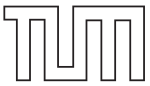
Professorship of
Hybrid Control Systems



Technische Universität München

Master Thesis

Profrssorship of Hybrid Control Systems

TECHNICAL UNIVERSITY MUNICH

Concept and implementation of a communication protocol between autonomous vehicles and a cloud-based controller

Georgios Lionas

Advisor:	Mahmoud Khaled M.Sc.
Professor:	Prof. Dr. Sebastian Steinhorst
Started on:	01.05.2019
Handed in on:	28.09.2019

Statutory declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Munich, 29.09.2019

Georgios Lionas

Abstract

In today's connected world, everything seems possible. Buzzwords, like big data and artificial intelligence can be found in many scientific papers and journals. These technologies are very interesting for autonomous driving. However there is a problem with them: Proof of functionality. These algorithms depend on training data and complex models that are interconnected. In order to prove the correct function of these systems, many tests have to be made. There is an alternative. It is called symbolic control. With symbolic control, a controller is created based on the underlying model and the specification. First, a discrete model is constructed from the continuous model of a vehicle. Next, specifications are defined, which define the bad states and the target states. Finally, a discrete controller is synthesised based on the specifications. This discrete controller avoids all bad states, while trying to reach the target set. The problem with symbolic control is like many problems in engineering the state-space problem. Models, with high-dimensional state and input spaces need many computing resources in order construct a controller. If a real-time construction is needed, these resources grow analogously. Nonetheless, there is a solution to this problem with cloud-based computing. Cloud providers provide the user with as many computing resources as needed. A symbolic control algorithm could run in the cloud providing real-time inputs to a vehicle. What remains to be solved, is the communication protocol connecting the vehicle with the cloud.
This thesis aims to create and implement a concept of a communication protocol that could be used to remotely control an autonomous vehicle. The protocol will be called Open Automotive Control Protocol. The protocol should be easy to use and provide the engineers with the flexibility they need. It should also be lightweight and secure. On top of the protocol, an enterprise application is built in order to provide a sample application, which automotive engineers could use in order to use the protocol.

Contents

List of Figures

List of Tables

Abbreviations

FSC Formally synthesised controller

IoT Internet of things

CPU Central processing unit

IT Information technology

IaaS Infrastructure as a service

SaaS Software as a service

PaaS Platform as a service

TUM Technical university of Munich

OSI Open system interconnection model

ADAS Advanced driver assistance system

ACC Adaptive cruise control

OEM Original equipment manufacturer

ECU Electronic control unit

SISO Single input, single output

ZOH Zero-order hold

LTL Linear temporal logic

HPC High performance computing

CN Compute nodes

CU Computing units

PE Processing elements

CK Computation kernel

REST Representational state transfer

SUMO Simulation of urban mobility

API Application programming interface

AUTOSAR Automotive open system architecture

HTTP Hypertext transfer protocol

DFD Data flow diagram

SQL Structured query language

MVC Model-view controller

JWT JSON web token

AES Advanced encryption standard

AWS Amazon web services

1

Introduction

The world has changed with a fast rate in recent years due to technological improvements. As time goes on, people realise that almost every real problem has a solution. The answer to many technical and non-technical problems is simply science. What seemed impossible fifty years ago, has become reality now or will become reality in the near future.

1.1 Motivation

In 1886, Carl Benz introduced a concept of a vehicle to the world that solved the problem of that time: Mobility. This was a crucial moment in history, which is considered the birth of the automobile. The cognition of distance changed. The perception of reality depends strongy on the environment in which the individual was raised and the experience of that human being in our physical world.

Figure 1.1: The first automobile from 1886, invented by Carl Benz

Since 1886 the automotive industry has evolved. The word "automotive" is a composition of the greek word "$\alpha\upsilon\tau\acute{o}\varsigma$" and the latin word "mobilis". It means that the

vehicle is moving autonomously. In 1886 it was not moving autonomously. The driver had to control the vehicle. He had to sense the environment and to provide actions to control the yaw and the acceleration of the vehicle. Today, the semantics of this word and the actual representation of that physical object, the automobile, in our world, are almost the same.

What is the reason for this transormation? The answer is the following: Information technology and computers. With Claude Shannon, exploring deep insights in the mathematical theory of information technology and the parallel inventions and improvements of hardware chips, it was a matter of time until the world realised the potential of this strong duo. What started with the definition of a bit as information measure has now become a world-wide phenomenon. The world as we know it today depends strongly on technology. Smart grids are distributing energy in an efficient manner, providing whole countries with energy. Biologists are exploring the human DNA to fight biological malfunctions with the help of software. Individuals rely on smart station towers, which control the data flow of an area, allowing them to communicate all over the world. The use of software in various areas has increased dramatically. There are some good reasons for that:

1. Software is easy to change

2. Software is cheap

3. Everyone can learn how to program and develop solutions

4. Digital communication is a de-facto standard

As software is used almost everywhere we live in a time period that is called Industry 4.0. It targets another industrial revolution with the help of software systems, interdevicecommunication, data-analysis and machine learning concepts. Buzzwords like big data or IoT are all over the scientific and industrial world. What sounded like science fictions some years ago, becomes now reality.

While new ideas and inventions find their place in the world, the automotive industry and academia are researching about autonomous vehicles. Today's vehicles are "cyber-phsical systems". According to [LS17], a cyber phyisical system is a composition of computation with physical processes. This provides the vehicle the capability to interact with the environment in a predefined logical manner. Figure 1.2 shows the structure of a cyber-physical system.

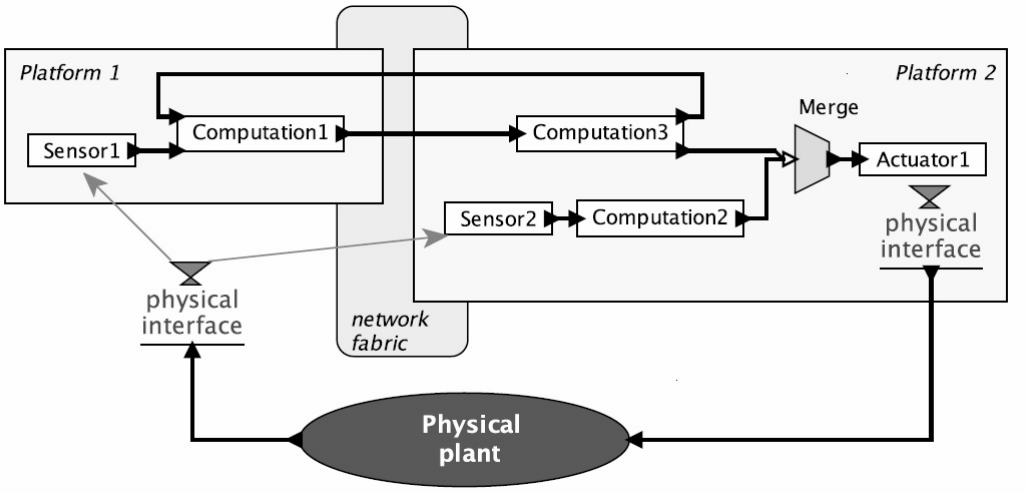


Figure 1.2: Structure of an cyber-physical system, [LS17]

In addition to the mainstream car manufacturers, computer related companies are now researching about autonomous driving. These companies do have a big advantage over the oldschool car manufacturers. They are much more specialized in software. While the model dynamics of vehicles have not changed drastically in the last years, software development take 180 degree turns every few years. With mostly mechanical engineers in their back, it is difficult to compete with these software experts in this field.

There are a lot of methods to achieve autonomous driving. In simple cases it could be done with a few if-else statements and some sensors. If it gets more complex, then some state-machine approach could solve these cases. Another approach is the machine learning approach, where the object constantly learns from some training data until some cost-function criterium is met. But there is a problem with these principles: Safety. How can you actually verify that the car will behave like expected? Can you make sure that there is no such situation, in which the car is going to hurt someone, a so called "bad state"? Pendants of these methods rely on tests of the systems to make sure that the system behaves like expected. There is usually some inherent test team that gets the same requirements like the developer team from the project manager and their job is to test the system against these requirements. Testing often requires low mathematical skills and is cheap. But there is a problem with this test-driven approach. Like Edsger W. Dijkstra said years ago:

> "Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence."

However there is an alternative approach, the formal synthesis design. Synthesis is the process of generating the description of a system in terms of related lower-level components from some high-level description of the expected behavior. With this approach we specify and design in contrast to specify-design-verify-refine [Mar18]. The synthesised

controller is formally correct-by-construction. In theory we would not need functional tests any more. Practically it is good to test the behavior, to make sure that the high-level specifications match the expected behavior.

However, there is a catch. The way this controller works, is through finite state abstractions of the system, subsequently performing some complex verification algorithms to calculate the right input, in the right state. Like many other engineering problems, it suffers from the state-space-explosion problem. This means that the complexity of the problem grows exponentially in the number of state variables. Traditional CPUs are not designed for this type of problems. The sequential mode of such computing platforms is a huge drawback in terms of efficiency. Instead, parallel computing platforms can mitigate the effect of the state-space explosion. If a task can be parallelized, then with corresponding computing resources, real-time computing can be done. If an autonomous vehicle could use this type of controller, it would be a formally correct system without any bad behavior. But these computing platforms are not well suited for vehicles. Firstly, they are expensive, and secondly, they are not well tested in the automotive industry. Instead there is another place today, that provides us with all computing resources that we need. The cloud.

1.2 Problem description and goal

According to [SB18] the term cloud computing refers to the serving of IT services, applications and data over a network, abstracting away the complex underlying infrastructure. It is also stated that cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources. The cloud provides the user with three types of services:

1. Software as a Service: Provides application access to different users over the network.
2. Platform as a Service: Provides application servers that vary in hardware configuration needs.
3. Infrastructure as a Service: Provides direct access to virtualized or containerized hardware.

With the services of the cloud every user can request a computing platform that meets his requirements. The problem of computing the formal synthesised controller actions has now a solution. We can do the expensive computations in the cloud, using Infrastructure as a Service.

The software prerequisite of this work is pFaces, an acceleration ecosystem for symbolic control[KZ], that runs in the cloud a symbolic control algorithm. The software pFaces is introduced in section 2.6.6 to provide the reader with the foundation that is required to follow along with the thesis.

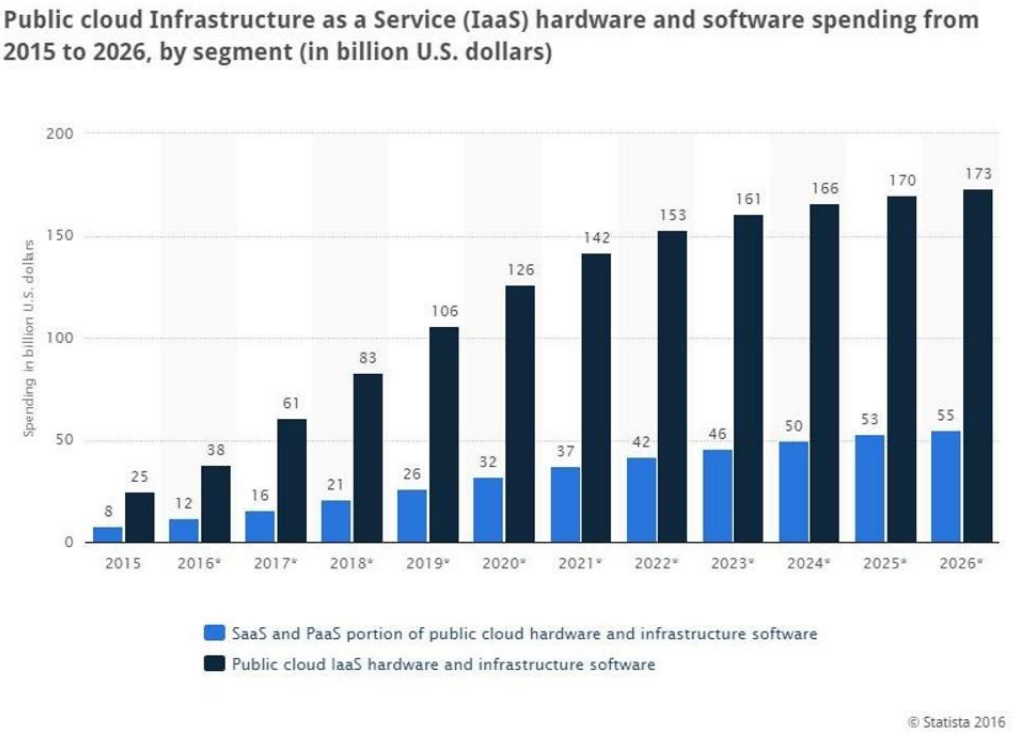


Figure 1.3: Cloud spending from 2015 to 2026 [Mic]

Also, we need an imaginary autonomous car , that has network capabilities and an user, who wants to use a formally correct controller for his autonomous vehicle. We will limit the possible use cases of the autonomous car to the highway, for convenience.

Goal of this masterthesis is to define a communication protocol standard that aims the reliable, safe and secure communication between the autonomous vehicle and the formal synthesised controller in the cloud. With this communication protocol, everyone could interact with a controller in the cloud. Remote control could be standardized. Testing modes will be included in the protocol itself, for rapid prototyping and integration purposes. This will hopefully have an impact to the future of autonomous driving. This masterthesis will provide a unique standard for that purpose. The protocol will be called: Open Automotive Control Protocol (OACP).

To be precise, the following list includes all the goals of this masterthesis:

- Conception and implementation of the required prerequisites, in order fulfill the registration of the vehicle in the controlling platform, to perform valid control actions, like registration, model generation and security.
- Design of a specification for the communication between a remote controller and an autonomous vehicle.

- Conception and implementation of a backend system that implements the OACP.
- Conception and implementation of the required interface for an autonomous car, so that it can use the remote control actions.
- Simulation and tests of the formal controller and the protocol in a simulation environment.

