but 20 bits turned out to be enough in these simulations.

3.1.2 Lists to vectors

When the simulator is implemented in hardware using CλaSH, it is not possible to use lists. There is no support for lists in CλaSH, because it is very hard to implement them efficient in hardware. Especially in the case of variable length lists, a lot of extra intelligence must be implemented to te able to work with them. There must be hardware reserved to perform calculations on these lists and if the number of elements is not known or not constant, this will require a lot of logics. Therefore, vectors with a fixed length should be used.

In the program from Section 2.4, all lists can be converted to vectors. Fortunately, most used lists already have a fixed length. For example, the list representing the state of a vehicle contains four elements and the calculation matrices from (2.14) contain four rows of four elements. These can immediately be changed to vectors of size 4.

The only list that has no fixed size, is the list of vehicles in the platoon. This raises a problem in, for example, the `addCar` function from Listing 2.6. This function returns a list of cars which is one element longer than the vehicle list at the input. To deal with this problem, a fixed number of vehicles in the platoon is defined. For this investigation, the platoon will consist of five cars resulting of data type `PlatoonVec`, which is a vector with five `Car` type elements. This means

that everywhere in the code, the vehicle platoon should be this vector with five elements.

This is achieved by defining a platoon vector, containing five elements with the value `Null`. Each time the `addCar` function is called, a `Car` is right shifted into the vector. When this is done five times, the platoon vector contains five `Car`s with the first car of the platoon on the first position. The resulting function is shown in Listing 3.8.

```
addCar :: (FPnumber,FPnumber,FPnumber) -> FPnumber -> PlatoonVec -> PlatoonVec
addCar (pos,spd,acc) myLength cars = (cars <<+ car) where
    Car {carid    = carId,
         position = posLast} = vlast cars

    state' = e' :> (v' :> (a' :> (j' :> Nil)))
    e'     = posLast - pos - myLength - ri - h*spd
    v'     = spd-veq
    a'     = acc
    j'     = 0
    car    = Car {carid     = carId + 1,
                  position  = pos,
                  speed     = spd,
                  carLength = myLength,
                  state     = state'
                 }
```

Listing 3.8: Adding a car to a vehicle string in CλaSH

When you compare this function with the function in Listing 2.6, you can see some small differences. The first one is that the data types of the function parameters and return values are different. The other difference is that all lists are now vectors. This holds not only for the input and output (`PlatoonVec`), but also from the state of the car, which is a vector of four `FPnumbers`.

Another function from Section 2.4 that is influenced by the change from lists to vectors, is the `drivePlatoon` function. First of all, it is important to mention that the CλaSH code is built up more modular. Due to the fact that errors in the synthesis of the generated VHDL code are hard to debug, the simulator is build up with two main modules: the module for an individual `Car`, and a module for the whole vehicle platoon. This makes it possible to synthesize and analyse the behaviour of one `Car` first, making debugging a bit easier. After that, the `Car` module can be used within the `Platoon` module. If any errors occur after that, it is most likely that there is an error in the platoon-specific code.

In the CλaSH implementation, there are two functions called `drive`: one in the `Car` module and one in the `Platoon` module. The first one, the `drive` function from Listing 2.2, is converted to the `drive` function in the `Car` module (Listing 3.9).

You can see that also this function uses different data types for its in- and outputs. The `Floats` are now `FPnumbers` and the lists are converted to vectors. This effect of the elimination of lists is visible when the old `drive` function and the CλaSH variant from the `Car` module are compared. The latter version makes use of vector functions. In most cases, these functions are the same as the functions for lists, except that their name starts with the letter *v* (`vmap`, `vzipWith`, `vhead`, `vtail`). The biggest difference is the way the speed error is extracted from the state variables. In the old Haskell code, this is done by pattern matching and the

```
drive :: CarType -> StateVec -> (CarType,StateVec)
drive car predState = (car',state') where
    car' = car {position = pos',
                speed = speed',
                state = state'}
    Car {  state = stateCar,
        position = posCar,
           speed = speedCar} = car

    state' = vzipWith (+) stateDiff stateCar where
           stateDiff = vmap (*dt) dstate
           dstate    = vzipWith (+) (mxv a0 stateCar) (mxv a1 predState)

    v'     = (vhead.vtail $ state')
    pos'   = (speed'+speedCar)*hlf*dt + posCar
    speed' = veq + v'
```

Listing 3.9: `drive` function from the `Car` module

second element of the list is fetched. In the CλaSH implementation, vectors are used and they do not support pattern matching. Therefore, the head of the tail of the vector is taken, which is also the second element. A last remark that should be made here, is that a variable `hlf` is used in the calculation of the new position `pos'`. This is due to the fact that all numbers in this calculation are `FPnumbers` and when `0.5` is written down here, it is not casted to an `FPnumber`. This will result in an error. Therefore, a constant of type `FPnumber` with value 0.5 is defined (Listing 3.10) and that constant is used here.

```
hlf = $$(fLit 0.5) :: FPnumber
```

Listing 3.10: `hlf` constant of type `FPnumber`

The `drivePlatoon` function from the old Haskell code is converted to the `drive` function in the `Platoon` module (Listing 3.11). The `Platoon.drive` function calls the function `Car.drive`. Also here, the `Floats` are now `FPnumbers` and the lists are vectors.

```
drive :: PlatoonVec -> (FPnumber,FPnumber) -> (PlatoonVec,Vec 5 StateVec)
drive cars (spdDes,accDes) = (cars',vmap state cars') where
    cars'     = (firstCar' :> rest')
    firstCar' = fixFirst.fst $ Car.drive (vhead cars) (0 :>
                                                       (spdDes - veq :>
                                                       (accDes :>
                                                       (0 :> Nil))))
    rest'     = vmap drive' $ vzip (vtail cars) (vinit cars)

    drive' (myCar,predState) = fst $ Car.drive myCar (state predState)
    fixFirst car = car {state=(0 :> vtail (state car))}
```

Listing 3.11: `drive` function from the `Platoon` module

As can be seen from this chapter, the conversion from "ordinary" Haskell code to CλaSH code is quite straightforward. If the lists used in the Haskell code have a fixed length, the conversion from lists to vectors is barely more than some changes in the syntax. Only if the lists differ in size, some changes in the design should be made. A note that should be made here, is that the indexation of elements in a vector in CλaSH is different from the indexation of elements in a list. Where in Haskell the leftmost element of a list has index 0, for vectors from the CλaSH library, this is the rightmost element. The reason for this, is that it is more intuitive to give the least significant bit of a bit vector the lowest index number. So if someone uses indexation of elements in his design, he should take care of the right indexing. In some cases this might require some extra modifications in the code before it can be compiled to VHDL using CλaSH. In this investigation, no indexing of elements using the operator `!!` is used, so there are no modifications required for this. If an element is selected, it is done using the `head` and `tail` functions, or a cascade of these functions.

3.2 VHDL simulation

When the discussed changes from Section 3.1 are applied to the Haskell code of the platoon description, the code is converted to VHDL with the CλaSH compiler. The resulting VHDL code is simulated in QuestaSim, a tool that can be used to write and simulate hardware designs which are written in VHDL. In this section, the resulting VHDL code is briefly discussed, following by the simulation results.

3.2.1 VHDL code

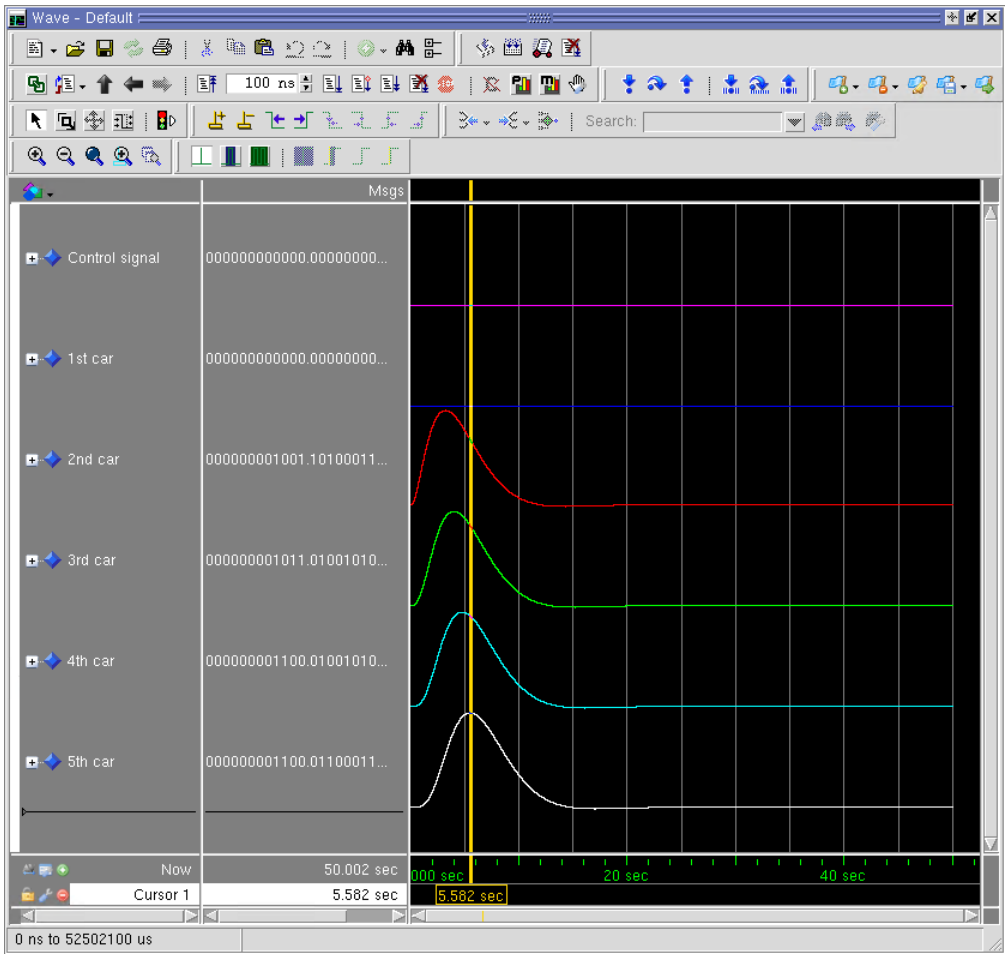
The VHDL code that is generated by CλaSH consists of several files, in this case 37 files. One of the things that can be noticed, is that the functions that are designed in Haskell are now separate VHDL files. For example, there are the files `drive_2.vhdl` and `drive_4.vhdl`, which contain the description of the `drive` functions for the `Car` and the `Platoon` module. Furthermore, a file `mxv_16.vhdl` can be identified, which represents the matrix vector multiplication. Another file that can be recognized immediately, is the file `addCar_27.vhdl`, which is the VHDL representation of the `addCar` function in Haskell.