1.3 Structure of the thesis

In chapter 2, we will cover all the basic building blocks needed for the work context. First we will cover the classical vehicle control loop. Then we will analyze the state of the art driver assistance systems and the different modes of automation in today's technology. In this section we will clarify the affiliation of the formal synthesised remote controller - vehicle system.

Afterwards that, we will introduce a mathematical model of a vehicle, that is going to play an important role in the controller synthesis. The symbolic control algorithm works with mathematical models, so it is important to gain a deep understanding of this topic. In addition we will analyze cost functions, and their role in the synthesis process.

Next, we will present the formal synthesis algorithm. This is important, because the function and the algorithm are not wide known in academia and industry. In order to gain added value of something, there is a huge requirement that needs to be satisfied: This something has to be used. The user of this system have to trust it. According to Mayer et al[MDS95]:

> "...trust [. . .] is the willingness of a party [trustor] to be vulnerable to the actions of another party [trustee] based on the expectation that the other will perform a particular action important to the trustor, irrespective of the ability to monitor or control that other party."

To improve the acceptance and trust from this stakeholders we need to focus on the mathematical background of the algorithm, so that the users have an understanding of the functionality of the algorithm.

After the introduction to the formal synthesis algorithm we will cover the functionality of pFaces and the the role of it in this thesis.

We will then focus on network communications and we will provide some requirements for the protocol that is going to be written. Finally, we will introduce some autonomous vehicle simulators, and we will choose one that should be used in future testing for our software.

In chapter 3, we will start with an abstract view of the environment. We will design data flow diagrams and make a component view, with all the interfaces and the stakeholders for this problem. This aims to provide the user with a general image of the software architecture.

After the general introduction, we will start with the design registration of an user and a vehicle. We will analyse security and usability issues. After the design we will implement a prototype and we will run it in in the cloud.

After the implementation of the registration and the testing, we will begin with the design of our OACP. We will start with basic requirements that the protocol should fulfill. Also, a classification in the OSI Protocol Layer will be made.

Next, some advanced features and exceptions are described. At the end, the data stream during the protocol including all the details is explained.

Last, a backend system is described that serves the controller service, and a concept of a cloud-based control manager is developed. A similar system for the vehicle side is implemented.

Finally, in chapter 4, a conclusion is formed from the previous. We will also spot improvements and refinements of the whole system.

2

Research and related work

In this chapter we will explain various topics and concepts that are needed for a further understanding. We will also cover some correlated papers and the state of the art synthesis algorithm. The chapter ends with an introduction to vehicle simulators, a comparison of them and an idea for future testing purposes.

2.1 Classical control loop in vehicles

The classical hybrid system driver-vehicle-street can be modelled as a cybernetic control loop with the following components [Dil, Dip]:

- Driver (Controller)
- Vehicle (Plant)
- Street, environment (Set value)

In figure 2.1 we can see the higly abstracted control system. The driver is acting as the composition of the controller and the sensors in the control loop. He acts based on the difference of the current and the set-value he wants to have and chooses the right action to handle the situation. He controls the system by applying longitudinal and transverse control inputs. The disturbances can affect the driver (e.g. fog) or the vehicle itself (e.g. crosswind).

2.2 Driver assistance and automation in automotive

Advanced driver assistance systems (ADAS) are a very popular topic in the automotive industry. ADAS are widely used nowadays and automotive companies make big investments to innovate and to develop new and more reliable, comfortable and safe driver assistance systems. Reichart and Haller gave the following definition for ADAS in 1995 [RH, W.]:

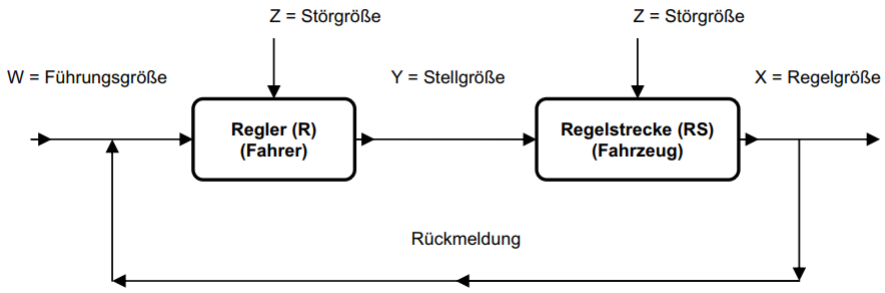


Figure 2.1: The classical control loop in vehicles

> "An advanced driver assistance system should help the driver to accomplish the task (or subtasks) of driving, according to his rules based on informations, interactions or autonomous behaviour".

An example of an ADAS is the Advanced control cruise (ACC) system which keeps a predefined distance to the front vehicle and if it is absent, the system keeps a constant speed.

ADAS have a wide range of applications. In the early days of ADAS, they improved the comfort of the driver. While more and more vehicles were produced and consumed and the traffic increased, ADAS helped with the safety during driving. Beside these two potential benefits, ADAS can improve efficiency of the fuel consumption. With complex algorithms, predictive models and landscape data, ADAS can reduce the braking and acceleration scenarios during a trip and therefore minimising fuel consumption and emissions [SZL16].

Beside these gains in everyday mobility problems, ADAS have some drawbacks. In order to achieve their desired behavior, they rely on complex sensors, big data amounts and complex algorithms. The development of such systems is not an easy task. Most often a divide & conquer approach is taken, where the project is split into smaller projects. OEMs typically outsource the development and testing to engineering companies in order to reduce costs and focus on the system requirements. Because they focus on the short-term cost of goods and want fast development times, the modifiability and reusability of the software is suffering. This increases again the costs, as a big part of the ADAS has to be redeveloped.

Another drawback of the increasing demand on ADAS comes with the fact that sensors for automobiles are often integrated with a small processor, in order to minimize data send via bus and to compute usefull information that the ECU needs. These integrated sensors are flexible but have some disadvantages, as [JSRG] analyses:

- Possible sensor duplication because of different data processing algorithms

- Unpredictable latencies because of the computing part
- Increased overhead for system integration and testing
- Inflexibility combining sensor and data processing algorithms

Moreover, they increase development time because of the overhead related to analyse and understand the datasheet.

2.2.1 Modes of automation

In order to classify ADAS regarding the automation state, there have been defined 6 stages of automation [Ver].

Stage 0 The driver is controlling the vehicle without any help from the system in stage 0. However the system could warn the driver in case of an emergency, without taking control of subfunctions.

Stage 1 In stage 1, the system can control one of the two main control inputs to the vehicle: The longitudinal or the transverse control. The driver has to constantly control the other control input.

Stage 2 Stage 2 automation systems can control both, the longitudinal and transverse control, but only in specific circumstances. During the automated driving mode, the driver has to control the actions from the system and he should be ready to take action if the system fails.

Stage 3 Stage 3 is the same as stage 2, except that the system recognises the limits of the automated control, and gives the driver feedback regarding the driving. The driver doesn't have to control the system behavior, but should take be able to take action of the system in a predefined time.

Stage 4 In stage 4, the driver can give the vehicle the full responsibility of the driving act in a predefined scenario.

Stage 5 Stage 5 automated systems have the capability to drive autonomously, in every situation. Stage 5 ADAS can be reconsidered as driverless vehicles.

We will attach stage 4 of automation for the autonomous system that is getting control from a formal controller.

2.3 Mathematical background

In this section we will give a brief introduction to the mathematics behind the model of a vehicle. In addition, we will discuss cost functions and decision theory. Both parts are needed for the conception part of the protocol implementation and for a deep understanding of the controller functionality.

2.4 Vehicle Models

In order to successfully control the vehicle, first one has to gain a deep understanding of the dynamics. There exist different models for different use cases. We will discuss and analyze vehicle models based on the paper "CommonRoad: Composable Benchmarks for Motion Planning on Roads", written by Matthias Althoff, Markus Koschi and Stefanie Manzinger at the TUM [AKM].

The mathematical model of a vehicle is a prerequisite for the formal synthesis algorithm, in order to compute the right inputs in the right states and get the desired output. However we need to understand that there will always be errors and uncertainties regarding the vehicle models, and the models of complex systems in general. We can choose the abstraction which we want to get from the model. There exist different models, as like discussed in [AKM]. In order to not blow the content up we will focus on two models from the paper [AKM] The models we will discuss are the following:

- Point-mass model
- Kinematic single-track model

We will not discuss the single-track model, or the multi-body model.

2.4.1 Point-mass model

The simplest vehicle model we can choose from, is the point-mass model. The point-mass model abstracts away the details of the vehicle, and gives us a point in space, that can be accelerated and deccelerated in bounds. The dynamics of the model are very basic, and give us a very rough approximation of the real vehicle. The model also ignores that vehicles have a minimum turning cycle.

Dynamics of the point-mass model

The dynamics of the point-mass model are given by the equation

$$\ddot{s_x} = \alpha_x$$

$$\ddot{s_y} = \alpha_y$$

$$\sqrt{\alpha_x^2 + \alpha_x^2} \leq \alpha_{max}$$

where $s_{x,y}$ is the discplacement in x or y direction, $\alpha_{x,y}$ is the acceleration in x or y direction respectively.

State-space model

After choosing $x_1 = s_x$, $x_2 = s_y$, $x_3 = \dot{s_x}$, $x_4 = \dot{s_y}$, $u_1 = \alpha_x$, $u_2 = \alpha_y$ we get the dynamics

$$\dot{x} = Ax + Bu$$

with

$$A = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

and

$$B = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

The only constraint is $\sqrt{u_1^2 + u_2^2} \leq \alpha_{max}$.

Parameters

The only parameter for the model is α_{max}.

2.4.2 Kinematic single-track model

The kinematic single-track model, is abstracting away the geometry of the front and rear wheels. This model assumes that the vehicle has only two wheels, one in front and one in the rear. Roll dynamics are not considered. Also any tire slip is not considered, meaning that the velocity is always aligned with the link between the two wheels. This model is often used in motion planning projects like in [PCY+16].

Dynamics of the kinematic single-track model

In order to define the model we will need additional parameters and variables. We will introduce now:

- The steering angle velocity: v_δ
- The steering angle: δ
- The heading: Ψ
- The wheelbase: l_{wb}
- The velocity above which the engine power is not sufficient to cause wheel slip: v_S

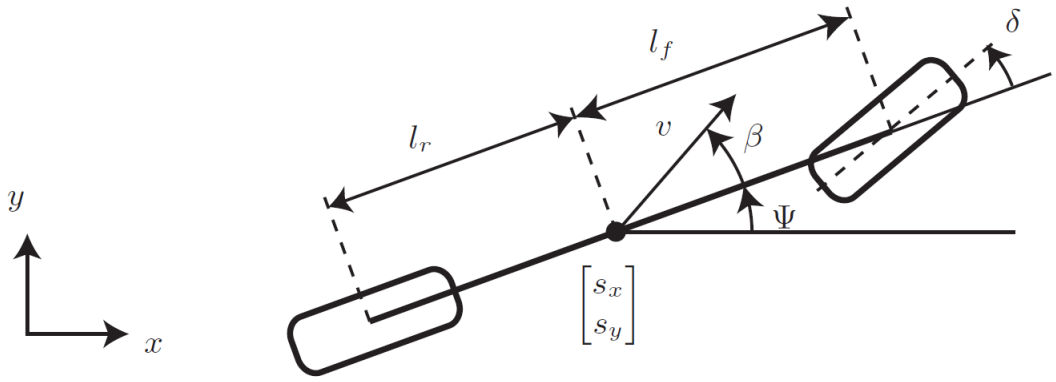


Figure 2.2: Single-track model, visualisation of the parameter from [AKM]

The dynamics are given by the following formulas:

$$\dot{\delta} = v_\delta$$

$$\dot{\Psi} = \frac{v}{l_{wb}} tan\delta$$

$$\dot{v} = \alpha_{longitude}$$

$$\dot{s_x} = v\cos(\Psi)$$

$$\dot{s_y} = v\sin(\Psi)$$

where $v_\delta \in [-v_{\delta,max}, v_{\delta,max}]$, $\delta \in [-\delta_{max}, \delta_{max}]$, $v \in [v_{min}, v_{max}]$, $\alpha_{longitude} \in [-\alpha_{max}, \alpha_{max,i}]$, where $\alpha_{max,i} = \alpha_{max}^{\frac{v_S}{v}}$ for $v > v_S$ and $\alpha_{max,i} = \alpha_{max}$ otherwise. Also $\sqrt{\alpha_{longitude}^2 + (v\dot{\Psi})^2} \leq \alpha_{max}$, where $\alpha_{latitude} = v\dot{\Psi}$.

State-space model

To derive the state-space model we define:$x_1 = s_x$, $x_2 = s_y$, $x_3 = \delta$, $x_4 = v$, $x_5 = \Psi$. The input variables are $u_1 = v_\delta$ and $u_2 = \alpha_{longitude}$. By insterting the variables in the dynamic equations we get the following system:

$$\dot{x_1} = x_4\cos(x_5)$$

$$\dot{x_2} = x_4\sin(x_5)$$

$$\dot{x_3} = \begin{cases} 0 & if(x_3 \leq \delta \wedge u_1 \leq 0) \vee (x_3 \geq \delta_{max} \wedge u_1 \geq 0)(E1) \\ v_{\delta,min} & if \neg E1 \wedge u_1 \leq u_{\delta,min} \\ v_{\delta,max} & if \neg E1 \wedge u_1 \geq u_{\delta,max} \\ u_1 & otherwise \end{cases}$$

$$\dot{x_4} = \begin{cases} 0 & if(x_4 \leq v_{min} \wedge u_2 \leq 0) \vee (x_4 \geq v_{max} \wedge u_2 \geq 0)(E2) \\ \alpha_{min} & if \neg E2 \wedge u_2 \leq \alpha min \\ \alpha_{max} & if \neg E2 \wedge u_2 \geq \alpha max \\ u_2 & otherwise \end{cases}$$

$$\dot{x_5} = \frac{x_4}{l_{wb}} \tan x_3.$$

The constraints can be converted into state-space form by exchanging the variables from the dynamics.

Parameters

The kinematic single-track model has three model parameters and eight constraint parameters. The three vehicle parameters are:

- Vehicle length (l)
- Vehicle width (w)
- Wheelbase (l_{wb})

The constraint parameters are:

- Minimum steering angle (δ_{min})
- Maximum steering angle (δ_{max})
- Minimum steering velocity ($v_{\delta,min}$)
- Maximum steering velocity ($v_{\delta,max}$)
- Minimum velocity (v_{min})
- Maximum velocity (v_{max})
- Switching velocity (v_S)
- Maximum acceleration (α_{max})

2.5 Cost functions and motion planning

In order to achieve a high configurability of the system, we need some sort of cost functions as middleware between the cloud-controller and the vehicle, that filters and responds with the optimal control inputs, in the right mode. Since the controller provides us all possible input sequences that can bring the system from state a to a state b in a predefined time-span, we have to choose the path that matches our expectations of the motion planning problem. Because of the modularity and the indepence of the algorithm

ID	Function	Optimization criterion
0	$J_A = \int_{t_0}^{t_f} \alpha^2 dt$	Acceleration
1	$J_J = \int_{t_0}^{t_f} \dot{\alpha}^2 dt$	Jerk
2	$J_{SA} = \int_{t_0}^{t_f} \delta^2 dt$	Steering angle
3	$J_{SR} = \int_{t_0}^{t_f} v_\delta^2 dt$	Steering velocity
4	$J_E = \int_{t_0}^{t_f} P(x,u) dt$	Engine power
5	$J_Y = \int_{t_0}^{t_f} \dot{\Psi}^2 dt$	Yaw rate
6	$J_{LC} = \int_{t_0}^{t_f} d(t)^2 dt$	Lane center offset
7	$J_V = \int_{t_0}^{t_f} (v_{des}(x(t)) - v(t))^2 dt$	Velocity offset
9	$J_O = \int_{t_0}^{t_f} (\theta_{des}(x(t)) - \theta(t))^2 dt$	Orientation offset
9	$J_D = \int_{t_0}^{t_f} max(\xi_1, ..., \xi_n) dt$	Distance to obstacles
10	$J_L = \int_{t_0}^{t_f} v dt$	Path length
11	$J_{ID} = \frac{1.0}{\tau}$	Inverse duration

Table 2.1: Overview of cost functions as discussed in [AKM]

itself, the cost function middleware can be created and configured very flexible based on the needs of the driver. We will first introduce the idea of cost functions as a highly abstracted concept and also introduce standardised cost functions for the vehicle motion planning problem. After that we will discuss the motion planning problem in a mathematical fashion and make a connection between the problem and the associated cost function. Most of the following information is taken from the paper CommonRoad: Composable Benchmarks for Motion Planning on Roads by Stefanie Manzinger, Markus Koschi and Matthias Althoff at the TUM [AKM].

2.5.1 Cost functions

A cost function is a function $J : D \to \mathbb{R}$, where D is a subset of a multi-crossproduct of the different parameter domains e.g. $D \subseteq A \times B \times C..$, where $A, B, C...$ are the different domains of interest. The cost function describes the associated cost of taking specific actions of the possible variables in the domain over a time period $[t_0, t_f[$ regarding that cost function. The input sequence with the lowest cost is the optimal sequence in order to reach the final set regarding the topic of interest that the cost function describes. In order to compare results of an algorithm or a function, we need cost functions for different domains. Cost functions can be also combined with a weight factor w to pick the best solution for a specific profile or configuration. Thus, by applying different weights and not changing the function map, we can create different profiles based on just the partial sum of the cost functions.

Following cost functions have been taken from the paper and will be presented in

2.1. As observed, there exist plenty of cost functions and we can create new ones by taking the linear combinations of these. For example, let's say, that we want to minimise

acceleration, meaning that we want a really smooth trip, and also that we want to have a large distance to obstacles, because we feel more safe with that. Also consider that acceleration is more important to us than the distance. With example parameters $w_A = 0.7$ and $w_D = 0.3$, we would get our personal cost function: $J_{opt} = w_A J_A + w_D J_D = 0.7 \int_{t_0}^{t_f} \alpha^2 dt + 0.3 \int_{t_0}^{t_f} max(\xi_1, ..., \xi_n) dt$, that we now have to minimise with an optimal path.

2.5.2 Motion planning

The motion planning problem is a very prominent problem in engineering. We will use the definition of the book "Spatial Representation and Motion Planning" by Angel Pasqual del PobilMiguel Angel Serna [PS95]:

> "Given a robot B and an environment E occupied by a set of obstacles C, find a motion for B among the obstacles in C that fulfills certain given conditions and is optimal in a certain sense."

Since this is a very abstract definition of the problem, we can apply the definition to various specific problems. In [PS95], following taxonomies are presented:

- Dimension of real space
- Complexity of the environment
- Physical nature of the robot
- Nature and dimensions of configuration space
- Object representation
- Motion omptimisation
- Motion conditions

Another definition found in [HHL19] is:

> "Given an initial configuration q_{init} and a goal configuration q_{goal}, find a path in the free space F between q_{init} and q_{goal}."

The two definitions may apper, but have a significant difference: The first definition, seeks to find a motion - meaning a trajectory - that satisfies the dynamics of the system. The second definition, searches for a path in the free space between the initial and the final configuration of the system.

Path planning is the pure planning of the waypoints that the system needs to reach in a specific order until the final state. There are no time considerations when analysing paths. This problem can be solved with search algorithms, like Djikstra.

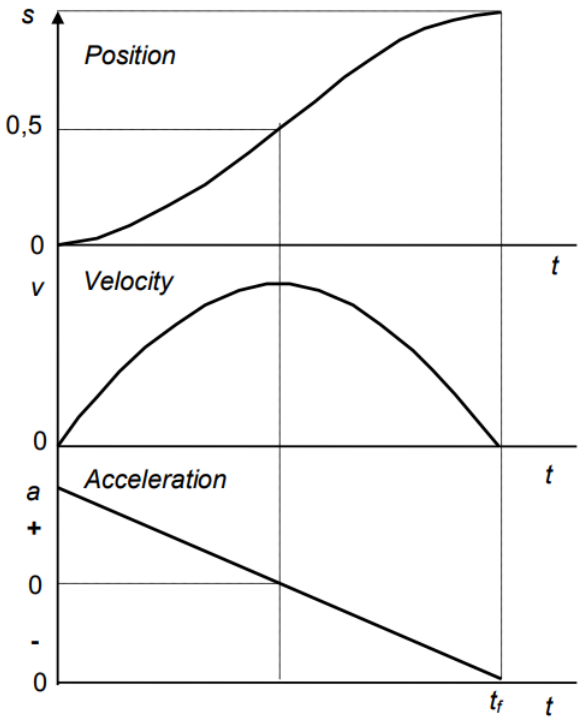


Figure 2.3: A simple motion diagram like shown in [Ton]

Motion planning, considers the time when planning the motion. This means that we can make assumptions on the velocity, the acceleration, the jerk and other derivatives of the position. Usually a spline, meaning a curve composed from several polynomials, is used to describe the systems motion. The graphical form of motion description is called motion diagram[Ton]. A very basic motion diagram is shown in figure 2.3. As we see, the problem is not limited to autonomous vehicles but rather general. The same problem arises with exploration robots, or drones. However the underlying configuration space, the dimension of real space, the motion optimization criteria and various other parameters to the problem change. We will now introduce our definition, which is based on [AKM]:

Given a dynamical system of the form $\dot{x}(t) = f(x(t), u(t))$ with an initial position in the state-space $x(t_0) = x_0$, where $x \in \mathbb{R}^n$ is the state-vector and $u \in \mathbb{R}^m$ is the input-vector. Also, a goal region $G \subset \mathbb{R}^n$, a cost function J and constraints on the system of the form $g_S(x(t), u(t), u) \leq 0$ are given. Find the input $u(t)$ for $t \in [t_0, t_f]$ s. t. $x(t_f) \in G$, while considering:

- $\dot{x}(t) = f(x(t), u(t))$
- $x(t) \in W_{S,free}(t)$
- $u^*(.) = arg\min_{u(.)} J(x(t), u(t), t_0, t_f)$

- $g_S(x(t), u(t), t) \leq 0$
- $O(x(t)) \in W_{S,free}(t)\ \forall t \in [t_0, t_f]$, where $W_{S,free}(t) \subset \mathbb{R}^2$.

The problem is solved when an input function is found satisfying all criteria. The collision with obstacles is avoided, meaning that the state of the system doesn't collide with states of other systems in the relevant state dimensions, as indicated in the last criterium.

2.6 Formal synthesis algorithm for embedded systems

The main topic of this masterthesis is to design and to implement a communication protocol for the communication between an autonomous vehicle and a formal synthesis algorithm running in the cloud. In order to understand the implications to the protocol itself and to gain system knowledge we will focus in this chapter in the math and the concept behind the formal synthesis algorithm. Most of the definitions are based on the lecture "Formal Synthesis of Embedded Systems" at the TUM.

2.6.1 System modeling

A system defines and represents the functional behavior of a technical or non-technical dynamical phenomenon, and the dynamic processes inside. The system model represents these processes in a mathematical fashion. The system is modelled as a closed environment, which can communicate with the extern environment with input signals and output signals. States exist inside the system, which capture the information of the past and save history data in an optimal manner [Uni15].

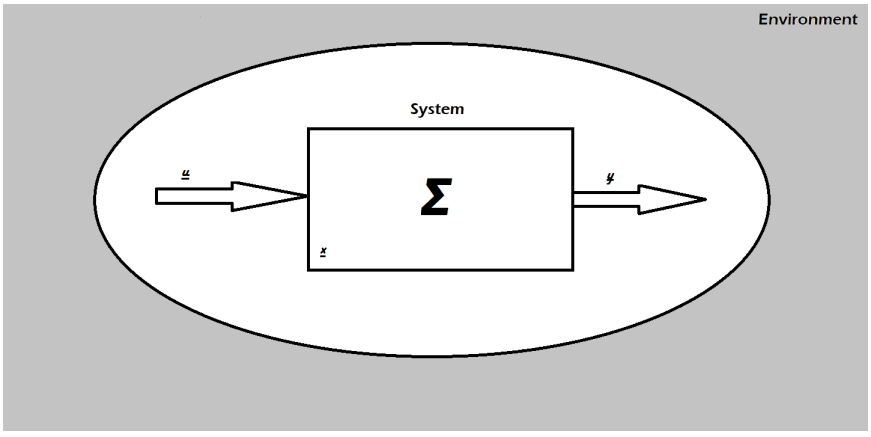


Figure 2.4: Abstracted system with inputs (u), outputs (y) and states (x)

Systems can be categorised in different fashions: We can differentiate systems based

on the time continuity and the number of the state set. Systems which evolve in continuous time are called continuous systems. On the other hand, systems that evolve discrete are called discrete systems. If a system can only have a finite number of states, it is called a finite system. If the number of states has no bound it is called infinite-state system. Hybrid systems are a combination of finite-state and infinite-state systems. [TA09].

We will consider dynamical systems of the form:

$$\left.\begin{aligned} x(t+1) &\in F((x(t), u(t)) \\ y(t) &\in H((x(t), u(t)) \end{aligned}\right\} \quad \text{System definition} \tag{2.1}$$

where x is the state vecor, u is the input vector, y is the output vector, F is the transition function and H is the output function.

In order to define a meaningful composition of this type of systems we need to define internal variables. This need stems from the fact that not all of the outputs from a system may be used as inputs to the other system. The remaining outputs can affect the state transition.

We define a system S as the Septuple $S = (X, X_0, U, V, Y, F, H)$ where:

1. X, U, V, Y are non-empty sets where
   - X is the state alphabet
   - X_0 is the initial state alphabet
   - U is the input alphabet
   - V is the internal input alphabet
   - Y is the output alphabet
2. $H : X \times U \to Y \times V$ is the output function and is strict i.e. $\forall (x, u) \in X \times U : H(x, u) \neq \emptyset$.
3. $F : X \times V \to X$ is the transition function.

Based on this definition we can categorise systems based on properties of the sets and the functions. We call a system

- Finite if X, U, V, Y are finite;
- Infinite if it is not finite;
- Autonomous if U is a singleton, i.e. $|U| = 1$;
- Basic if $U = V$ and $(y, v) \in H(x, u) \Rightarrow v = u$;
- Static if X is a singleton;

- Moore if the output does not depend on the input
- Moore with state output if $X = Y$ and $(y, v) \in H(x, u) \rightarrow y = x$

The system S is called simple if it is basic and Moore with state output. Based on these definitions a simple system S can be denoted by $S = (X, X_0, U, F)$, where the original system definition can be recovered by (X, X_0, U, U, X, F, id) where id is the identity relation.

A pair $(x, v) \in X \times V$ with $F(x, v) = \emptyset$ is called blocking.

In order to understand the system definition, we will analyze a simple continuous linear system like defined in most control-theory books and then define the corresponding system with our definition.

Example of a system Let us start with a simple SISO (Single input, single output) linear continuous system of the form $x' = Ax + Bu$ with $A = -2$, $B = 1$ and $x_0 = 0$. The output should be the state itself i.e. $y = x$. Since the system is described by a differential equation, we can solve the equation with a variable input and get a function $x(t)$, which describes the system's behaviour for an input. The solution to this differential equation is:

$$x(t) = e^{At}x_0 + \int_0^t e^{A(t-\tau)}bu(\tau)d\tau$$

where $t_0 = 0$. Let us consider that the input is a constant function with $u(t) = 1$ and $x_0 = 0$. Then the solution becomes: $x(t) = -0.5e^{-2t} + 0.5$.

In order to discretise the system and apply our definition of a system, we can use the solution of the differential equation in every time step as the transition function of the system:

$$F(x, u) = \{e^{At}x + (\int_0^\tau e^{As}ds)\, Bu\}$$

Basically, we look at the system in every time step, we set $x_{current} = x_0$ and $t = t_0$ and solve the differential equation for $t \in [t_0, t_0 + h]$, with h being the sample time. What follows, is an infinite simple system. If we compare the two output signals from the continuous and the discretised system, we understand that the discretised system gets input every sample time and it is held constant. This discretisation of the input signal is called zero-order hold (ZOH). The output signal is then sampled and held constant during the sample period.