To run simulations in QuestaSim, a test bench must be written which describes the behaviour of the input signals and the clock of the hardware. CλaSH has generated a VHDL test bench file which can be changed to use your own input signals. To be able to make a good comparison between the behaviour of the VHDL code and the Haskell code, the same simulations as in Section 2.5 are performed on the VHDL code in QuestaSim. This means that also the same parameters from Table 2.2 are used.

One thing that is changed for the simulation, is the time scale. At default, the clock runs at a speed of 1MHz. However, the simulations in Haskell are performed with a clock period of 10ms (because δt is 0.01 seconds). This translates to 100Hz. Therefore, the default clock signal is changed so it is 5ms high and 5ms low. The whole simulation will cover 50 seconds, which is the same as the simulations performed in Haskell.

3.2.2 Simulation results

The first simulation that is performed, is the car that stands still at position 100, with the other cars at position 0, 6, 12 and 18. The result of that simulation is shown in Figure 3.1.



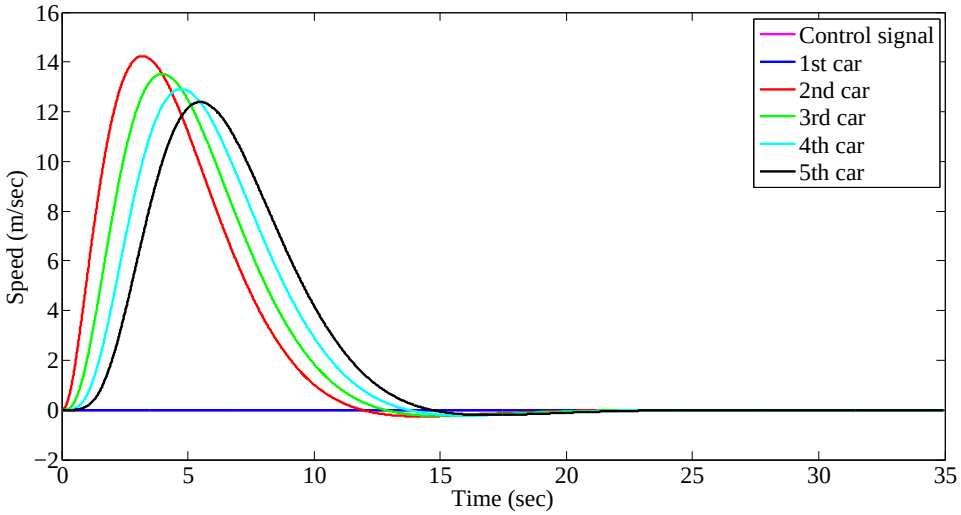
(a) Vehicle speeds

Figure 3.1: Behaviour of the controlled vehicle string with desired speed 0 as input.

When looking into the binary values that are visible in the second grey column, an analysis can be performed on the exact values of the signals. These binary values show the values of the fixed-point output signal of the speed at the location of the yellow line. This line is placed at 5.5 seconds, which is at the maximum speed of vehicle 5. The value of this maximum is visible as 1100.01100011. This corresponds to a decimal value of around 12.39. This corresponds with the maximum value of vehicle 5 in Figure 2.2a. The same analysis can be performed for the other speed graphs and they all correspond to the simulations in Haskell. This is an indication that the VHDL code behaves the same as the Haskell code.

Unfortunately, it is not possible in QuestaSim to show the output signals in the same graph,

so they have to be placed beneath each other. Therefore, the data from QuestaSim is exported and this data is used to create the same chart layout as Figure 2.2. The graph is shown in Figure 3.2a.



(a) Vehicle speeds

Figure 3.2: Behaviour of the controlled vehicle string with desired speed 0 as input.

This graph is exactly the same as Figure 2.2. This is an indication that the conversion to fixed-point numbers did not have influence on the behaviour of the system and that the translation to VHDL in general did not introduce errors in the system.

It is not possible to draw a position graph, which is done in Figure 2.2b. The absolute position of a vehicle is not available as an input or output of the system. This is not an issue, because in Listing 2.2 is shown that the position of a vehicle is calculated as the integral of the speed. This means that when the speed graph is equal in both simulations, the position graph will be equal too.

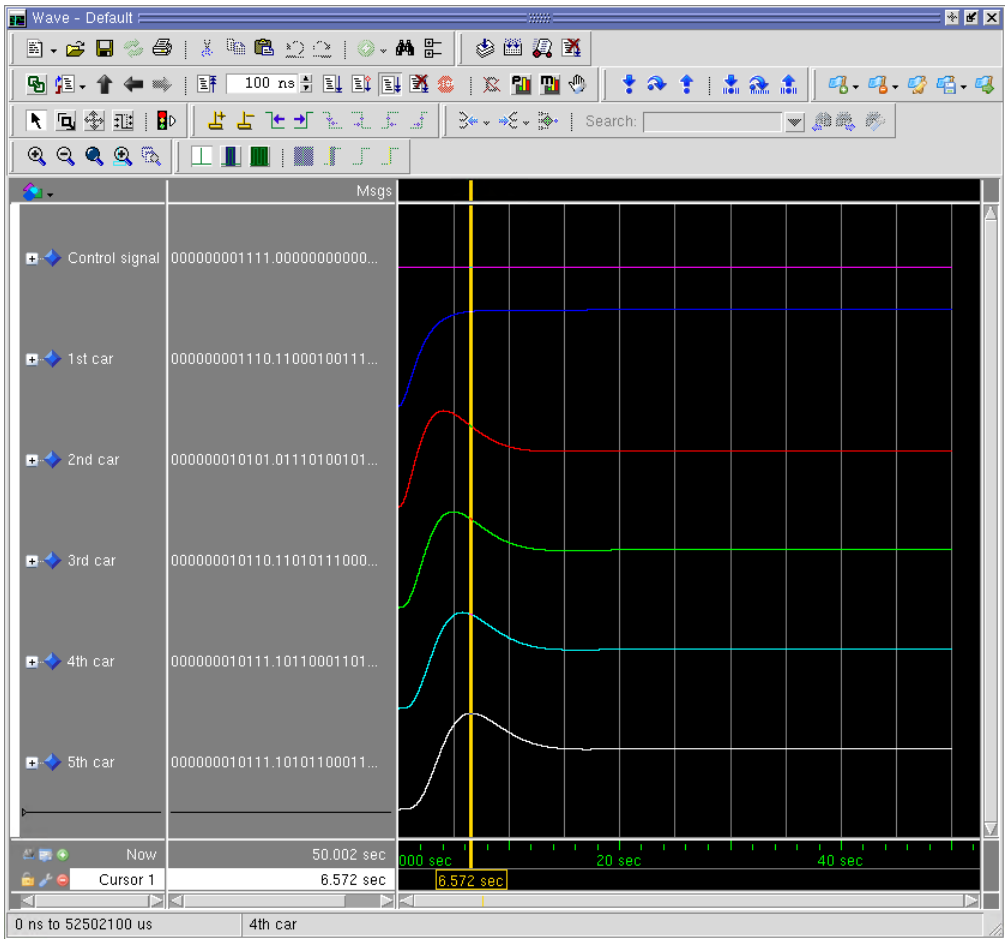
In the second simulation, the desired speed was set at 15m/s. The results of the VHDL simulation are shown in Figure 3.3.

Again, the shapes of the graphs correspond with the shape of Figure 2.3a and the values at the moment that the speed of vehicle 5 is at its maximum indicate identical behaviour as well. Please note that technically speaking not the absolute speed is shown, but the speed error. This is the speed difference between the actual speed and the steady state speed. If the steady state speed is set to 0, the speed difference is equal to the absolute speed. This change has no influence on the behaviour of the controller, as explained in Section 2.2.

To enable better comparison with the simulation results from Chapter 2, the values from QuestaSim are again exported and plotted in MATLAB. The resulting graphs are shown in Figure 3.4. Again, they do not differ from the Haskell simulation from Figure 2.3a.

The third simulation is the simulation with three different speeds over time. The results of that simulation are shown in Figure 3.5 and can be compared to Figure 2.4a.

As expected, these simulations results also correspond with the simulation of the Haskell

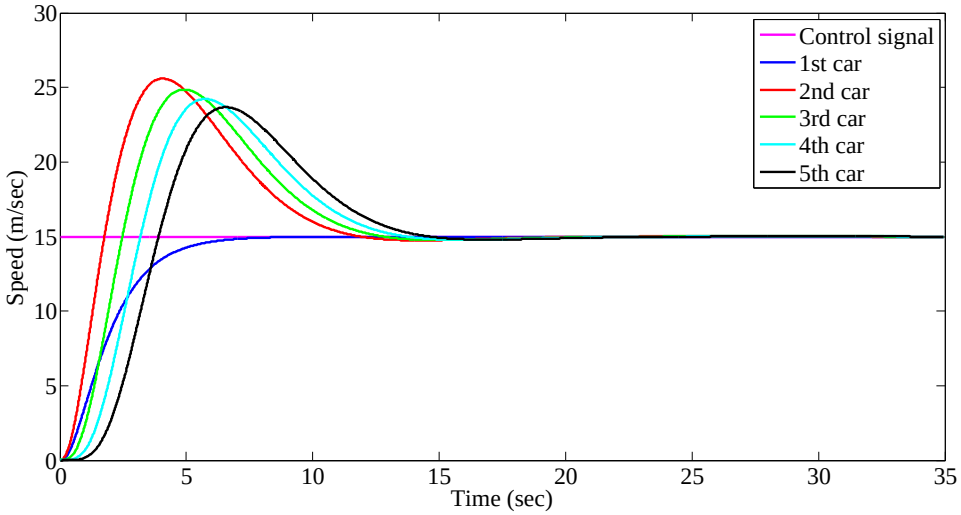


(a) Vehicle speeds

Figure 3.3: Behaviour of the controlled vehicle string with desired constant speed as input.