In case of finite systems, we can draw the behavior of the systems graphically. The states are illustrated by circles. Initial states are marked with incoming arrows. Transitions from one state to another are drawn with arrows. On the top of the arrows, we notate the input, the internal input and the output that are participating in this state change. In fig 2.7 we can see a simple system with two states.

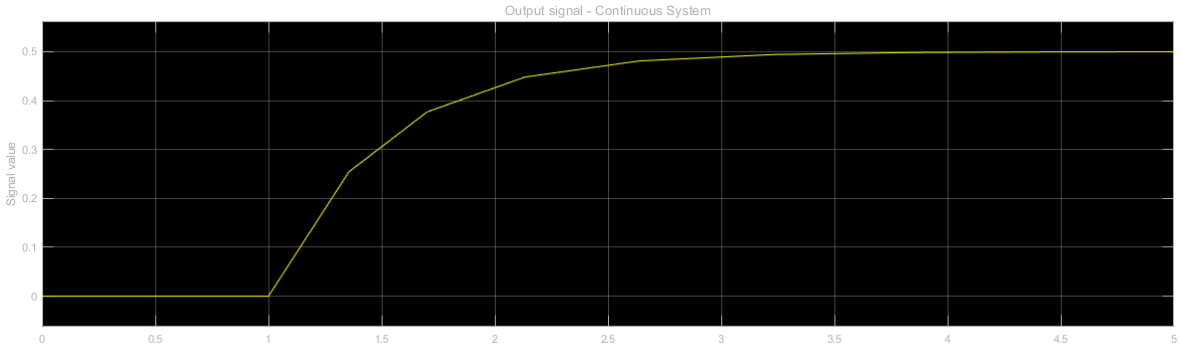


Figure 2.5: Outputsignal of the continuous system with a constant input

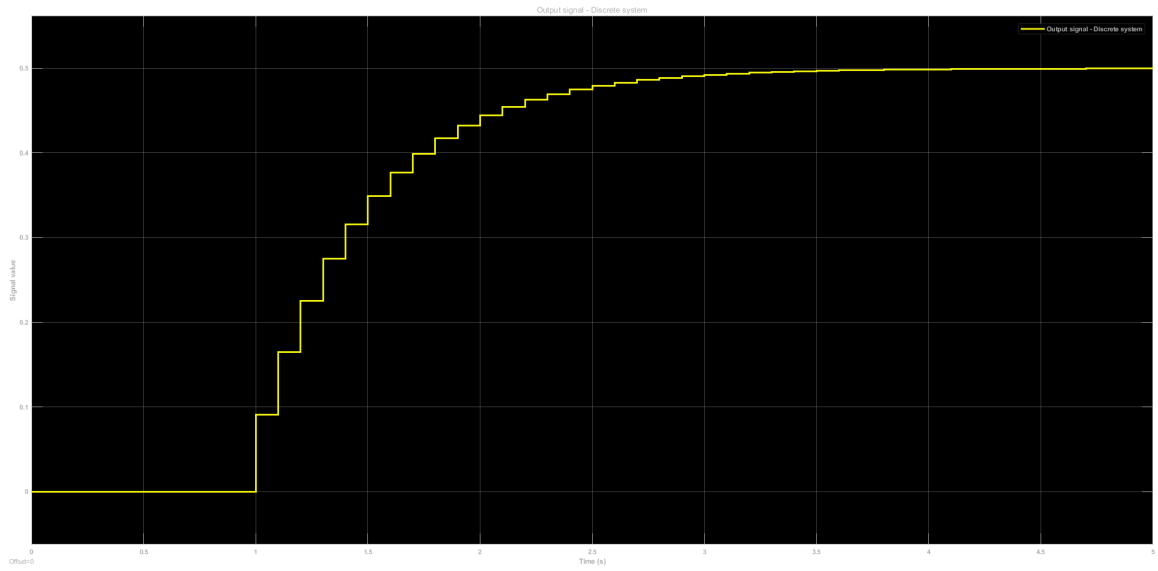


Figure 2.6: Output signal of the discrete system with a constant input

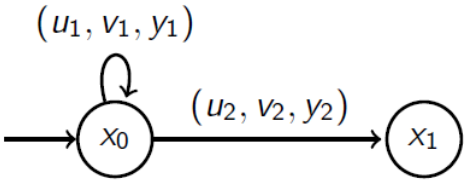


Figure 2.7: A simple state chart

2.6.2 High level specifications

After the system definition, we have to think about the behavior, we want the system to adopt. The system itself, is considered close, meaning that we can't change the behavior of the system directly by intersecting the states or the transitions of the system itself. Our actual task is to find another System C, which can be connected to the plant system, in such a manner that the overall system is behaving and fulfilling our specifications. We will start by analyzing the behavior and the solutions of a system. After that we will talk about the mathematical notation of high level specifications. Finally we will analyze the connection between these two mathematical concepts.

Solution and behavior of a system Let us consider a system

$$S = (X, X_0, U, V, Y, F, H).$$

For a fixed $T \in \mathbb{N} \cup \infty$, the quadruple $(u, v, x, y) \in (U \times V \times X \times Y)^{[0;T[}$ is a solution of S on $[0; T[$ if and only if

- $x(0) \in X_0$
- $\forall t \in [0; T-1[: x(t+1) \in F(x(t), v(t))$
- $\forall t \in [0; T[: (y(t), v(t)) \in H(x(t), u(t))$

A solution is a unique path through the states of the state machine. It doesn't matter if the last state is a blocking state or not, since we can choose the parameter T and it can be finite. For every T, there can exist different solutions.

The behavior $B(S)$ of S are all tuple sequences $(u, y)^{[0;T[}$ where:

- $\exists_{v,x,T}(u, v, x, y)$ is a solution of S
- $T < \infty \implies (x(T-1), v(T-1))$ are blocking

So the behavior of a system, consists of the sequences of the system which end in a blocking state and all the infinite sequences, of the inputs and outputs. The behavior is in an abstract way, the actual behavior of a system if we think of the system as a closed system with an interface (in- and outputs) and stress out all the possible outcomes. In a mathematical notation:

$$(u, y) \in (U \times Y)^{\infty} = \cup_{\tau \in \mathbb{N}} (U \times Y)^{[0;\tau[} \cup (U \times Y)^{\omega}$$

.

As an example let us consider the system $S_1 = (X, X_0, U, V, Y, F, H)$ where

- $X = \{x_1, x_2\}$

- $X_0 = \{x_1\}$
- $U = \{u_1, u_2, u_3\}$
- $V = U$
- $Y = \{y_1, y_2, y_3\}$
- $F(x_1, u_1) = F(x_1, u_2) = \{x_1\}$;
  $F(x_1, u_3) = F(x_2, u_1) = F(x_2, u_2) = F(x_2, u_3) = \{x_2\}$;
- $H(x_1, u_1) = H(x_1, u_2) = H(x_1, u_3) = \{y_1\}$;
  $H(x_2, u_1) = H(x_2, u_2) = \{y_2\}$;
  $H(x_2, u_3) = \{y_3\}$;

We can draw the corresponding state chart, as seen in figure 2.8: We can now, define

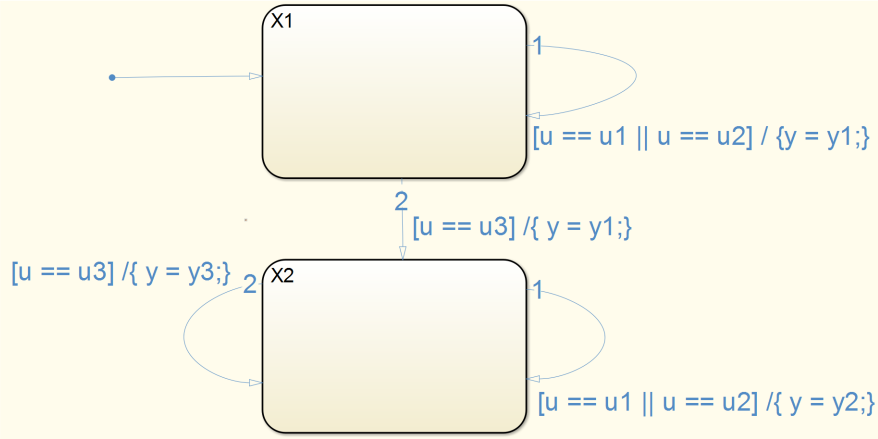


Figure 2.8: The state chart of the system S_1

the behavior of this system. To define the behavior, we use ω-regular expressions over sets. The behavior is composed of all possible tuple sequences of the in- and outputs. The behavior of the system S_1 is:

$$B(S_1) = L(((u_1, y_1)+(u_2, y_1))^{\omega}+((u_1, y_1)+(u_2, y_1))^{*}(u_3, y_1)((u_1, y_2)+(u_2, y_2)+(u_3, y_3))^{\omega})$$

For example, the infinite path $(u_1, y_1)(u_2, y_1)(u_1, y_1)(u_2, y_1)...$, defines a behavior of the system S_1.

Specifications Specifications and requirements are often written in natural language. However, a mathematical notation is needed in order to apply the formal synthesis algorithm to a plant system. Let's define specifications. Given a set Z, any subset $\Sigma \subseteq Z^\infty$ is called a specification (or property) on Z. In other words, every finite or infinite sequence of elements of the set, is called a specification.

For example let's consider the set $Z = \{a, b, c\}$. The following ,single-element sets of Z^∞ are specifications on Z:

- $w_1(t) = aaa...$ (Every time step a should hold)
- $w_2(t) = abcccababcabccabcab...$ (After an a, there should always hold b)
- $w_3(t) = c$ (In the first time step c should hold)

We can naturally express these types of properties with the help of linear temporal logic. Linear temporal logic formulas stand for properties of sequences. Linear Temporal Logic is an extension of propositional logic to formally define properties within a linear-time perspective. Temporal logic allows the specification of the relative order of events. LTL formulas are always defined on a finite set of atomic propositions. LTL formulas are built from these atomic propositions and are closed under the boolean connectivenes, the unary temporal operator $\circ$ (next-time) and the binary temporal operator U (until) [dGV13, RWR17, BK08]. The underlying time domain is discrete. The time modalities are time-abstract, meaning that time gets passed by the incremental time tick, and all implications to the S.I unit second, have to be done manually.

We can create sequences of sets consisting of atomic propositions and then check if the trace satisfies the LTL formula. In fact, let φ be a LTL formula over the atomic proposition set $AP = \{a, b\}$ and $w \in AP^{[0;\infty[}$. Figure 2.9 shows the intuitive idea behind "w satisfies φ". The set of all satisfying sequences is denoted by $P(\varphi) = \{w : [0; \infty[\to AP \mid w \models \varphi\}$.

More information regarding LTL can be found in [BK08, dGV13].

Specifications on systems, realisability Now, that we defined specifications on sets, we can finally create a connection between a system and a specification. What matters for us, is the behavior of the system because this is the interface with the system, consisting of the actuators we can control and the data we can actually measure. The behavior of a system is defined as the finite and infinite sequences of tuples from the in- and outputs, that are solutions of the system and end because a state is blocking or the sequence is infinite.

Given a system $S = (X, X_0, U, V, Y, F, H)$ and a specification R over $U \times Y$, we say that S *satisfies* R if

$$B(S) \subseteq R.$$

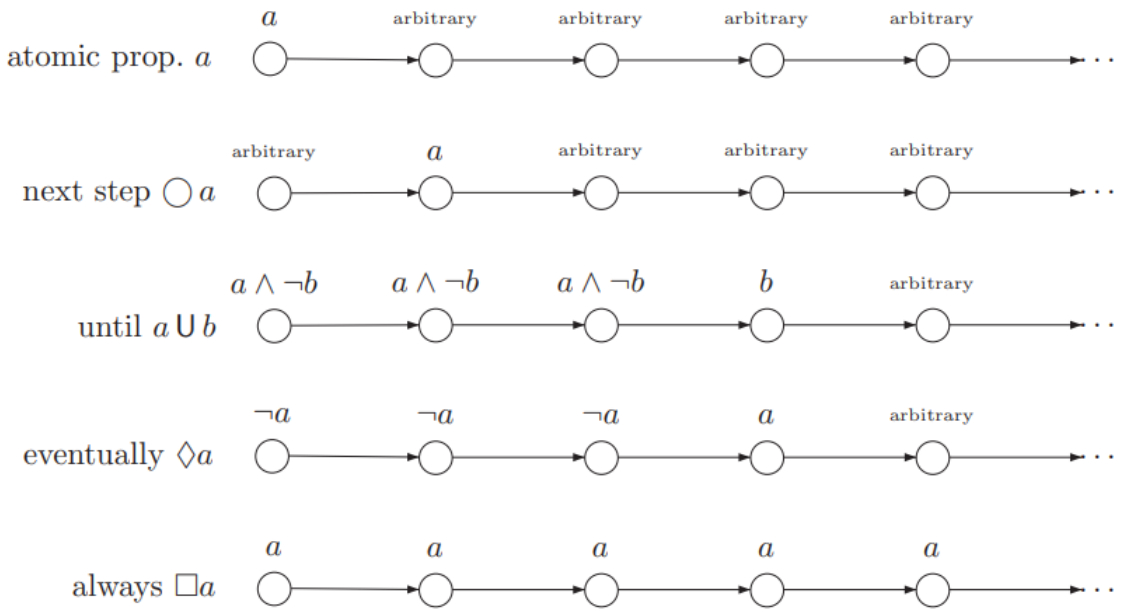


Figure 2.9: Intuition behind "w satisfies φ", from the book "Principles of model checking" [BK08]

In other words, this means that a system satisfies a requirement, if the behavior it imposes, is a subset of all possible solutions that fulfill the requirement.

On the other hand, given a specification R over $U \times Y$, we say that R is realisable on S if there exists a system C (the controller), which is feedback composable with the system S and

$$B(C \times S) \subseteq R$$

.This definition shows us, that we have to control the output of the system, by applying appropriate input in the right state.

We can naturally define specifications on a set of atomic propositions using linear temporal logic [RWR17, BK08]. Like already noted, the set of all satisfying sequences is denoted by $P(\varphi)$. To make a connection between the LTL formula φ on the set AP, and the behavior of a system S, we have to create a strict labeling function $L : U \times Y \to AP$. Every element of the behavior of the system $(u, y) \in B(S)$, induces a sequence $w \in (2^{AP})^{\infty}$, with $w(t) = L(u(t), y(t))$. $P(\varphi)$ is the set of all satisfying sequences for the LTL formula φ and a property over $(2^{AP})^{\infty}$. We can now define the property $P(\varphi)$ over $(U \times Y)^{\infty}$, with the labeling function L:

$$P_L(\varphi) = \{(u, y) \in (U \times Y)^{\infty} | L \circ (u, y) \in P(\varphi)\}.$$

We say that:

1. (u, y) satisfies φ if $(u, y) \in P_L(\varphi)$
2. S satisfies φ (under L), if S satisfies $P_L(\varphi)$
3. φ is realisable on S (under L), if $P_L(\varphi)$ is realisable on S

2.6.3 Finite abstractions of continuous systems

Since we are interested in controlling continuous systems, we will have to abstract the continuous system to a finite abstraction in order to apply symbolic control.

To understand the finite abstraction we first have to define the set of admissible inputs, as well feedback refinement relations, as defined in [RWR17].
We define the set of admissible inputs $U_S(x) = \{u \in U | F(x, u) \neq \emptyset\}$. This set of a state x contains all possible inputs that will not block the system, i.e. $F(x, u) \neq \emptyset$.

Now let S_1 and S_2 be simple systems, where the sets are indexed with the system index. Assume that $U_2 \subseteq U_1$. A *strict relation* $Q \subseteq X_1 \times X_2$ is a *feedback refinement relation* from S_1 to S_2 if following conditions hold for all $(x_1, x_2) \in Q$:

1. $U_{S_2}(x_2) \subseteq U_{S_1}(x_1)$
2. $u \in U_{S_2}(x_2) \rightarrow Q(F_1(x_1, u)) \subseteq F_2(x_2, u)$.

A feedback refinement relation associates states of systems with states of other systems. Additionally the FRR imposes requirements on the local dynamics of the associated states [RWR17]. More information can be found in [RWR17].

Consider a continuous system of the form

$$\Sigma : \dot{x}(t) = f(x(t), u),$$

where $x(t) \in X \subseteq \mathbb{R}^n$ is the state vector and $u \in U \subseteq \mathbb{R}^m$ is an input vector. We partition the set X into a finite partition $\overline{X}$, that is constructed by a set of hyper-rectangles of identical widths $\eta \in {\mathbb{R}_+}^n$. The component wise widths don't have to be same, i.e. $\eta \in {\mathbb{R}_+}^n$. We also define $\overline{U}$ as a finite subset of U, i.e. $\overline{U} \subset U$. Note the difference between $\overline{X}$ and $\overline{U}$, since $\overline{X}$ is a set containing sets, and $\overline{U}$ is a set containing vectors.
We define $x_{x,u}(.)$ as the trajectory satisfying the differential equation at almost every $t \in [0, \tau[$, where $\tau \in \mathbb{R}_+$ is the sampling period, with the initial condition $x_{x,u}(0) = x$ and the input u being constant. A finite abstraction of the system Σ, is then defined as a finite-state system $\overline{\Sigma} = (\overline{X}, \overline{U}, T)$, where $T \in \overline{X} \times \overline{U} \times \overline{X}$ is a transition relation that fulfills the fact, that there exists a feedback refinement relation $R \in X \times \overline{X}$ from Σ to $\overline{\Sigma}$ [KZ]. Since discretisation naturally introduces some errors in the transition of states, we want to bound the error by over-approximating the reachable sets starting from a set $\overline{x} \in \overline{X}$ when the input $\overline{u}$ is applied. This over approximation can be thought as a

function $\Omega^f : \overline{X} \times \overline{U} \to X^2$. An over-approximation of all the reachable sets can then be obtained by the map $\omega^f : \overline{X} \times \overline{U} \to 2^{\overline{X}}$, defined by $O^f(\overline{x}, \overline{u}) = Q \circ \Omega^f(\overline{x}, \overline{u})$, with Q being a quantization map, like defined in [KZ].

2.6.4 Symbolic controller synthesis of a provably correct controller

We are now able to define the synthesis problem: Given a system $S = (X, X_0, U, V, Y, F, H)$. Given a set Z and a specification Σ on Z. The system S satisfies the specification Σ on $U \times Y$ if $B(S) \subseteq \Sigma$. Given a specification Σ on $U \times Y$, the system C solves the control problem (S, Σ) if C is feedback composable with S and the closel loop $C \times S$ satisfies Σ [RWR17, TA09].

So the problem is, given a specification on the input and output sequences and a system, to find a controller, that can be connected to the system, which enforces this behavior that fulfills the specification.

We will introduce an algorithm for the solution of symbolic controllers, like described in the lecture "Formal synthesis of embedded systems" at the TUM. Also data from [KZ, BK08, RWR17, TA09, MPS] was used.

Given a simple system $S = (X, X_0, U, F, H)$ with the original system definition being: $S = (X, X_0, U, U, X, F, H)$. We first define the predecessor map for $Z \subseteq U \times X$ by

$$pre(Z) = \{(u, x) \in U \times X | \emptyset \neq F(x, u) \subseteq \pi_X(Z)\}.$$

The map $\pi_X(Z)$ is defined as $\pi_X(Z) = \{x \in X | \exists_{u \in U}(u, x) \in Z\}$. It is just the set with all states x of the set Z, i.e. $\pi_X(Z) \subseteq X$. The predecessor map is the set of non-blocking input/state pairs for which all successor states are in the projection of Z on X. Also notice that the predecessor map, which is defined in $pre : 2^{U \times X} \to 2^{U \times X}$, is monotone, meaning that the function takes a set and produces at least a set with the same number of elements i.e. $|Z| \leq |pre(Z)|$ and $Z \subseteq pre(Z)$.

Let us introduce a set of atomic propositions AP. We will focus on reachability specifications on simple propositional formulas. Consider a LTL formula φ over the set AP with the propositional formula ψ. We construct the winning set $Z_\psi = \{((u, x) \in U \times X | L(u, x) \models \psi\}$, where $L : U \times X \to 2^{AP}$ is a labeling function, mapping every element of $U \times X$ to a set whose elements $\alpha \in AP$, [KZ, RWR17].

We consider now the monotone function $G : 2^{U \times X} \to w^{U \times X}$ which is defined by $G(Z) = pre(Z) \cup Z_\psi$. We will compute $Z_\infty = \mu Z.G(Z)$ starting with $Z_0 = \emptyset$. We adopt the notation from $\mu-$calculus with μ as the minimal fixed point operator and Z is the operated variable. That means we first construct the winning set and starting with that, we compute the set of reachable states based on the winning set. The theorem

says that: $\exists$ finite system C so that $C \times S$ satisfies ψ under L iff $X_0 \subseteq \pi_X(\mu Z.G(Z))$. In other words, if we can reach the desired state from the initial states of the system, meaning there exists a path from X_0 to Z_ψ, the specification is realizable and there exists a controller which can enforce the behavior. Notice that every element of X_0 has to be in the final state $\mu Z.G(Z)$, since we have to make sure that the specification is realizable from all starting states because the initial state is chosen non deterministically.

The synthesised controller is a static system $C = (\{q\}, \{q\}, X, X, U, id, H_c)$ with only a dummy state q, the states X as input and U as output. The strict output map $H_c(q, x) = H_{c,state}(x) \times \{x\}$ is defined on $H_{c,state}(x) : X \to 2^U$ by

$$H_{c,state}(x) = \beta(j(x) < \infty, \{u \in U | (u, x) \in \mu^{j(x)} Z.G(Z)\}, U),$$

where β is the operator which assigns $H_{c,state}(x)$ the second argument if the first argument is true, and otherwise the third argument. The function $j(x) : X \to \mathbb{N} \cup \{\infty\}$ is given by

$$j(x) = inf\{i \in \mathbb{N} \cup \{\infty\} | x \in \pi_x(\mu^i Z.G(Z))\}.$$

The system C is then feedback composable with the system S and $C \times S$ satisfies ϕ [KZ, RWR17, RZ, TA09].

The idea behind this controller is simple. We apply the monotone function G to the plant system. The function outputs in every step the set of reachable states starting from the desired state. If the algorithm ends and the initial states are inside the final set, then there exist paths from the initial states to the desired states. To get from an initial state to the desired state, we just have to apply the appropriate input from the input-state tuple, that we found out with the fixed-point algorithm first (note $j(x)$). If we recursively apply the right input in the right state, we can get from all initial states to the desired states and thus the specification is met.

2.6.5 Practical implementation of formal techniques

At this point we have analysed the main theoretical part of the prerequisites. In this section we will focus on a practical implementation of the symbolic control algorithm, that we will use as backend server that implements the algorithm, in order to test, debug and validate our protocol.

Model-based techniques allow us to algorithmically solve the control or verification problem. However these formal techniques suffer from the state-space explosion problem. The problem with symbolic control and formal verification techniques is the computation part[KZ].

There exist implementations of symbolic control, like in [HMMS, RZ], but as noted in [KZ], these are only serial implementations that utilise only one CPU. The serial implementation of the algorithm is counter-optimal, and bounds the set of systems that

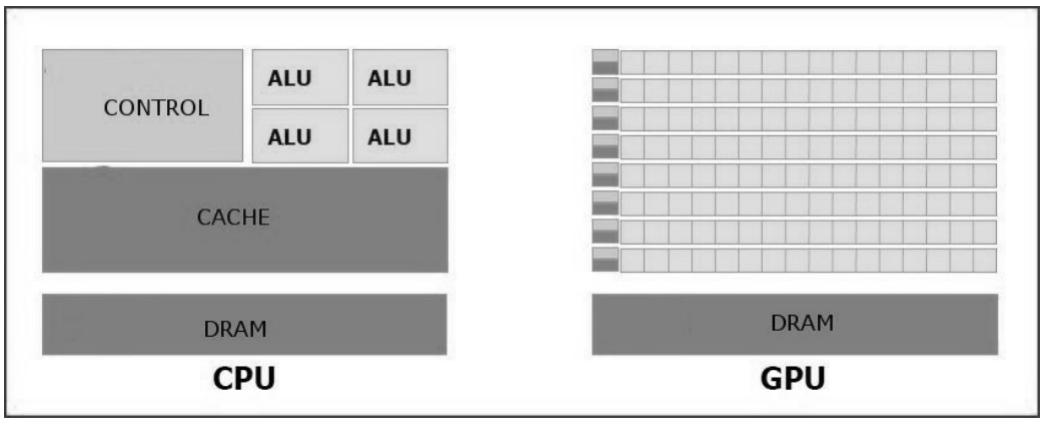


Figure 2.10: Difference between GPU and CPU, as noted in [OAD]

can be analysed with that approach to small systems with limited number of states and inputs. Also, the controllers cannot be computed online, because the serial implementations are too slow.

In the past years we have seen a stagnation in the processing speed of processing units, disobeying moores law [JK12]. CPU Vendors like Intel are trying to offset the stagnation with multi-core hardware architectures and complex circuits that should speed up the processor. However, these approaches increase the complexity of the processor. The increasing complexity of these systems can have significant security issues. In 2018, two security issues were reported that affect most commercial CPUs, Meltdown and Spectre [Mor18]. Both security issues, are correlated with the "out-of-order execution" feature of intel CPUs, which is an optimisation technique that utilizes all execution units of a CPU core at the maximum. Practically, the CPUs supporting out-of-order execution, allow running operations speculatively, with a probability that the branch is not taken and the results will be omitted. More information on these security flaws can be found in [Mor18].

In recent years another approach for achieving high throughput is getting more popular: Parallel computing. Several years ago, engineers noted the potential of using GPUs for general purpose applications. GPUs can speed up parts of application that require numerical computations. The reason for that, is that more hardware circuits are assigned in the GPU to handle computations, instead of flow control or caching, like in today's CPUs as seen in figure 2.10 [OAD].

Tasks that can be parallelized achieve a significant speedup in order of magnitudes, like in [KZ]. In order to use this type of computing platforms, several standards and programming models evolved like CUDA and OpenCL [OAD, JK12]. These platforms allow developers to write code that can run on a simple CPU, as well on complex hardwarecluster consisting of GPUs and CPUs without the need to rewrite code. Last, cloud

services providers, which support IaaS (Infrastructure as a Service), give customers the option to lease hardware systems via the internet [SB18]. With this solution, everyone can book hardware clusters and use them to run a parallel algorithm with little cost.

In [KZ], the authors present us a software-ecosystem called pFaces, that can be used for the utilization of high-performance computing (HPC) platforms. In [MM], pFaces is migrated to the cloud with Amazon Web Services (AWS). Since we will use pFaces as a generic accelerator that runs in the cloud on top of a hardware cluster in this thesis, we will briefly analyse the structure, function and capabilities of pFaces in the next section.

2.6.6 pFaces: A software ecosystem for parallel computation

pFaces, is a software platform that manages and supervises the execution of parallel algorithms on existing computing resources. This software tries to utilize all computing resources under management, in order to reduce the computation time of algorithms [KZ].

pFaces supports high performance computing (HPC) platforms. This platforms consist of an interconnection network that connects different compute nodes (CN). Each CN connects different computing units (CUs), like CPUs or GPUs. Each CN has a predefined set of processing elements (PE). PEs are the hardware circuits doing mathematical and logical computations. In order to minimize computation time, pFaces utilises all available PEs in this type of heterogeneous systems [KZ]. In figure 2.11 we can see the computing model considered in pFaces.