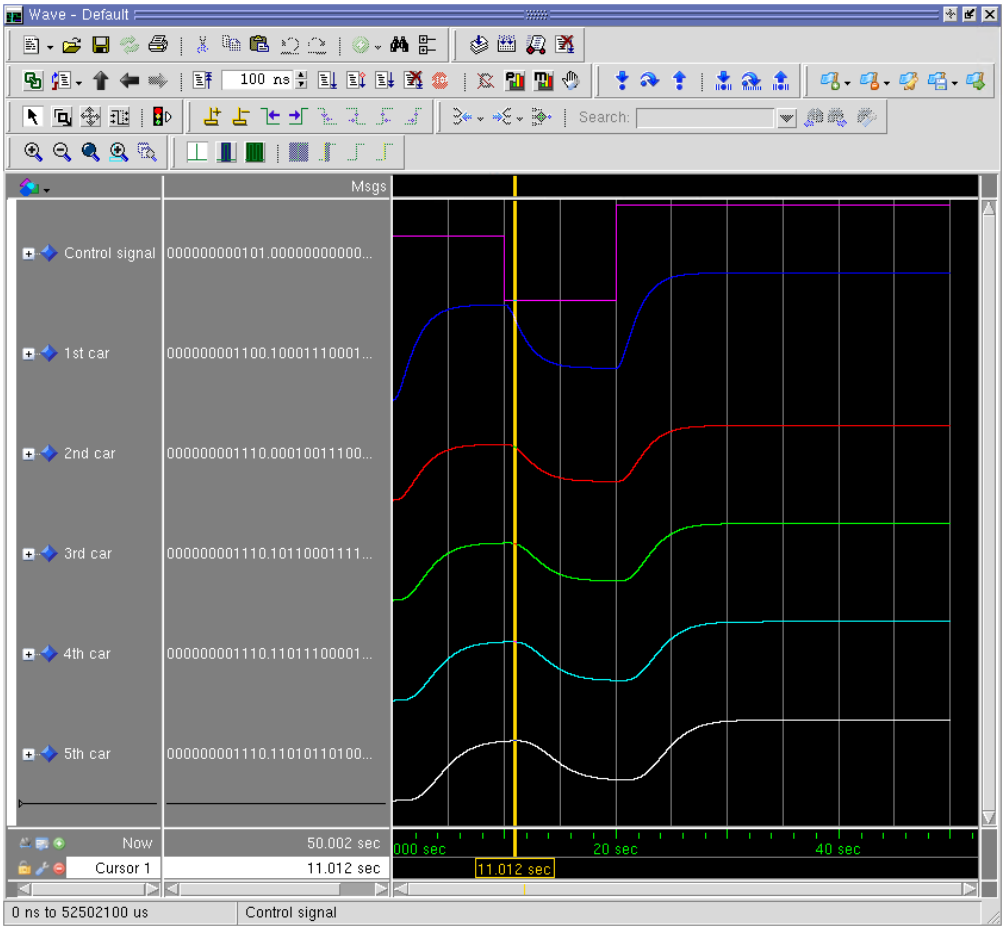
code, which is clearly visible when comparing the graph of the data from QuestaSim in Figure 3.6 with the Haskell simulation in Figure 2.4a.

The simulation of the system with a periodic function like Equation 2.17 is not performed in QuestaSim. The reason for that, is that such a function can only be defined using a lookup table in VHDL. Due to a lack of time, this table is not constructed and therefore, no simulations are performed with a periodic function as input.



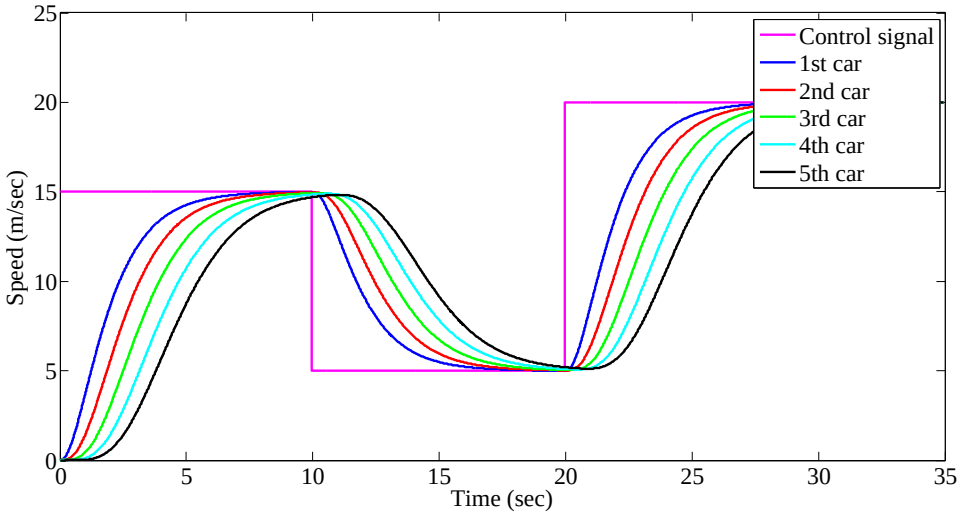
(a) Vehicle speeds

Figure 3.4: Behaviour of the controlled vehicle string with desired constant speed as input.



(a) Vehicle speeds

Figure 3.5: Response of the controlled vehicle string with different speed steps.



(a) Vehicle speeds

Figure 3.6: Response of the controlled vehicle string with different speed steps.

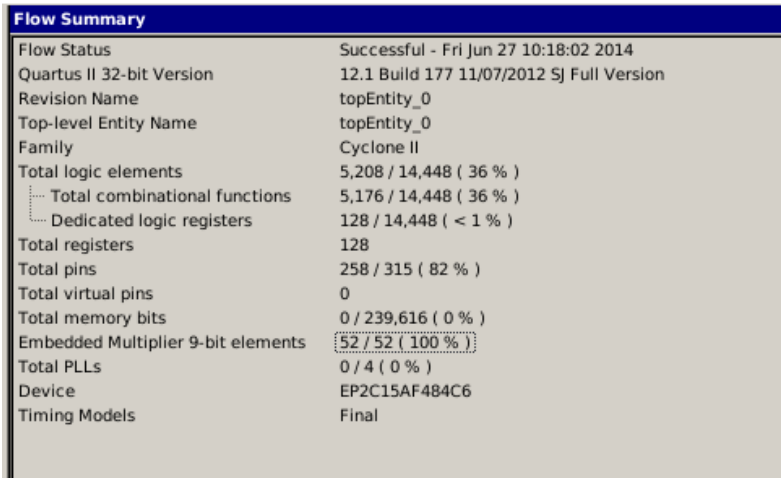
Chapter 4

Synthesis results

The simulation results from Subsection 3.2.2 show that the VHDL code behaves as expected. The next step in the hardware design process, is to synthesize the code so it can be programmed onto an FPGA. This chapter first describes the synthesis process and its results, followed by an overview of the resulting RTL (Register-transfer Level) scheme.

4.1 Synthesis

The VHDL code that is generated by CλaSH is fully synthesisable. This synthesis is performed in the program *Quartus*. For this research, a common used FPGA is selected as the target device: the 2C15 from the Altera Cyclone II series. The first synthesis that is performed, is only for the `Car` module. Not the `Platoon` module is synthesised, because it requires too many input and output pins, which are not available on the FPGAs that can be selected in Quartus. The summary of this synthesis is shown in Figure 4.1.



Flow Summary	
Flow Status	Successful - Fri Jun 27 10:18:02 2014
Quartus II 32-bit Version	12.1 Build 177 11/07/2012 SJ Full Version
Revision Name	topEntity_0
Top-level Entity Name	topEntity_0
Family	Cyclone II
Total logic elements	5,208 / 14,448 (36 %)
Total combinational functions	5,176 / 14,448 (36 %)
Dedicated logic registers	128 / 14,448 (< 1 %)
Total registers	128
Total pins	258 / 315 (82 %)
Total virtual pins	0
Total memory bits	0 / 239,616 (0 %)
Embedded Multiplier 9-bit elements	52 / 52 (100 %)
Total PLLs	0 / 4 (0 %)
Device	EP2C15AF484C6
Timing Models	Final

Figure 4.1: Synthesis summary of CACC model

In this summary is visible how many resources of the FPGA are used. The most interesting ones are the logic elements, the registers, the pins and the embedded multiplier elements. In this synthesis, 36% of the logic elements are used and less than 1% of the dedicated logic registers (128). The maximum achievable clock speed of this design is 21.01MHz.

There are 258 input/output pins used. This is as much as expected. The inputs of the system are a clock and a reset signal and the state of the previous vehicle. This state is a vector of four 32 bits fixed-point numbers, which adds up to 128 lines. The output of the system is the new state of the vehicle, which is also four times 32 bits. This adds up to a total number of 258 pins.

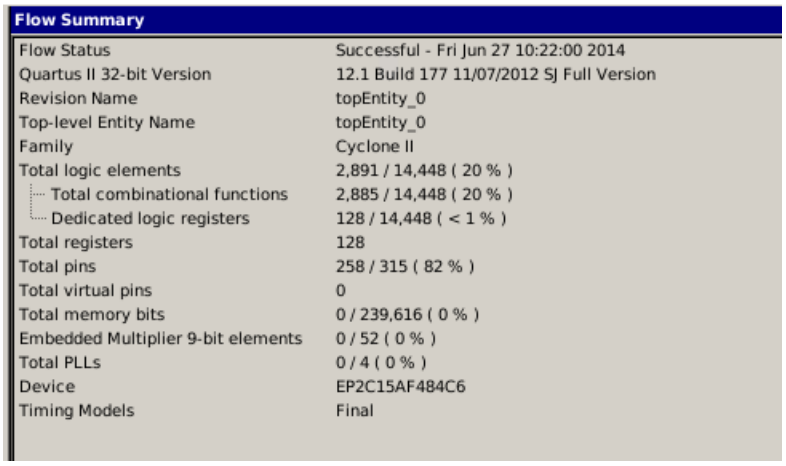
The result of this synthesis uses all embedded multipliers of the FPGA. As an experiment, there is also synthesis performed on a device with more multipliers and the result again used all multipliers. The cause of this, is that all multiplications that are performed in the system are with constant values, namely the values of the control matrices. It seems that the synthesis tool prefers embedded multipliers but it is able to use logic for the cases that there is no multiplier left.