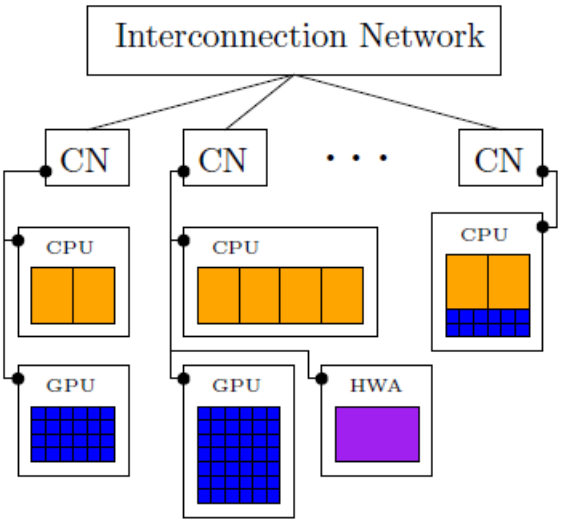


Figure 2.11: Computing model considered in pFaces [KZ]

The software is built aiming to utilise computing resources, independent from the ac-

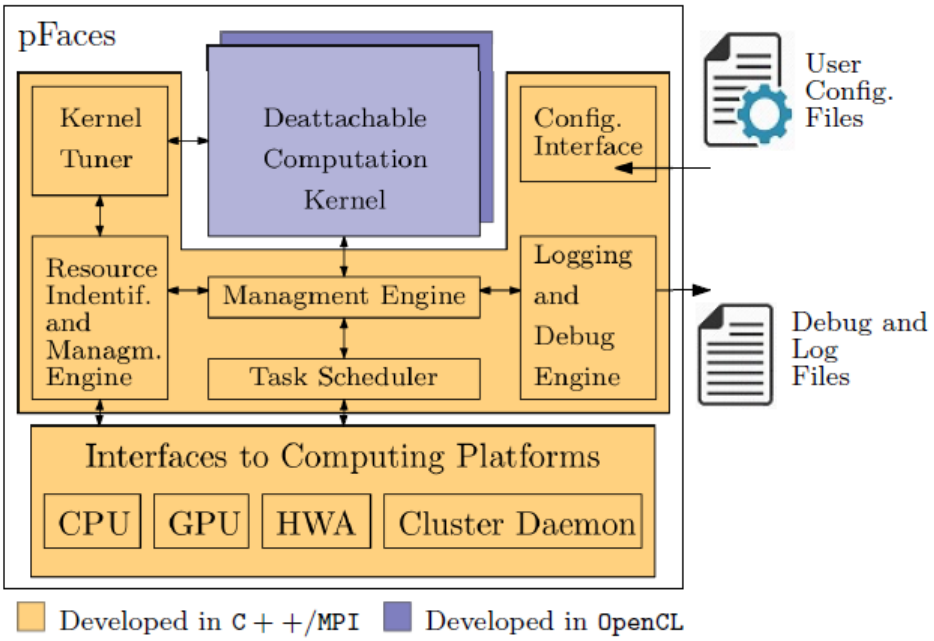


Figure 2.12: Internal structure of pFaces [KZ]

tual algorithm which is called Computation Kernel (CK). The management ecosystem is independent from the computation Kernel. Figure 2.12 shows the internal software structure of pFaces. pFaces goal is to utilize all available PEs in a heterogeneous system. Therefore pFaces manages the runtime of this computation by assigning computation jobs to PEs.

The main modules of pFaces are:

- Resource Identification and Management Engine : Identifies the underlying hardware resources and runs part of the kernel
- Kernel Tuner Module : Approximates the computing power of each CU
- Task scheduler Module : Runs the kernel as efficient as possible
- Configuration Interface Module : Gives an interface to the user via text configuration files
- Logging and Debugging Module : Provides hints, suggestions, debugging and state information about the executing kernel
- Computation Kernel : The job to be accelerated

In order to accelerate an algorithm with pFaces, the software should be written in OpenCL with some extensions defined by pFaces.

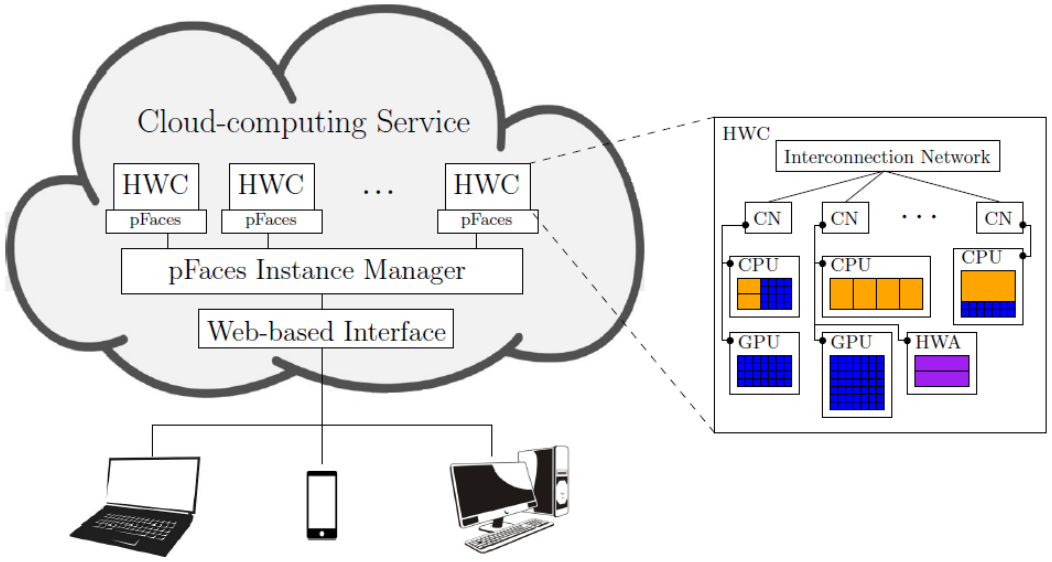


Figure 2.13: The classic approach to interface with pFaces in the cloud [MM]

For our considerations, we will interface with pFaces through a REST Api and not through the basic web-interface which is shown in figure 2.13. The components of the implemented system are discussed in chapter 3. More information on pFaces can be found in [KZ, MM].

2.7 Protocol design principles

2.8 Simulators

In this chapter we will introduce automotive simulators which can be used for testing and debugging purposes.

2.8.1 SUMO

According to the paper SUMO - Simulation of Urban Mobility: An Overview [Beh], SUMO is an open source, highly portable, microscopic and continuous road traffic simulation package designed to handle large road networks. It is being developed and maintained by employees of the Institute of Transportation Systems at the German Aerospace Center. It is open source.

SUMO is not just a traffic simulator, but a complete simulation suite with many tools and interfaces. With SUMO we can configure the simulation environment very flexible. For example we can build a simulation environment that supports light systems, bus stops and vehicles with different behavior [Joe].

The documentation has a high quality and is very readable. There is a separate user and developer documentation. The user documentation focuses on the various tools and the possibilities one has with SUMO. The developer documentation focuses on the extensibility of the simulator and code, test and debug capabilities of SUMO[Pab].

SUMO has a rich ecosystem for map generation. The tool netgenerate generates a map based on different preconfigured settings. The tool netconvert is capable of converting many digital network formats[HHH].

SUMO has two different simulation environments. There is a command for a non-graphical simulation, that is used for testing and integration purposes, and one for graphical simulation, where the vehicle and the environment are rendered.

SUMO can output information of the simulation, such as the vehicle position or traffic key values in every time step. There is also a noise emission and fuel consumption model.

Finally, SUMO has an API that can be accessed from every programming language through TCP/IP sockets. SUMO includes a Python and Java API builtin.

2.8.2 Movsim

Movsim stands for Multi-model open-source vehicular-traffic Simulator. It is a software tool that is used in evaluation and monitoring of control software of autonomous vehicles. Movsim is written in JAVA. The source code can be found in github. Because it is open-source, it can be me modified and extended with a desired behavior. The code is divided in different modules and submodules, which can be exchanged and combined in various situations. For example, the traffic simulation can output a csv file, that can be used for analysing the data. The functionality and the simulation trace can be configured with an xml configuration file. The simulator can output various information in the csv format[GBH+].

The models that are implemented are derived from the textbook Traffic Flow Dynamics by Martin Treiber and Arne Kesting[TK13].

Movsim supports many features such as:

- Multiple models of different model classes
- Physics-based model for fuel consumption and emissions based Traffic Flow Dynamics[TK13]‖
- Drivers behavioral models
- Multi-lane simulator

The documentation is very understandable and the tool is relatively easy to use. Because it is written in JAVA, the JAVA Runtime is required to run the simulator. MovSim is a great simulator for academic purposes, because the underlying models features are implemented based on scientific papers and books.

2.8.3 Carla

Carla is a simulator for autonomous vehicles. It is open source, written in C++ and very extensible. Carla runs on Linux and Windows machines. The framework provides the user with an incredible flexibility. Almost everything can be configured, like roads, sensors or the weather. Carla is also very extensible. Users can create digital assets like vehicles or buildings and use them in their simulation. This is great for simulations in different environments[DRC+].

Carla has a simulator and a powerful python api. The simulator is responsible for doing all the computations and rendering, while the API module just connects to the simulator and controls it. The Carla documentation is very readable and easy to understand. The documentation structure is intuitive, so that the user can find fast what he needs. With the API the user can spawn vehicles, control the weather, attach and detach sensors, control the vehicles and many more. Carla also got an impressive GUI.

One overlooked feature of Carla is that we can disable the rendering process. In case of automated tests or performance reasons, one can choose to disable the rendering. With this feature Carla does not render and just computes the state of the world[DRC+]. This gives a performance boost to the simulation that allows engineers to run tests and simulations very fast.

2.8.4 Comparison of simulators

In this section, we will compare the simulators based on some predefined criteria. Every criterium has got an individual weight to compensate detail criteria. Each criterium can get 1 up to 10 points. The simulator with the greatest percentage is the most sophisticated.

Carla satisfies almost every criterium. It has also a very active community and regular updates.

SUMO is a real good simulator with a flexible configuration. However, the visualisation is limited. The API is fine, but the Carla API is really powerful and intuitive.

MovSim was the light weighted simulator in our simulator comparison. The capabilities are bounded. However MovSim is a great simulator for beginners and students that want to learn more about the functionalities of simulators and the physics behind them.

Figure 2.14: Carla simulator - The information is shown left

Criteria (Weight in %) \ Simulator	SUMO	MovSim	Carla
Lightsystems (6)	8	4	7
Map and streets (9)	8	4	9
Vehicle diversity (7)	8	5	10
Fuel calculation (8)	10	10	8
Map import and generation (9)	8	5	10
Simulation information (8)	7	3	9
Visualisation (7)	6	3	9
Interaction with Api (9)	5	5	8
Documentation (10)	9	5	8
Support (8)	5	2	9
Active community (6)	6	1	10
Last update (5)	6	3	9
Additional tool support (8)	9	2	9
Total	73.9	41.3	88.3

Table 2.2: Comparison of the simulators

Based on this short analysis, we will recommend Carla for future simulation purposes.

3

Concept and implementation of a control protocol and an application for symbolic control

In this chapter we will focus on the enterprise application that we will develop. The main topic of the thesis, is to create a protocol for the remote control of autonomous vehicles over the cloud. With this protocol, we will connect a mock vehicle that is implemented using C#, to a self-implemented server and get the symbolic control from pFaces. The whole application consists of a registration system, a database, and a web frontend too.

We will call the designed protocol Open Automotive Control Protocol, and in shorthand notation OACP.

3.1 Requirement and specification analysis

In this section we will fix the requirements of the application and the parts that compose the application.

We will first analyse our imagined user experience and the steps a user should take in order to use our system.

To use the protocol, there should be a registration system where the user can register and manage all their vehicles. The user should be able to create a new vehicle with different protocol modes and download a flash extract for each vehicle and protocol mode, that will later be flashed to the ECU of a vehicle. This flash extract contains all configuration and communication settings in order for the protocol itself to function correctly.

The ECU should run a standardised AUTOSAR communication thread with a sampling

time of dtthread, where the communication with the server is handled. The communication with the server is scheduled in a discrete time manner, and the thread gets called every dtthread seconds. In the worst case, after a wcet time dtmax the optimized inputs are received. The ECU should then apply all inputs in a time discrete sequential fashion with a sampling time of dtinput.

The server should listen for incoming connections at a predefined port. It should also be able to handle multiple clients at once. The server should also validate the vehicle with the configuration saved in a database

The protocol itself should have security mechanisms built-in. There should be a standardised way to communicate with a participant. It should provide authentication and encryption schemes. It should be extensible and easy to use. Also, there should be some safety concept in order to use the protocol and make sure that the communication is safe. The protocol should have different protocol modes that differ in logging, testing, debugging and calibration authorization schemes. The protocol should also provide a way to alter the driving mode, or optimize the control inputs in some fashion. Last but not least, the state and control communication, should be unique and deterministic.

3.2 High level architecture of the enterprise application

In this section we will start the project with a high level architecture of the application, based on the requirements of section 3.1. We will create a system drawing, with all required systems and their connection. After that we will analyse the information flow of the whole system to get more requirements for the implementation.

3.2.1 Component model

The application as a whole consists of distributed systems that are connected with different protocols.

We first need a database to store the data about the user and the vehicle configurations.

Next, we need a management front-end application that takes care of the user interface. The user should download the flash extract of a created vehicle and calibrate the ECU with that configuration in order for a vehicle to successfully connect to the server.

The front-end application connects with a back-end application via a REST API over HTTP. The back-end application accepts requests from different clients with different authorization rules, and gives standardised interfaces to alter and query the database.

pFaces runs alongside the other systems, and listens on a specified TCP/IP port for incoming requests. When a client connects to pFaces, the system opens a new port for the specific symbolic control problem and delegates the communication between pFaces and the client to that port. Every control action is then requested with that endpoint.