It is possible to omit the use of embedded multipliers in the synthesis of Quartus by adding the lines from Listing 4.12 to the VHDL files that use multiplication.

```
attribute multstyle : string;
attribute multstyle of structural : architecture is "logic";
```

Listing 4.12: Omit hardware multipliers in VHDL

These lines force the synthesis tool to use logic for all multiplications. After adding these lines, a new synthesis is performed. In Figure 4.2, the synthesis summary is shown.



Flow Summary	
Flow Status	Successful - Fri Jun 27 10:22:00 2014
Quartus II 32-bit Version	12.1 Build 177 11/07/2012 SJ Full Version
Revision Name	topEntity_0
Top-level Entity Name	topEntity_0
Family	Cyclone II
Total logic elements	2,891 / 14,448 (20 %)
Total combinational functions	2,885 / 14,448 (20 %)
Dedicated logic registers	128 / 14,448 (< 1 %)
Total registers	128
Total pins	258 / 315 (82 %)
Total virtual pins	0
Total memory bits	0 / 239,616 (0 %)
Embedded Multiplier 9-bit elements	0 / 52 (0 %)
Total PLLs	0 / 4 (0 %)
Device	EP2C15AF484C6
Timing Models	Final

Figure 4.2: Synthesis summary of CACC model without hardware multipliers

As expected, there are now no embedded multiplier elements used. Noticeable is that there are also fewer logic elements used. In the previous run, 36% of the available units were used, in

this run only 20% of the units. This is most likely caused by optimizations in the logic. It might, however, result in a lower maximum clock frequency because of longer combinatorial paths.

It turns out that multiplication in pure logic is also slightly better for the maximum clock frequency that can be achieved, because it is changed to 23.44MHz. This is an increase of 11.5%.

4.2 RTL views

When the VHDL code is synthesized, it is possible to generate a RTL view of the design. As the name says, this is a view on the register-transfer level. This means that actual hardware parts which are used in the implementation can be identified.

In Figure 4.3, the RTL view of the whole system is shown. Of course, no real parts can be identified in this view, but what can be seen are a lot of wires, the registers containing the current state (the small blue blocks) and the complete processing unit as one green block.

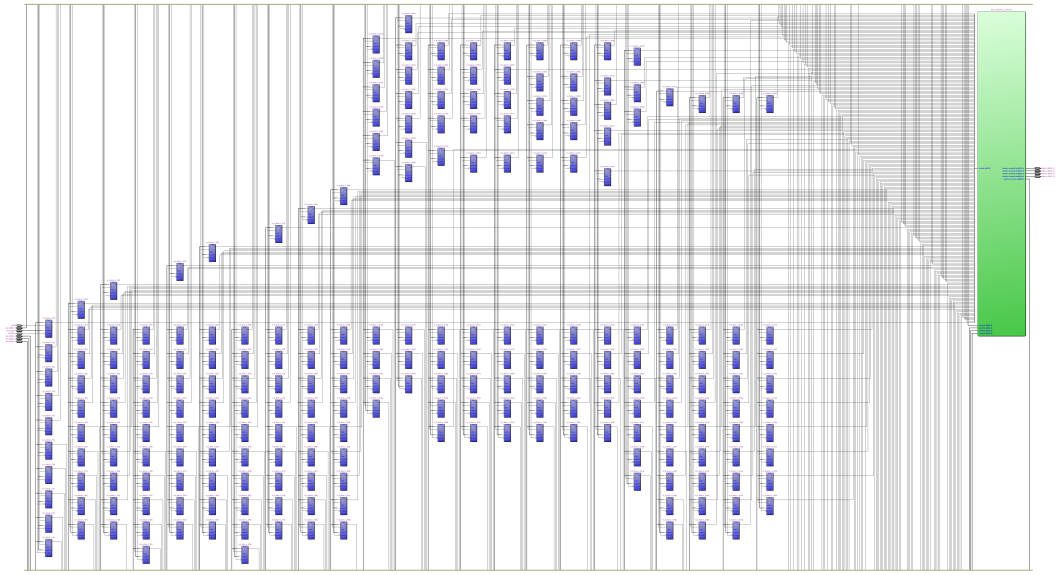


Figure 4.3: RTL view of the complete system

In the RTL viewer in Quartus, it is possible to zoom in on a block. When this is done for the green block, some more detail is visible. In the next zoom level of the green block, the view from Figure 4.4 becomes visible[1]. This represents the complete `Car.drive` function from Listing 3.9. The two big green blocks are the matrix-vector multiplications (`mxv` functions from Listing 2.3). On the left, the input from the predecessor is connected to the upper big green block (multiplication of the state of the predecessor and matrix `a1`) and the current state of the car is connected to the lower big green block (multiplication of the state of the current vehicle

[1] For better visibility, the two blocks on the left are shortened.

and matrix `a0`) via some smaller blocks. These smaller blocks are the logics to extract the state vector from the complete state of the system, which includes the speed, position and length of the vehicle. On the right half of the image, the `vzipWith` operations of the `Car.drive` function are performed and the new position and speed of the vehicle are calculated.

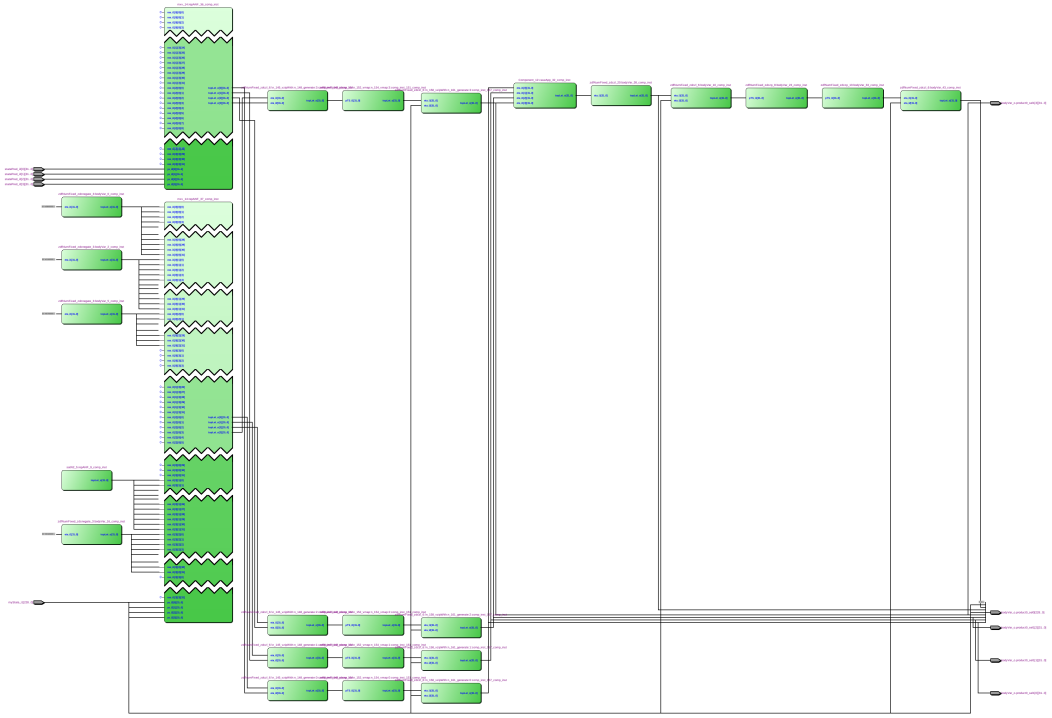


Figure 4.4: RTL view of the `Car.drive` function

In Figure 4.4, the most interesting parts are the two big blocks for the matrix-vector multiplications. When zooming in on one of these blocks, the image from Figure 4.5 becomes visible.

The matrix-vector multiplication is implemented as a `map` of dot products. In this investigation, the matrix is 4x4 and the state vector has a length of four, so the `map` results in four different dot products. This is visible in the RTL view, because there are four green blocks visible on the right side of the image. Each block represents one dot product. The fixed point numbers used in this investigation are 32 bits long. This explains why there are so many lines drawn as input. Unfortunately, Quartus sometimes merges these lines to one (because it represents one number), but sometimes there are drawn all individually.

The last step is to zoom in to a single dot product. A dot product is a sequential sum of different multiplications and this is also clearly visible in the RTL view of Figure 4.6, where four individual multipliers (Figure 4.7a) are visible on the left. The output of two multipliers go into an adder (Figure 4.7b) and the output of that adder goes in another adder, together with the output of another multiplier. This creates the cascade of the `fold` function.

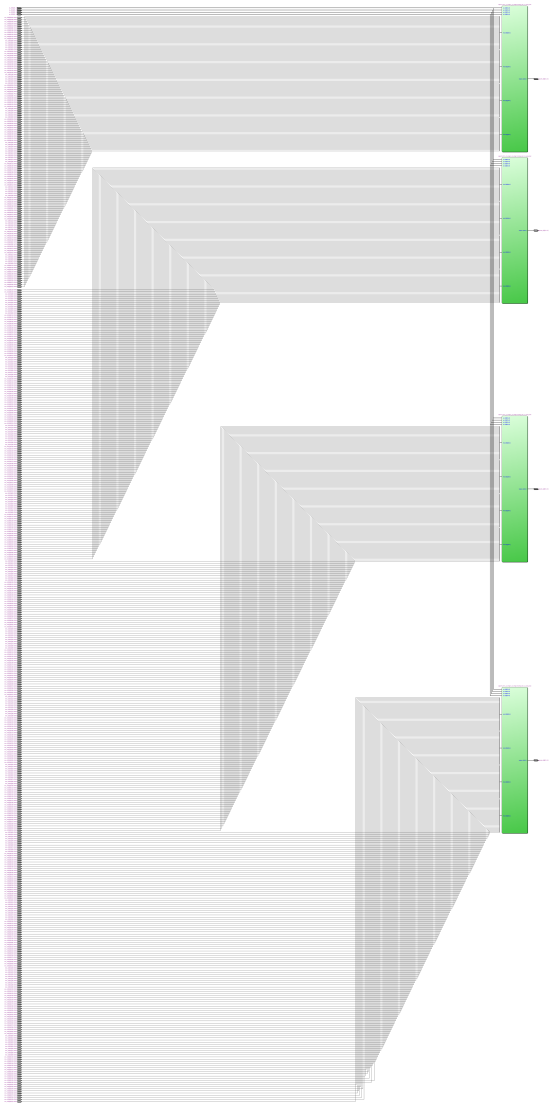


Figure 4.5: RTL view of the `mxv` function

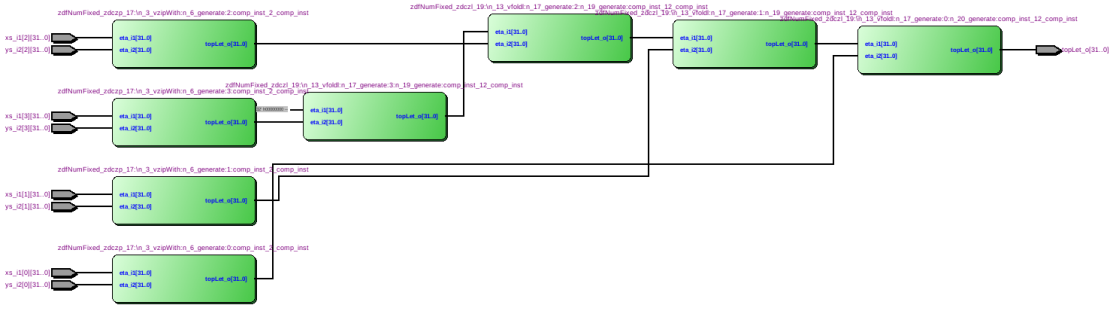


Figure 4.6: RTL view of the `(.*.)` function (dot product)

(a) Multiplication in the `zipWith`

(b) Adder in the `fold`

Figure 4.7: RTL view of the components in the dot product `(.*.)`

Chapter 5

Conclusion

In this investigation, a hardware design method was used which differs fundamentally from the traditional hardware design approach. This research focused on using a functional programming language to describe the behaviour of a cyber-physical system to investigate if there is a positive influence on the overall hardware design flow. The system that was chosen for this investigation is a CACC system.

The investigation started with an analysis of the control model using the mathematical relations between the input, the state and the state changes of the system. This analysis turned out to be useful to get a good insight into the structure of the system and it formed a good basis for the description of the system in the used functional programming language Haskell. Due to the nature of this language, this transformation from math to code was quite trivial, reducing the chance to introduce errors in the design. The developed system in Haskell was simulated using the GHC interactive environment to verify the correctness of the model and to find errors in the transformation from math to Haskell. In this simulation a virtual vehicle was placed in front of a platoon. This virtual vehicle was given several speeds and the response of the following vehicles in the vehicle string was analysed. The system turned out to behave as expected, indicating that the transformation from math to Haskell was performed correct.

The most common language to describe a hardware design, is VHDL or Verilog. If the description in one of these languages is synthesisable, actual hardware can be made of it. In this investigation, the compiler CλaSH is used to generate synthesisable VHDL code from a hardware description written in a subset of Haskell. The next step in the design process, was to rewrite the Haskell description to a CλaSH compatible subset of Haskell, such that the CλaSH compiler could compile it. There were just a few small modifications required, which include the conversion from lists to fixed size vectors and the change from floating-point data types to fixed-point data types. Both vectors and fixed-point numbers are available in the CλaSH Prelude library, so this library is used for this transformation. The transformation to CλaSH compatible Haskell code turned out to be straightforward with again a small chance for introducing errors. If any errors were made in this conversion, they were found in the next step: verification of the CλaSH compatible Haskell code. Using the interpreter of the GHC interactive environment, the code was again checked on correctness. All differences with the simulation results of the stock Haskell code were caused by the inaccuracy of the fixed-point numbers. By choosing the

number of bits in the fixed-point data type right, these errors were reduced to an acceptable level.

The last step in this design process was to compile the CλaSH compatible Haskell code to VHDL. This is an automated step so this step is trivial. After the compilation, the VHDL code was again verified and simulated with different speed functions for a platoon leader and the results were compared to the same simulations performed on the Haskell code. As expected, the results were consistent and the CλaSH compiler did not introduce new errors in the model. A property of the VHDL code that the CλaSH compiler produces is that it is fully synthesisable so with the resulting VHDL, actual hardware could be made. In this investigation, synthesis is performed for a common FPGA and it turns out that the whole simulator for five vehicles could run on one FPGA using only 20% of the logical units in the FPGA.

One thing that should be noted, is that in the beginning of the design process, while implementing the mathematical description into Haskell, the description was made with actual hardware and CλaSH limitations in mind. For example, the mathematical model was represented with a Mealy machine in mind. A Mealy machine uses its current state and the inputs to calculate outputs and a new state and the mathematical model was designed that way. If this choice was made later on, a lot of extra transformations had to be performed, introducing potential new errors. Furthermore, recursion was omitted from the beginning due to the lack of support on this matter in the CλaSH compiler. This is something that should be kept in mind while using this design methodology, because the use of recursion is very common in Haskell.

No quantitative comparison with the traditional design flow is performed in this investigation. It is expected that the design process takes more time when using the traditional way, consisting of writing a mathematical model in an imperative language like C, testing, rewriting it in a hardware description language like VHDL and testing again. All these conversions require a lot of modifications in the code, increasing the chance of errors.

To conclude this work, the research question asked in the beginning of this investigation could be answered:

- *To what extend is a hardware design method using a functional programming language suitable for the simulation and implementation of cyber-physical systems?*

When looking to the overall design process, it can be concluded that the design steps were straightforward and each transformation that was performed introduced just a small chance on errors. All of these errors could be found and corrected without a lot of effort, resulting in a fast and efficient hardware design process. The strong relation to the mathematical description of the problem keeps the whole design process structured and it makes this design approach especially suited for the hardware design of strong math-related systems, such as cyber-physical systems. This investigation shows that that there is potential for further improvement of this hardware design methodology and on the simulation of the designs.

Chapter 6

Recommendations

During the design process in this investigation and when reflecting to the process afterward, some things can be learned for further research, to improve the used design method and to place the conclusions in the right context. These things will be discussed in this chapter

6.1 Modular control systems toolbox

The last recommendation is somewhat related to the first one in this chapter. Control engineers often use block diagrams to represent a control system. Each block in the system is a function and the connections between the blocks are signals. It is expected that every individual block can be easily implemented in Haskell using higher-order functions.

When using a functional programming language like Haskell, higher-order functions can be exploited within the hardware design. Especially when small time differences are involved such as in cyber-physical systems, the use of these functions offers a great potential in efficient and fast simulations. During simulation, each signal can be represented as a function. When using a functional programming language, this function can be used as an argument for a higher-order function. This means that you can for example delay a signal with an exact amount of time, without changing the whole step size of the system. This will make simulation more efficient and therefore a lot faster. Besides that, higher-order functions can make for-loops superfluous and this eliminates the chance that for example off-by-one errors are introduced.

Unfortunately, this potential of using higher-order functions is not exploited enough in this investigation. The main cause for this, is that the CACC system does not require this precision. The incoming signals are from the speed sensor, the distance sensor and the wireless communication which reports the desired acceleration of a predecessor. All these signals have a period which is relatively long, even longer than the clock period of the whole system. The simulation step size will not be influenced by the required precision of the sensor data, so there is no fundamental need for this special use of higher-order functions.

A recommendation for further research is to implement a system which makes use of higher-order functions to be able to disconnect this relation between individual signal precision and the simulation step size, which is used for the whole system. If this is implemented, this would not only be an advantage for hardware design, but also for the simulation of embedded systems in

general. Due to the high potential of using a functional programming language in the simulation of cyber-physical systems, it might be very useful to create some kind of framework or toolbox which implements these block diagrams. By using a functional programming language, each block can for example have its own time step for integration, based on the needs at that point.

One of the greatest challenges in implementing such a toolbox is the implementation of feedback loops, because it might introduce dependencies which can not be fulfilled if each block does its calculation individually without any delay. There might be a solution for that by making use of the lazy evaluation in Haskell and the ability to work with infinite long lists, but this will introduce new difficulties for the implementation on FPGAs.

6.2 Adjustments in the model

In Chapter 1 of this report, it is mentioned that there will be investigated how much effort it would take to modify the designed CACC model. It is expected that modifications in the used mathematical model can lead to a new hardware description with just a little work, when intermediate steps from math to hardware are relatively easy. In this investigation, it is tried to expand the model with two-vehicle look-ahead techniques, but it was not successful due to the changes in the mathematical model. Due to the fact that the input of a direct predecessor is not independent of the input of the second predecessor, the model has to change a lot.

While this is mainly an issue on the mathematical model itself rather than the hardware design method that was used, it is recommended to investigate the effort it takes to modify the mathematical model to come to a new hardware design. If a mathematical model and a modified mathematical model are both available, one should be able to investigate this.

6.3 Improvements to CλaSH

While the CλaSH compiler is able to generate VHDL code from Haskell code, some features that are common in Haskell are not supported yet. The most important one is recursion. While there are often ways to get around recursion, recursion is very common and intuitive to use for people that have experience in writing Haskell. It would be a great addition to CλaSH if it gets support for recursion.

Another thing that could be improved, is the fixed-point library in the CλaSH prelude. This library is very recently build in CλaSH and only the most basic functionality is implemented. However, declaration of a fixed-point variable or constant makes use of template Haskell which is very limiting. Besides the extra effort that is needed to declare fixed-point variables, it also makes it very hard to simulate a CλaSH program. CλaSH offers a way to simulate the design by defining an input signal and an expected output signal. CλaSH will run the program using the given input and compare the output of the system with the given expected output. A signal can be defined as a list of values. The easiest way to do this in Haskell, is to use list comprehension:

```
sinewave :: [Float]
sinewave = [ sin t | t <- [0,0.01..(2*pi)]]
```

This function will return a list containing the values of $sin(t)$ on the interval $[0, 2\pi]$ with step size 0.01. If the input of the designed system must be a fixed-point number, list comprehension can not be used due to the requirement of template Haskell. This means that only very simple input signals can be used to simulate the design.

Another missing feature, is the division operation for fixed-point numbers. The CλaSH prelude has support for addition, subtraction and multiplication, but it lacks support for division and other basic mathematical operations. In combination with the way a constant is defined, it might be hard to define constants which are related to another constant, for example the multiplication matrix in (2.14). The values for h and τ are constant fixed-point numbers but due to the limitation that you can not divide fixed-point numbers, all values in the matrix are calculated first and filled in hard coded as constant value. This means that the code must be changed on several places if only the value for h changes, because a lot of values in the matrix must be changed.