The OACPProtocol, is implemented as a dynamiced linked library (.dll), that gives applications OACP capabilities. It provides a standardised message communication scheme, data querying of responses and encryption capabilities.

The OACP-Server is an application by itself, handling incoming vehicle control requests and referencing the OACPProtocol, in order to have a standardised communication with clients. The server is communicating with pFaces and the back-end application with HTTP. The server gets also connected with the vehicle over TCP/IP and the OACPProtocol itself.

The vehicle is an abstract client, referencing the OACPProtocol in order to use it. It can be a real physical vehicle, a robot-car, a simulation or anything else that fulfills the requirements of the protocol. In any case, the client runs a timed thread that handles the communication with the server. The inputs are then placed in an input queue, and are applied in a discrete fashion. In figure 3.1 we can see the different components of the system and the connections between them.

3.2.2 Data Flow Diagram

A Data Flow Diagram (DFD) is a graphical diagram that visualises the data flow within a system. We will create a data flow diagram to visualise the data transformations in our application.

In order to draw the data flow diagram we will discuss the steps a user has to take in order to use the protocol.

1. Create an user account
2. Create a new vehicle
3. Download the flash extract for the vehicle
4. Calibrate the ECU of the vehicle and apply the configuration
5. The vehicle connects to the server and the OACP session starts

In our architecture we got two external entities: The application user and the vehicle. We will model only one database as a central repository, that contains many tables. The DFD can be seen in figure 3.2.

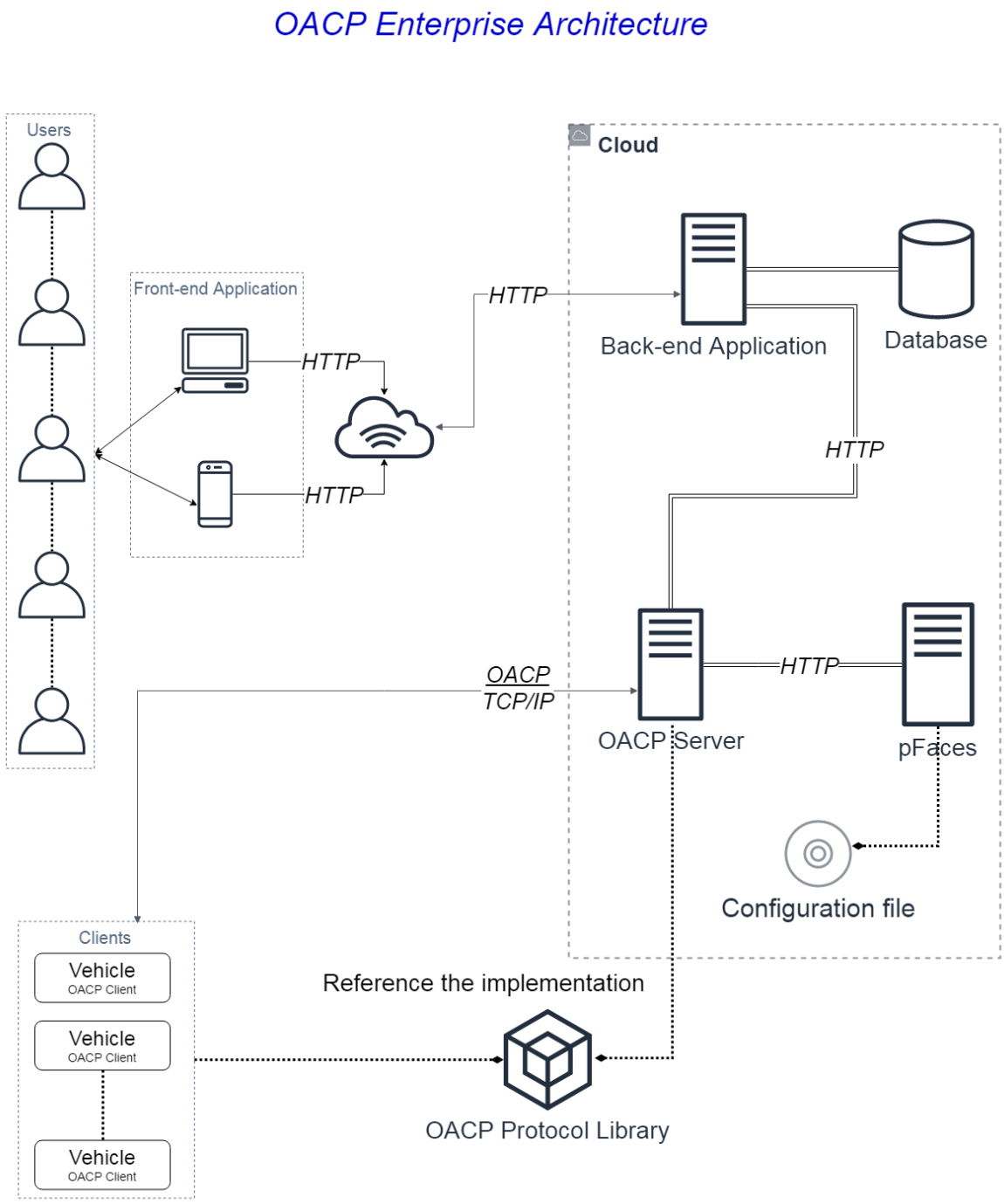


Figure 3.1: OACP Enterprise architecture

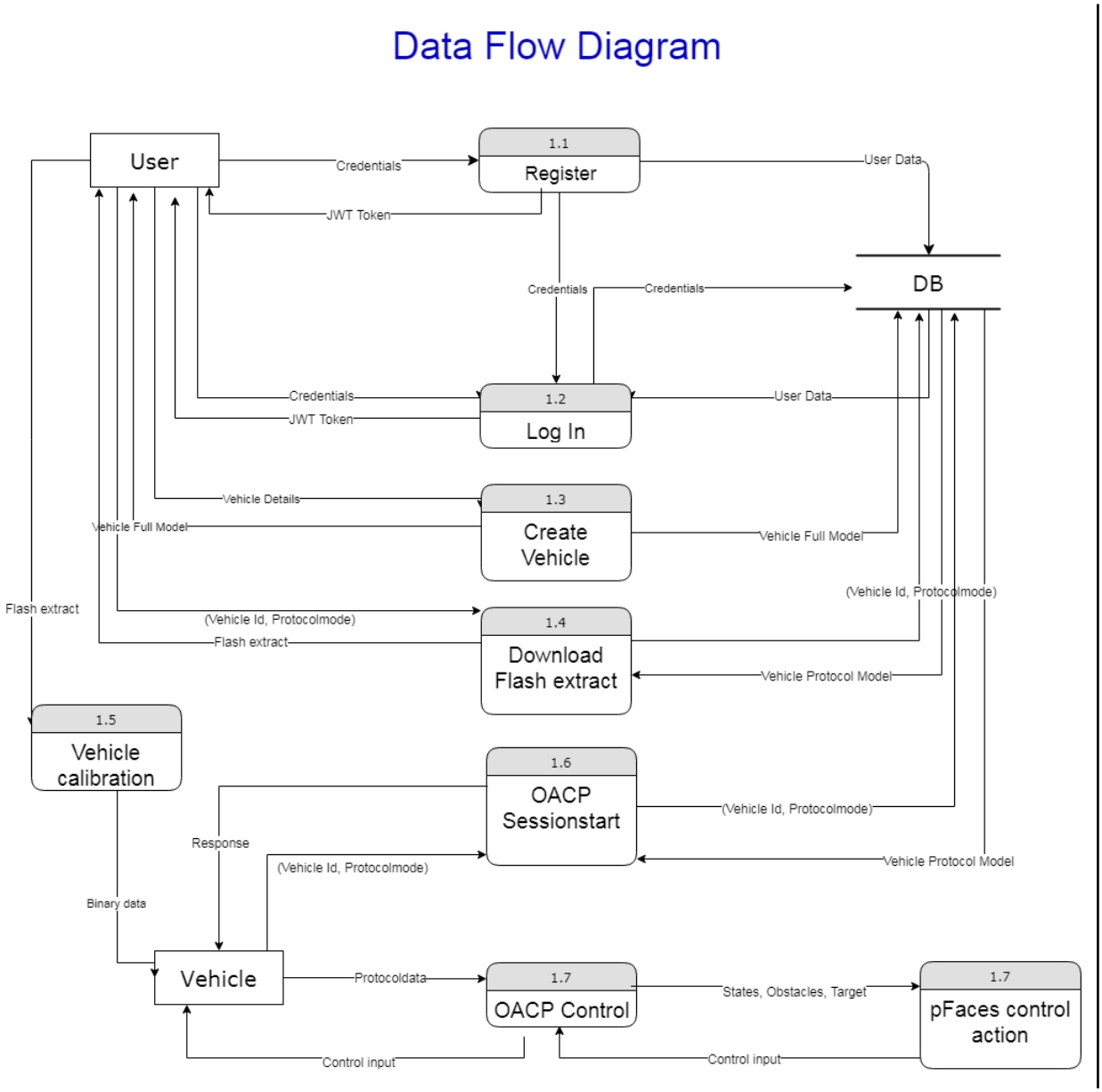


Figure 3.2: Data Flow Diagram of the application

3.3 Management system

In this section we will take a look on the registration system. The registration system is responsible for providing an interface to the user in order to register and manage the corresponding vehicle data.

We will implement the registration system using ASP.NET Core for the backend functionality and Angular for the frontend functionality. The frontend will communicate with the backend using a HTTP REST Api. We separate the two applications in order to have a clean separation of concerns and to enable users to build their own frontend without changing the backend.

3.3.1 Database

To save application data we will use Microsoft SQL Server. SQL Server is a relational database management system. We can connect to a SQL Server instance remotely, with TCP/IP on the port 1433. To connect to a SQL Server instance from our application, we have to provide a connection string that contains different settings, like User Id and Password. After that we can write SQL Queries and read and modify data. We will not write the SQL Queries by ourselves, since we will use Entity Framework, which is an object-relational mapper that encodes our C# queries in SQL Queries.

3.3.2 Backend

The backend is responsible for providing a complete API to the frontend for the user and vehicle management. We will build our backend in ASP.NET Core in C#. We will try to implement the system with clean architecture and a clear separation between different modules of the application. Thus, there are 3 different projects that we will implement:

- OACPCore - The main module that implements the entities, provides standardised access to those and is independent of the other modules
- OACPPersistence - The database access layer, that manages the communication with a database and provides entities for manipulation that get saved into the database
- OACPWebApi - The interface to the application as a REST Api, using MVC (Model-View-Controller) that makes calls to the OACPCore project

OACPCore

The OACPCore project is the main part of the application that contains the business logic of the system. We first have to think about all the different entities that we have to deal with in our software.

First, we have to model the user, which is the entity that can create vehicles and manage them. We can model a user with the standardised Identity Model of ASP.NET core by inheriting from IdentityUser. IdentityUser is a base class from ASP.NET Core that provides us with standard properties which are very useful in user management software, like:

- Username - The username, with which the user can authenticate for this application
- LockOutEnd - Is the account locked out?
- PhoneNumber - The telephone number for the user
- SecurityStamp - A random value that must change whenever a user's credentials change
- PasswordHash - A salted and hashed representation of the password for the user
- NormalisedEmail - The normalised email address for the user

We will extend the base class and add three more properties for the actual entity AppUser:

- FirstName - The first name of the user
- LastName - The last name of the user
- Vehicles - A list that contains all vehicles that belong to the user

We will also add a constructor, that accepts the username and the email address as required values to create a new user. Notice that the password is not needed, since the IdentityModel library provides us with standardised access methods to the user.

The IdentityModel library creates default object that handle the user management. There exist a user manager, a signing manager and even a role manager. In order to use these classes, we have to provide store implementations, meaning we have to provide classes that implement a specific interface that handles the actual connection with a database, that manages how the data is stored and so on. ASP.NET Core provides a default implementation of these user stores with the entity framework, which we will use. This means we only have to add the registration of these user stores to our application and we can then use all the helper classes of this library.

The next entity we need to model is the vehicle itself. A vehicle should always be assigned to a user and have an unique Id. We also need to store a name for the vehicle (this can be the model of the vehicle, like BMW 320i). Also we need to store the mathematical model of it, because this is needed in the symbolic control algorithm. Last but

not least we add a list with protocol modes to the vehicle.

The protocol modes are standardised values in the protocol, which should control different variants for the current session. For our protocol we define the three values:

- LIVE
- DEBUG
- TEST

Every vehicle can be activated for different protocol modes. To capture this we create our last class, the ProtocolModeCredentials class. This class models a protocol mode with all the metadata. It contains a mode, a boolean flag indicating if the mode is currently activated and a secret key that is valid only for this particular mode. It also contains a reference to the vehicle for data store reasons. We also add a constructor to this class, that accepts a vehicle Id, a protocol mode, and the isActive flag. The secret key gets auto generated and assigned to each mode.

The next thing we need to do is to define some services, that provide the interface to the core module. We create a UserService, a VehicleService and a ProtocolModeService class. We then implement standardised methods to query and modify the values in the database, like GetUserByUsername, or CreateUser and so on. We need to keep the references to actual implementations to a minimum, to make use of the dependency injection buil-in feature of ASP.NET Core. Thus, we create an interface of the class and make the class implement that interface. This means we also have an IUserService, an IVehicleService and an IProtocolModeService interface. Last but not least we need to register all interfaces, the database, and do some processing in the start of the application. The problem is, that the OACPCore project, does not contain a reference to the OACPPersistence project, which handles the database connection. In order to maintain our clean architecture, we create an interface called IPersistenceConfiguration, which defines two methods: AddDbContext, and AddUserStores. We will then inject that interface to a method called AddApplication, that handles all initialisation logic. The actual implementation is generated when the OACPWebApi project starts, which is the starting project of our system. The OACPWebApi project will inject an implementation of that interface (of the OACPPersistence class), to the OACPCore project that will call the two methods in that method.