Bibliography

[1] Rajesh Rajamani and Chunyu Zhu. Semi-autonomous adaptive cruise control systems. *Vehicular Technology, IEEE Transactions on*, 51(5):1186–1192, 2002.

[2] Shahdan Sapiee, Mohd Razali & Sudin. Road vehicle following system with adaptive controller gain using model reference adaptive control method. *International Journal of Simulation Systems, Science & Technology*, 11(5):24–32, 2009.

[3] Gerrit Naus, René Vugts, Jeroen Ploeg, René van de Molengraft, and Maarten Steinbuch. Cooperative adaptive cruise control, design and experiments. In *American Control Conference (ACC), 2010*, pages 6145–6150. IEEE, 2010.

[4] Gerrit JL Naus, René PA Vugts, Jeroen Ploeg, MJG Van de Molengraft, and Maarten Steinbuch. String-stable cacc design and experimental validation: A frequency-domain approach. *Vehicular Technology, IEEE Transactions on*, 59(9):4268–4279, 2010.

[5] Dik de Bruin, Joris Kroon, Richard van Klaveren, and Martin Nelisse. Design and test of a cooperative adaptive cruise control system. In *Intelligent Vehicles Symposium, 2004 IEEE*, pages 392–396. IEEE, 2004.

[6] Yingbo Zhao, Paolo Minero, and Vijay Gupta. On disturbance propagation in vehicle platoon control systems. In *American Control Conference (ACC), 2012*, pages 6041–6046. IEEE, 2012.

[7] Chi-Ying Liang and Huei Peng. Optimal adaptive cruise control with guaranteed string stability. *Vehicle System Dynamics*, 32(4-5):313–330, 1999.

[8] Chi-Ying Liang and Huei Peng. String stability analysis of adaptive cruise controlled vehicles. *JSME International Journal Series C*, 43(3):671–677, 2000.

[9] Lingyun Xiao and Feng Gao. Practical string stability of platoon of adaptive cruise control vehicles. *Intelligent Transportation Systems, IEEE Transactions on*, 12(4):1184–1194, 2011.

[10] Xiangheng Liu, Andrea Goldsmith, SS Mahal, and J Karl Hedrick. Effects of communication delay on string stability in vehicle platoons. In *Intelligent Transportation Systems, 2001. Proceedings. 2001 IEEE*, pages 625–630. IEEE, 2001.

[11] Jeroen Ploeg, Elham Semsar-Kazerooni, Guido Lijster, Nathan van de Wouw, and Henk Nijmeijer. Graceful degradation of cacc performance subject to unreliable wireless communication. In *Intelligent Transportation Systems (ITSC 2013), 2013 16th International IEEE Annual Conference on*, pages 1210–1216. IEEE, 2013.

[12] Shahdan Sudin and Peter A Cook. Two-vehicle look-ahead convoy control systems. In *Vehicular Technology Conference, 2004. VTC 2004-Spring. 2004 IEEE 59th*, volume 5, pages 2935–2939. IEEE, 2004.

[13] R Hallouzi, V Verdult, H Hellendoorn, and J Ploeg. Experimental evaluation of a cooperative driving set-up based on inter-vehicle communication. In *Proc. IFAC Symposium on Intelligent Autonomous Vehicles*, 2004.

[14] C. Kreuzen. Cooperative adaptive cruise control using information from multiple predecessors in combination with mpc. Master's thesis, Delft University of Technology, May 2012.

[15] Jeroen Ploeg, Dipan P Shukla, Nathan van de Wouw, and Henk Nijmeijer. Controller synthesis for string stability of vehicle platoons. *IEEE Transactions on Intelligent Transportation Systems*, 15(2), 2014.

[16] Swaroop Darbha and KR Rajagopal. Intelligent cruise control systems and traffic flow stability. *Transportation Research Part C: Emerging Technologies*, 7(6):329–352, 1999.

[17] Hilal Diab, I Ben Makhlouf, and Stefan Kowalewski. A platoon of vehicles approaching an intersection: A testing platform for safe intersections. In *Intelligent Transportation Systems (ITSC), 2012 15th International IEEE Conference on*, pages 1918–1923. IEEE, 2012.

[18] Jeroen Ploeg, Bart TM Scheepers, Ellen van Nunen, Nathan van de Wouw, and Henk Nijmeijer. Design and experimental evaluation of cooperative adaptive cruise control. In *Intelligent Transportation Systems (ITSC), 2011 14th International IEEE Conference on*, pages 260–265. IEEE, 2011.

[19] A. Firooznia. Controller design for string stability of infinite vehicle strings. Master's thesis, Eindhoven University of Technology, 2012.

[20] Roger W Brockett. Poles, zeros, and feedback: State space interpretation. *Automatic Control, IEEE Transactions on*, 10(2):129–135, 1965.

[21] Mary Sheeran. Hardware design and functional programming: a perfect match. *J. UCS*, 11(7):1135–1158, 2005.

[22] David M Goblirsch. An introduction to haskell with applications to digital signal processing. In *Proceedings of the 1994 ACM symposium on Applied computing*, pages 425–430. ACM, 1994.

[23] Zhonghai Lu, Ingo Sander, and Axel Jantsch. A case study of hardware and software synthesis in forsyde. In *System Synthesis, 2002. 15th International Symposium on*, pages 86–91. IEEE, 2002.

[24] Jim Grundy, Tom Melham, and John O'leary. A reflective functional language for hardware design and theorem proving. *Journal of Functional Programming*, 16(02):157–196, 2006.

[25] John Launchbury, Jeffrey R Lewis, and Byron Cook. On embedding a microarchitectural design language within haskell. *ACM SIGPLAN Notices*, 34(9):60–69, 1999.

[26] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in haskell. In *ACM SIGPLAN Notices*, volume 34, pages 174–184. ACM, 1998.

[27] Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. CλaSH: structural descriptions of synchronous hardware using haskell. In *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*, pages 714–721. IEEE, 2010.

[28] Koen Claessen. Embedded languages for describing and verifying hardware. *x*, 2001.

[29] J. Kuper. Hardware specification with CλaSH. Technical report, University of Twente, 2013.

[30] R. Wester, C. P. R. Baaij, and J. Kuper. A two step hardware design method using cλash. In *22nd International Conference on Field Programmable Logic and Applications, FPL 2012, Oslo, Norway*, pages 181–188, USA, August 2012. IEEE Computer Society.

[31] LL Hoberock. A survey of longitudinal acceleration comfort studies in ground transportation vehicles. *Journal of Dynamic Systems, Measurement, and Control*, 99(2):76–84, 1977.

[32] Christiaan Baaij. The clash-prelude package. Hackage, 2014. URL `http://hackage.haskell.org/package/clash-prelude`.

Appendix A

Introduction to Haskell

For the functional description of the design of the CACC simulator in this investigation, the programming language *Haskell* is used. For a better understanding of the code snippets in this report, a brief introduction of this language is given in this appendix. After reading this appendix, the reader should be able to understand most Haskell code that is written in this report. Please note that there are be many properties of and features in the language that are not discussed in this appendix but which broaden the possibilities of the language.

A.1 General

The programming language Haskell is a functional programming language, released in 1990. The language is named after the American mathematician Haskell Curry (1900-1982). A functional language is based on (mathematical) function application, which maps inputs to outputs. Haskell is also a purely functional language, which means that a function has no side effects and that there are no imperative language features like, for example, assignment statements. In that sense, it is comparable to a mathematical description, in which the statement $x = x + 1$ is invalid. Where such a statement is very common in imperative languages, they can not exist in a functional programming language.

Another feature of Haskell, is that it is a lazy language. This means that statements or expressions are only evaluated if their value is needed. This enables the use of infinite long lists and special cases of recursion. Both will be discussed later on in this appendix.

Haskell has an publicly available specification and programs that are written can be compiled to assembly code to create standalone programs. There also exist interactive interpreters which give the possibility to evaluate parts of Haskell code in a shell. The most common compiler and interpreter is the GHC. Due to the higher abstraction layer, Haskell programs are often slower than their counterparts written in an imperative language like C. They also tend to use more memory on the machine, but improvements in the compiler might address these properties in the future. On the other hand, development time can be a lot lower compared to an imperative approach, especially for complex mathematical problems. For the design of hardware and the compilation to VHDL, this is not an issue because the written program will not run on generic computer hardware.

A.2 Basic language components

In this section, the basic components of Haskell are presented, which form the core of the language.

A.2.1 Types

Just like other programming languages, Haskell has some basic types to represent data. Most types that are common in other programming language are also present in Haskell, like `Bool`, `Char`, `Int`, `Float` and `Double`. In addition, there is an unbound `Integer` type available in Haskell which can be used to represent numbers larger than the maximum integer value. The name of a data type always starts with a capital letter in Haskell.

Haskell also has support for user defined types. In a lot of cases, this makes the storage of data easier and more intuitive compared to the basic types. For example, grammatical genders can be stored in a user defined type `Gender`:

```
data Gender = Masculine | Feminine | Neuter
```

A value of a data type can also have "properties", consisting of other data types. For example, input devices for a personal computer can be represented by the following data type `InputDevice`:

```
data InputDevice = Keyboard Int |
                   Mouse    Int |
                   Joystick Int Int |
                   Touch
```

In this example, the "value" `Keyboard` and `Mouse` have a property of type `Int` representing the number of buttons. The `Joystick` has two properties: the number of buttons and the number of axes. The `Touch` type does not have any properties. Now, a mouse with three buttons can be represented as `Mouse 3`.

A.2.2 Lists

One of the most used data structures in Haskell is a list. A list can be seen as a sequence of values just like a mathematical sequence. All elements in a list must consist of the same type. Lists are defined in square brackets:

```
[2,4,5,6,10]
['H','a','s','k','e','l','l']
[True,True,False,True]
[]
```

The last line shows an empty list and due to the fact that Haskell is a lazy language, lists can also have an infinite length (contain an infinite number of elements). Some syntactic features exist to create these infinite lists, such as:

```
[1..]   --[1,2,3,4..
[2,4..] --[2,4,6,8..
[9,5..] --[9,5,1,-4..
```

Please note that the two dashes -- indicate comment. A list can also build up with the *cons* operator, written as a colon. This operator places an element in front of a list:

```
1 : [2,3]       --[1,2,3]
9 : [1..]       --[9,1,2,3..
3 : 5 : 6 : []  --[3,5,6]
```

As you can see on the last line, the colon associates to the right. This cons operator can be used in pattern matching which will be discussed later in this appendix.

Selecting an element in a list can be done with some predefined functions, or by using the index (starting at 0) of the element using double exclamation marks:

```
head [3,8,4]            --  3
last [3,7,2]            --  2
['a','b','c','d'] !! 2  -- 'c'
```

A.2.3 Functions

In the previous section, there are already two functions presented: `head` and `last`. There exist a lot more predefined functions in the language which are widely used. However, the language is only useful if it is possible to define your own functions. The process of making your own function is shown with an example of a function `increment`:

```
increment :: Int -> Int
increment n = n + 1
```

On the first line the *type declaration* is shown. On the left side of the double colon the name of the function is given, in this case `increment`. On the right side an arrow (`->`) indicates that it is a function. Left of the arrow is the input type mentioned and on the right of the arrow the output. In this case, the function `increment` maps the input of type `Int` to an output of type `Int`. After the type declaration, one or more lines containing the *function definition* is given. In the function definition, the first word is the name of the function and the rest (in this case the letter `n`) is argument. With this example function, the value `1` is added to the input number `n`, which is the correct behaviour of an increment.

If someone wants the successor of the number `4`, he would write `increment 4`, which will return `5`. People that are used to imperative programming languages would expect brackets around the function arguments, but in Haskell this is not required. The language is left associative, so executing `increment 4 * 5` is equivalent to `(increment 4) * 5` and equals 25.

It is also allowed to make a function with multiple arguments. In that case, each argument is represented with a data type and an arrow. Look for example to this function `adder`:

```
adder :: Int -> Int -> Int
adder n m = n + m
```

Here, the first two occurrences of `Int` in the function declaration represent arguments, the last occurrence is the output[1]. Analogous to a mathematical notation, the function can not have two or more outputs. It is, however, possible to output multiple values in a single data type. This can be done with a list containing multiple values, or by using a tuple, as in this function `sqrtDual`:

```
sqrtDual :: Float -> (Float,Float)
sqrtDual n = (a,b) where
    a = sqrt n
    b = -1 * (sqrt n)
```

This function calculates the square root of a `Float` and returns a tuple with the positive and negative value of the result. In the function declaration you see only one arrow, which means that there is one input and one output. Due to the fact that the output is a tuple of two, both values from the result are returned.

In the definition of `sqrtDual` you can see the word `where`. This is used in Haskell to define a local definition of variables and functions. You can compare it with a *local scope* in some imperative programming languages. In this function, the function call `sqrtDual` returns a tuple with `a` and `b`, but these did not exist yet in the program. They are defined within the function `sqrtDual` (note the indent). The `where` clause is often used to make a function easier to read, or to define helper variables within a function.

A property of Haskell, is that parentheses are often not required. The language has even some syntactic features to eliminate parentheses. As stated before, the language is left associative. However, there are situations that you want it to be right associative, in particular when the result of one function is used in another one. This is best shown with an example.

If someone want a function which strips the first element of a list (`tail`), then reverses the result (`reverse`) and finally filters it to only positive numbers (`filter (>=0)`), he could write the following[2]:

```
myFunction :: Real a => [a] -> [a]
myFunction xs = filter (>=0) (reverse (tail xs))
```

As you can see, a lot of parenthesis are used. Haskell features a *function application* operator `$`, which associates to the right. This means that using this operator, the function `myFunction` can be rewritten as follows:

```
myFunction :: Real a => [a] -> [a]
myFunction xs = (filter (>=0).reverse.tail) xs
```

The dot in between the different functions indicates *function composition*. It is the Haskell implementation of the mathematical function composition:

$$(f \circ g)(x) = f(g(x)) \tag{A.1}$$

[1] More correctly, the function `adder` requires one argument and returns a function that on its turn requires one argument and has one output. The function declaration is analogous to `Int -> (Int -> Int)`. Most of the times it is possible to interpret it as multiple inputs, but sometimes this property of returning a function with one argument less, which is called partial application, is used.

[2] The used type class for `a` here is `Real` and not `Num`, because `Num` also includes complex numbers and therefore the operator >= is not available for the class `Num`.

In this case, it means that the result of the function `tail` is used as the input for the function `reverse` and that result is used as the input for the function `filter`. Using the function application operator `$` and function composition, a lot of parenthesis can be eliminated, making the code more compact and in case of many nested functions also better readable.

A.2.4 Pattern matching

Very often, functions in Haskell will make use of *pattern matching*. Pattern matching can be seen as multiple function definitions and the first line that matches the applied input is evaluated. An example to make this more clear:

```
isEmpty :: [Int] -> Bool
isEmpty [] = True
isEmpty _  = False
```

The function `isEmpty` must be called with a list of `Int`s as argument and returns a `Bool`, which can be seen in the type declaration. The interpreter then evaluates the line that matches first. If the argument is an empty list (`[]`) it returns `True`. If the list is not empty (e.g. it does not match `[]`), the next line is evaluated. Here, an underscore `_` is a *wildcard* and it matches always. So, if the list is not empty, the function always returns `False`.

Pattern matching can also be used to extract elements from a list. If the function `head` would not be available in Haskell, you could define your own function `firstElement` as follows:

```
firstElement :: [Int] -> Int
firstElement []     = error "no first element in empty list"
firstElement (x:xs) = x
```

The first line of the type definition is evaluated when the list is empty. Such a list has no first element, so an error is raised. When the input does not match an empty list, the next line is evaluated. The list at the input is split up in two parts with the cons operator into the first part `x` and the rest `xs`. The return value is `x`. Please note that an empty list also is a list. If the input of the function is a list of one element, `x` will contain this element and `xs` will be an empty list. Due to the fact that the value of `xs` is never used in this function, it might be replaced with the wildcard character.

A final note here, is that it is not required to write a type declaration in front of a function. It is however very useful when debugging code because the interpreter gives you detailed information about conflicting types in case of an error in the code.

Haskell also has support for polymorphic functions, which are functions that can handle an input of different types. In that case, the type in a type definition is given with a single letter, for example:

```
firstElement :: [a] -> a
firstElement []     = error "no first element in empty list"
firstElement (x:xs) = x
```

Regardless of the of elements in a list, it is possible to return the first element. This makes it intuitive to make this function `firstElement` polymorphic. That is done by using a letter in the type declaration. The function declaration indicates now that the return type is equal to the type of elements in the list.

If the letter 'a' is used in the function definition, it always represents the same data type. If the function would return a different data type, another letter should be used. The previous discussed `increment` function can only be used for numerical types and not, for example, for `Bool` types. The function can be made polymorph this way:

```
increment :: Num a => a -> a
increment n = n + 1
```

In the function declaration, there is a limitation given to 'a'. The part between the double colon `::` and the arrow `=>` tells that only numerical data types are allowed for 'a'. This results in a function that can also increment `Float`s and `Double`s.

A.2.5 Recursion

Recursive functions and recursive data types are an important feature in Haskell. Many functions rely on recursion. Most of the time, recursion in functions is implemented using pattern matching. One pattern is defined for the edge case to prevent infinite recursion and another pattern is used for the actual algorithm. A good example is a `listReverse` function, which will reverse the order of elements in a list. There exists a default `reverse` function in Haskell, so this example is just to make the idea of recursion in Haskell clear.

```
listReverse :: [a] -> [a]
listReverse []     = []
listReverse (x:xs) = listReverse xs ++ [x]
```

This recursive function returns an empty list if it is called with an empty list as argument. If a list is given as parameter, the first element of the list is concatenated (using the `++` operator) to the end of the reverse of the rest of the list. Each iteration, `xs` will become one smaller, until it is an empty list. This will end the recursion and the final result is returned.

Besides recursion in functions, it is also possible to use recursion in data types. This means that the data type might have a value which is another value of the same type. This feature can be used for example in tree data structures:

```
data TreeNode = Leaf   Int |
                Parent Int TreeNode TreeNode
```

In this data type, each node in the tree has an `Int` value. The most basic value is the `Leaf`. Besides this `Leaf`, there is a value `Parent` which has an `Int` value and two children of type `TreeNode`, which can be either a `Leaf` or a `Parent`.

A.2.6 Higher-order functions

The real power of Haskell, especially for cyber-physical systems, comes from the support for higher-order functions. These are functions of which an argument is another function. Many predefined higher-order functions exist in Haskell. In this example, the `map` function is used. This is a default function in Haskell, but it can be used to explain how higher-order functions work.

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

The `map` function applies the function `f` which is given as first argument (showed in the function declaration as `(a->b)`) to each element in the list in the second argument. It returns a list containing the result for each of these operations. As you can see, this function makes use of recursion and pattern matching. If one would calculate the square of each element in a list `xs`, this could be done by running `map (^2) xs`.

A.3 Example

In this section, the different properties of Haskell are shown in one function. As an example, a function `integrate` is used, which is able to calculate the values of the integral of a given function:

```
data Technique = Midpoint | Trapezoid | Simpson

-- Calculates the integral with technique t of function f,
-- starting at value l and using time step dt
integrate :: (Fractional a) => Technique -> (a -> a) -> a -> a -> [a]
integrate t f l dt = y where
    y = scanl (+) 0 (hInt t l)
    hInt Midpoint  n = f (n+0.5*dt) * dt : hInt Midpoint (n+dt)
    hInt Trapezoid n = (dt*0.5*((f n)+(f (n+dt)))) : hInt Trapezoid (n+dt)
    hInt Simpson   n = dt/6 * (f (n) + 4*f (n+dt) + f (n+2*dt)) :
                       dt/6 * (f (n) + 4*f (n+dt) + f (n+2*dt)) :
                       hInt Simpson (n + 2*dt)
```

The function takes four arguments: an integration technique `t` (of the self defined type `Technique`), a function `f` which must be integrated, a lower limit `l` at which the integration will start and a step size `dt`. The function returns a list of the integration values starting at `l`, following by a new value every `dt`. In the function declaration you can see that polymorphism is used and that type ‘a’ must be of type `Fractional`.

Every integration technique calculates the integral at an interval related to the step size `dt`. Recursion is used to do this calculation until infinity. In this case, the Haskell build-in function `scanl` applies the function `(+)` to the new calculated value and the previous return value. The starting value (“the previous return value at the first iteration”) is `0`. This results in the sum of the integral over all individual intervals, which is an approximation of the actual integration.

The implementation of each integration technique follows quite directly from the mathematical description of the algorithm. This will now be shown by discussing each method individually. In the Haskell code, pattern matching is used to evaluate the right parts of the code.

The `Midpoint` technique assumes the value in an interval `dt` constant with a value equal to the outcome of the function exactly in the middle of the interval. For example, the integral of a function $f(n)$ between values $n = 2$ and $n = 3$ will be $f(2.5) \cdot dt$. Due to the use of the `scanl` function in the shared part of the code, the first element of the resulting list will always be 0. Therefore, calling the helper function `hInt` with value `n` should calculate the integral over interval $[n, n + dt]$, which is approximated by taking the value at $t = n + 0.5 \cdot dt$.

The `Trapezoid` algorithm approaches the shape of function $f(n)$ as a linear function at one interval `dt`. The area under the line is then calculated by averaging the values of $f(n)$ at the start and at the end of the interval and multiplying it with dt. For example, on the interval between $n = 2$ and $n = 3$, the mathematical notation would be $\frac{1}{2} \cdot (f(2) + f(3)) \cdot dt$.

The `Simpson` method used Simpson's rule to calculate the integral. This algorithm uses quadratic polynomials to approach the function within a certain interval. There are three points required before this approximation can be done, so there are two intervals taken at once. The integral

$$\int_{n-dt}^{n+dt} f(n)\mathrm{d}n \tag{A.2}$$

is calculated using

$$\frac{dt}{3}(f(n - dt) + 4 \cdot f(n) + f(n + dt)) \tag{A.3}$$

. In this Haskell-implementation of the algorithm, the value of the integral at interval $[n - dt, n]$ is assumed to be half of the integral at interval $[n - dt, n + dt]$. This is implemented by repeating this value once, before calculating the new value.

If one would calculate the integral of the function x^2 at the interval $[5, 6]$ with step size 0.001 using Simpson's Rule, the following command is executed:

```
(integrate Simpson (^2) 5 0.001)!!1000
```

, which returns element 1000 from the resulting list. One could of course define a wrapper around this function, which calculates the value with index 1000 from the given interval and the step size (for example $(6 - 5)/dt$).

Appendix B

Two-vehicle look-ahead

In this investigation is tried to implement a two-vehicle look-ahead control method in the model. Due to the nature of the used mathematical model of the one-vehicle look-ahead method, this expansion to two-vehicle look-ahead was not trivial. Because of time restrictions, this model is not finished but it is briefly discussed here for future reference.

B.1 Expansion of current model

The first approach, was to extend the used mathematical model for one-vehicle look-ahead to a two-vehicle look-ahead model using the same analysis as used in Section 2.2. For a two-vehicle look-ahead model, the desired position $s_{r,i}$ can be constructed as a combination of $s_{r,i,i-1}$ and $s_{r,i,i-2}$. These variables are the desired positions of vehicle i when only taking into account information of its direct predecessor ($s_{r,i,i-1}$) and the vehicles of before that one ($s_{r,i,i-2}$) respectively. This leads to the expression

$$s_{r,i} = p_1 s_{r,i,i-1} + p_2 s_{r,i,i-2} \tag{B.1}$$

, in which the weights $p_1 + p_2 = 1$.

The desired positions with respect to only one vehicle at a time, are

$$s_{r,i,i-1} = s_{i-1} - L_i - r_i - hv_i \tag{B.2}$$

and

$$s_{r,i,i-2} = s_{i-2} - 2L_i - 2r_i - h(v_i + v_{i-1}) \tag{B.3}$$

The position error e_i of a vehicle can be calculated by subtracting the actual position from each desired position:

$$\begin{aligned} e_i &= p_1(s_{r,i,i-1} - s_i) + p_2(s_{r,i,i-2} - s_i) \\ &= p_1(s_{i-1} - s_i - L_i - r_i - hv_i) + \\ &\quad p_2(s_{i-2} - s_i - 2L_i - 2r_i - h(v_i + v_{i-1})) \end{aligned} \tag{B.4}$$

Analogous to the analysis of one-vehicle look-ahead, three error states $e_{n,i}$ are defined:

$$\begin{pmatrix} e_{1,i} \\ e_{2,i} \\ e_{3,i} \end{pmatrix} = \begin{pmatrix} e_i \\ \dot{e}_i \\ \ddot{e}_i \end{pmatrix} = \begin{pmatrix} p_1(s_{i-1} - s_i - L_i - r_i - hv_i) + \\ p_2(s_{i-2} - s_i - 2L_i - 2r_i - h(v_i + v_{i-1})) \\ \\ p_1(v_{i-1} - v_i - ha_i) + \\ p_2(v_{i-2} - v_i - h(a_i + a_{i-1})) \\ \\ p_1(a_{i-1} - a_i - h\dot{a}_i) + \\ p_2(a_{i-2} - a_i - h(\dot{a}_i + \dot{a}_{i-1})) \end{pmatrix}, \quad 2 \leq i \leq m \tag{B.5}$$

, with the time derivatives (compare the steps from (2.7))

$$\dot{e}_{1,i} = \dot{e}_i = e_{2,i} \tag{B.6}$$

,

$$\dot{e}_{2,i} = \ddot{e}_i = e_{3,i} \tag{B.7}$$

and

$$\begin{aligned} \dot{e}_{3,i} &= \dddot{e}_i \\ &= p_1(\dddot{s}_{i-1} - \dddot{s}_i - h\dddot{v}_i) + p_2(\dddot{s}_{i-2} - \dddot{s}_i - h(\dddot{v}_i + \dddot{v}_{i-1})) \\ &= p_1(\ddot{a}_{i-1} - \ddot{a}_i - h\ddot{a}_i) + p_2(\ddot{a}_{i-2} - \ddot{a}_i - h(\ddot{a}_i + \ddot{a}_{i-1})) \\ &= p_1(-\frac{1}{\tau}\dot{a}_{i-1} + \frac{1}{\tau}\dot{u}_{i-1} + \frac{1}{\tau}\dot{a}_i - \frac{1}{\tau}\dot{u}_i - h(-\frac{1}{\tau}\ddot{a}_i + \frac{1}{\tau}\ddot{u}_i)) + \\ &\quad p_2(-\frac{1}{\tau}\dot{a}_{i-2} + \frac{1}{\tau}\dot{u}_{i-2} + \frac{1}{\tau}\dot{a}_i - \frac{1}{\tau}\dot{u}_i - h(-\frac{1}{\tau}\ddot{a}_i + \frac{1}{\tau}\ddot{u}_i - \frac{1}{\tau}\ddot{a}_{i-1} + \frac{1}{\tau}\ddot{u}_{i-1})) \\ &= -\frac{1}{\tau}e_{3,i} - \frac{1}{\tau}q_i + \frac{1}{\tau}(p_1u_{i-1} + p_2u_{i-2}) + \frac{1}{\tau}p_2h\dot{u}_{i-1} \end{aligned} \tag{B.8}$$

In (B.8), a new variable q_i is defined. This is the input of the actual controller and is defined as

$$q_i \triangleq p_1h(\dot{u}_i + u_i) + p_2h(\dot{u}_i + u_i) \tag{B.9}$$

This leaves the term

$$\frac{1}{\tau}p_2h\dot{u}_{i-1} \tag{B.10}$$

, which includes the time derivative of the desired acceleration of the predecessor. Unfortunately, this information is not known to the current vehicle, so this model can not be used in practice.

B.2 Two separate input vectors

The second approach that is used for a two-vehicle look-ahead control strategy, is to use the information of the two direct predecessors as independent state vectors. Using weights comparable to the p variables in Equation B.1, the calculation of the state change as shown in Equation 2.14 can be expanded as follows:

$$\begin{pmatrix} \dot{e}_i \\ \dot{v}_i \\ \dot{a}_i \\ \dot{\jmath}_i \end{pmatrix} = \begin{pmatrix} 0 & -1 & -h & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ \frac{k_p}{h\tau} & -\frac{k_d}{h\tau} & -\frac{1+k_d h+k_{dd}}{h\tau} & -\frac{h+\tau+k_{dd}h}{h\tau} \end{pmatrix} \begin{pmatrix} e_i \\ v_i \\ a_i \\ \jmath_i \end{pmatrix} + \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & \frac{k_d}{h\tau} & \frac{1+k_{dd}}{h\tau} & \frac{1}{h} \end{pmatrix} \left(p_1 \begin{pmatrix} e_{i-1} \\ v_{i-1} \\ a_{i-1} \\ \jmath_{i-1} \end{pmatrix} + p_2 \begin{pmatrix} e_{i-2} \\ v_{i-2} \\ a_{i-2} \\ \jmath_{i-2} \end{pmatrix} \right) \tag{B.11}$$

, where $p_1 + p_2 = 1$.

With this model, a weighted average between the state errors of both predecessors is used. This means that when p_1 is set to 0, no information of the direct predecessor is used, but only of the vehicle in front of the direct predecessor. This will result in a faster system, because information about changes in the speed and the acceleration of the vehicle string is faster available for the current vehicle. However, to avoid collision, it is important to keep using the information of the direct predecessor. If that vehicle rapidly lowers its speed for some reason, a collision will occur if no information of that vehicle was used. Using this model for the control strategy could lead to a desired position which is in front of the direct predecessor. The same holds for the desired speed, acceleration and jerk, which could all be larger than the values for the direct predecessor. This means that there should be some kind of trade-off between fast response and collision avoidance when determining the values for p_1 and p_2.

In Equation B.11, the original matrix a_1 is still visible. One could also think of a model in which the state vectors of both predecessors is multiplied with completely independent matrices. In that case, the state space equation can be constructed as follows:

$$\begin{pmatrix} \dot{e}_i \\ \dot{v}_i \\ \dot{a}_i \\ \dot{\jmath}_i \end{pmatrix} = A_0 \begin{pmatrix} e_i \\ v_i \\ a_i \\ \jmath_i \end{pmatrix} + A_1 \begin{pmatrix} e_{i-1} \\ v_{i-1} \\ a_{i-1} \\ \jmath_{i-1} \end{pmatrix} + A_2 \begin{pmatrix} e_{i-2} \\ v_{i-2} \\ a_{i-2} \\ \jmath_{i-2} \end{pmatrix} \tag{B.12}$$

To determine the values in the A matrices, a similar derivation as in Section B.1 could be performed, but also for this case it is required that the term $\dot{u}_{i-1}$ is substituted in terms that are available to the current vehicle.