We also add some configuration to our IdentityModel, like the password strength, and the unique email property. After that we register all of our services to the dependency injection system and we save the OACPCore project.

OACPPersistence

The OACPPersistence project, is the data access layer in our system. It is a very small project that only contains the PersistenceConfiguration that the OACPCore project

needs and an ApplicationDbContext class that models a database connection. This DbContext class defines all entities that can be accessed through that class. We define the three entities AppUsers, vehicles and protocol mode credentials.

OACPWebApi

Finally, the OACPWebApi project is the starting point of our whole application. Since we created a modular architecture, we could build a desktop application that runs locally, a web interface, an app that runs in smartphones or anything else. We will build a web interface in order to enable users to access our application via HTTP. For the whole application we will use asynchronous functions that do not block the system when the request queue is blown up.

ASP.NET Core has a built-in server called Kestrel, which we will use. Kestrel accepts a startup class as a parameter, with which we can configure the middleware that is getting called for every HTTP request that arrives. The framework also provides us with a strong logging functionality and configuration handling. We save all of our configurations in the appsettings.json file. We can then access every property of that particular json file with a dictionary approach. In the configuration file, we will save our database connection string, some logging functionality and our security settings.

We will use MVC (Model-View-Controller) for our system. MVC allows us to modify a model with the help of a controller that gets called whenever a particular route is accessed. The controller then transforms that model in to a view and returns the response back to the user.

To use MVC, we first implement the controllers. The controllers, have an Route Attribute with a string parameter, that configures the controller to fire whenever that route is matched. With the Http<"Verb"> Attribute, we can configure methods to get called whenever the corresponding Http-Verb is used in conjunction with the request. Finally the Authorize Attribute configures the controller actions to only get called when the user is authenticated.

To authorise our user we will make use of the JWT (Json Web Token) Bearer Authentication Scheme. According to ..., JSON Web Tokens are an open, industry standard RFC 7519 method for representing claims securely between two parties. The JSON Web Token is a human readable string, that contains claims about a given user. It contains a header, a payload and a cryptographically signed signature. In the header we specify the algorithm and the type of the token. In the payload section we define the claims that the application needs from the user, which we need in order to verify the user and not to make database class to get extra information. The signature contains the hash of the concatenation of the base64 encoded strings of the header and the payload with the algorithm specified in the header. With this approach, we can make sure that the token is generated from the server by calculating the hash of the concatenation of

the header and the payload and then comparing to signature. The unique properties of cryptographic hash functions make the JSON Web Token secure. We register the JWT Bearer authentication scheme with our application and then we can finally start implementing the controllers.

The controller classes should delegate the workload to services that can be called from different controllers. This means that the logic of our web app, is implemented in services again, that wrap the functionality of the OACPCore services in a controller-friendly way. For each entity we create an I<Entity>ControllerService interface and the corresponding <Entity>Controller Service class that implements that interface. The implementation is straight forward and we will not worry about the implementation details.

After that we create our ViewModels, that contain the response format of the controller. This is also very trivial and we will let the implementation details out.

Finally we compile our application and get a dynamic linked library (.dll), that can run on the ASP.NET Core runtime.

3.3.3 Web interface

In order to use the backend system, we will implement a front-end web application with Angular. Angular is a JavaScript Framework built by Google, which supports dynamic data binding, a well architected project and asynchronous calls. Since it is relatively straight forward, we will not discuss with the implementation details and focus on the user interaction concept.

The user can register to the OACP Service and after that log in. He also can create vehicles and choose which protocol modes he wants to use for that specific vehicle. After that he can view all the vehicles in a table and then view the vehicle details by clicking on the vehicle-details button. In the vehicle details the user has the ability to change the name, the mathematical model and to change the activationflag of the protocol modes. Finally, he can download a flash extract which is a file with all the configuration the vehicle should need in order to successfully connect to the server.

3.3.4 Production ready deployment in AWS

In order for a user to use the management system, we will get a virtual machine in Amazon Web Services (AWS). In this machine we will install Internet Information Services (IIS), which is a standard web server for Windows.

To host the final web application we create a new root folder for the web application. We create a new website in IIS and bind the website to the port 80. Next, we add an application to the website under the relative path /oacp/, and choose the physical location of our angular production build to resolve the dependencies. After that we create

a new application under the relative path /oacpback/, and choose the physical location of our ASP.NET Core backend application. We use this architecture to successfully call the backend application from our frontend, since the browser would prevent any http requests to domains other than the initial domain. For example, the frontend can create a new user with a POST request in : http://*:80/oacpback/api/account and get a jwt token with a POST request in: http://*:80/oacpback/api/auth/login.

We will also install SQL Server in this machine. Next, we modify the connection string in our ASP.NET Core application and create a server user, that has full access to the database. The service is now running and waiting for client requests.

3.4 Concept of the OACP

In this section we will create the concept and finally implement the Open Automotive Control Protocol. We will start with writing some basic requirements for the protocol. We will make some decisions regarding the protocols message format, state handling, error handling, security and safety. After that we will go along with the implementation.

3.4.1 Goal of the OACP

To start with the concept we first need to define the goal of the protocol.

The protocol should be used by vehicles that have a fast internet connection. The goal of the protocol is to safely navigate a vehicle from a starting point to an end destination. In order to do so, the vehicle should be registered in an OACP Service, like the one we implemented in the last section, to apply authentication and security schemes.

The OACP Server that accepts vehicle clients, is responsible for the control input generation, data gathering, navigation, path planning, connection check, optimisation of the control sequence and authorisation. The client - the vehicle - is responsible for exact state measurement and sensor fusion, initiation of the connection, encryption initiation and the actuator task execution. We will use a symbolic controller implementation for the control input generation, but it is not mandatory. One could also implement a classic discrete controller with this approach, since the controller implementation is not specified.

3.4.2 Design of the OACP

The design of a protocol involves the choice of protocol properties. We will define following properties:

- Communication pattern
- Transmission pattern

- Design goals
- Message format
- Message structure
- Communication rules
- Security mechanisms
- Error handling

Communication pattern A vehicle should communicate with a server and gather control input information. There should be no external references. This implies that the vehicle does not need to connect to other vehicles, nor that it needs additional information about other external entities. Since there are only two parties involved we will use a client-server architecture for the protocol. This means that one party (the vehicle) initiates the communication and the other (the server) responds accordingly.

However since this is a prototype and we do not know what requirements will exist in the future, we have the capability to enforce inter-vehicle communication via the server. This means, a server could possibly act as a proxy for inter-vehicle communication for vehicles of a specified radius r in a predefined area.

Transmission pattern Only two parties are involved in communication at a time. Since one vehicle connects to one server, we use a One-to-one transmission pattern for the communication.

Design goals In this paragraph we will define the framework for communication.

We first need a fast, lightweight communication between the two parties, since we have to act in real-time to prevent a bad state. Any latencies and delays disturb our controller and can affect the safety of the vehicle.

Next, we need reliable exchanges, but not necessary in our protocol layer, since we can use the reliability of the protocol stack under our protocol.

The protocol should have authentication capabilities built-in. This is mandatory, since a hacker could overwrite our messages and thus change the response of the server. For that reason the protocol should also support encryption of messages.

Message format We can choose between a human-readable format and a binary format. Because there isn't much overhead in a text-based format and it is easy to log and understand by people, (which is a very crucial point in automotive software engineering) debug and trace a communication we will choose the human-readable version. For our own convenience, we will encode the characters in ASCII and don't bother with encodings.

Message structure We will use the JSON format (JavaScript Object Notation) to define our message structure. JSON is a very popular standard that describes the structure of a data. We will define the required fields for a message and additional fields that have to be populated in certain states.

Communication rules In this paragraph we will define some basic communication rules.

First, the server should listen to some port. The communication starts when the client connects to the specified port.

The communication between the two parties is handled by request messages and response messages. The client sends request messages and the server responds with response messages.

In every request message from the client, there should be a command field, which specifies the command the client wants to execute. Also, there should be a state field which specifies the current state of the protocol in the clientside. The initial state is IDLE.

The server responds with response messages. Every response message should define at least the following two fields: The response result and the state of the server. Because both parties send the current state, each of them can synchronise and detect possible errors. Other necessary fields that are result of the command applied, should be integrated in this response.

Command messages may require additional data from the client. A full description can be found in the subsection 3.4.3.6.

The server responds with response messages. A full description can be found in the section 3.4.3.6.

Every state accepts a predefined set of commands. The commands that can be used in every state are called admissible commands of that state. In case that the client sends a non-admissible command, the server should respond with a list of admissible commands.

Security mechanisms In order to have a secure transmission between the two parties, the protocol should support authentication and encryption.
Authentication will be supported with a challenge - response protocol. Encryption should be widely supported, whereas the two parties involved decide which encryption scheme to apply and what parameters to use for that.

Error handling In order to prevent communication deadlocks, the client should get information from the server about the current state and admissible inputs that lead to transition of the state machine. In an error case, the server should attach an error message to the last message and the communication should be closed.

3.4.3 Protocol Specification

The main design goals have now be chosen. We will try now to describe the protocol information. This includes state charts, commands, valid responses, communication rules etc. Every state has a defined responsibility. Transitions between states are taken based on the command of the client and the result of the server.

3.4.3.1 Protocol states

We will define following states and assign responsibilities for each state:

1. IDLE - Handle real time check, database check, validity check, mode check.
2. SESSION - Create session dictionary, authenticate the vehicle.
3. PREDRIVE - Handle encryption configuration, initialise environment (dictionary) for new trip, validate environment based on data from the database, gather data.
4. SESSIONINITIALIZED - Validate the current model of the vehicle before the control loop starts.
5. CONTROLLOOP - Exchange state and input information. Basically this is like a classic control loop, where the vehicle sends the state and receives an input from the server.
6. CRASH - Handle crash state, receive data, call 101.
7. SESSIONCLOSED - Close the connection, release all other resources.

3.4.3.2 Commands

In this subsection we will define the protocol commands. Each command corresponds to an action for the server. Not every command is admissible in each state.

1. CONN - Used to start the session with the server

2. AUTH - Used to start the authentication pattern with the server
3. INIT - Used to initialise the environment
4. ENDINIT - Used to signal the end of initialisation
5. ENC - Used to encrypt the stream
6. DRIVE - Used to start the control loop with the server
7. GETCTL - Used to get a control action from the server
8. DCONN - Used to close the current session
9. CRASH - Used to signal a crash event
10. DATA - Used to send data to the server for logging, debugging, testing etc.
11. ENDDRIVE - Used to end the control loop

3.4.3.3 Responses

In this subsection we will define the protocol responses. Each command message has to be followed by a response message that notifies the client of the execution result of the last command.

1. OK - Signals that the last command was successfully completed
2. NOK - Signals that there was an error with the last command
3. TRY - Used to signal that the last command needs more data in order to complete The TRY response, is used to point out to the client that the command has not been finished yet and needs more data or time to be completed. The last command that was sent to the server is finished either by sending OK or NOK. If a TRY response gets sent, it indicates that the command is still valid and something has to be done until it is finished. This depends on the current command and state.

3.4.3.4 Fields

In this subsection we will define all fields that are needed for a proper communication.

- CMD - Command of a message
- RES - Response of a message
- STATE - Current protocol state of the sending party
- CREDENTIALS - Credentials used for authentication

- IDLE
  - CONN
  - DCONN
- SESSION
  - AUTH
  - DCONN
- PREDRIVE
  - INIT
  - ENDINIT
  - ENC
  - DCONN
- SESSIONINITIALIZED
  - DRIVE
  - DCONN
- CONTROLLOOP
  - GETCTL
  - CRASH
  - ENDDRIVE
- CRASH
  - DATA
  - DCONN
- SESSIONCLOSED
  - No input

Figure 3.3: Admissible commands for every state

- VEHICLEID - The vehicle id of the client
- PLAIN - The plaintext message that will be used for authentication
- IV - The initial value, either used for authentication or encryption
- CHALLENGE - The object, that contains all information needed for authentication (PLAIN and IV)
- CHALLENGERESPONSE - The field, that contains all information needed to validate a vehicle (CHALLENGE and CIPHER)
- CIPHER - The encrypted value of a plaintext (part of the CHALLENGERESPONSE field)
- INITVALUES - Dictionary that contains key-value pairs that initialise the session environment
- MODE - The protocol mode that will be used for the current session (part of the CREDENTIALS)
- STATES - The current state of the vehicle
- OBSTACLES - List of rectangles that specify a bad state for the vehicle
- CONTROLINPUTS - List of inputs to be applied to the vehicle
- REALTIMECHECK - Field that holds all information regarding a possible real-time check of the connection
- ADMISSIBLECOMMAND - List with admissible inputs (sent from the server in case of non-admissible input)
- MISSING - Field that contains a list of keys that need to be initialised before the control loop starts
- VEHICLEMODEL - Field that stores all information regarding the mathematical model of the vehicle
- MODELSTATE - Field inside VEHICLEMODEL that contains a list with all state names of the vehicle
- MODELINPUT - Field inside VEHICLEMODEL that contains a list with all input names of the vehicle
- DIFFEQ - Field inside VEHICLEMODEL that contains a list with differential equations with states named like MODELSTATES and inputs named like MODELINPUT
- ENCRYPTIONDATA - Stores all relevant data to encrypt the stream

- CIPHERMODE - Field inside ENCRYPTIONDATA that specified the ciphermode to be used
- PADDINGMODE - Field inside ENCRYPTIONDATA that specified the padding-mode to be used
- TARGET - A rectangle that specifies the current target of the vehicle

3.4.3.5 Protocol state chart

In figure 3.4 we can look at the OACP Connection State Chart. The main control loop is in the state CONTROLLOOP. Until the distributed systems reach that state different checks take place to ensure a proper communication.

3.4.3.6 Communication rules and data streams

In this section we will analyse the data streams that take place during a typical protocol session. We will analyse the semantics of each state and provide example communication messages for clarification of possible misunderstanding.

Connection First, the server has to listen on some predefined endpoint. When a vehicle connects to that endpoint, the communication is started and both systems are in the state IDLE.

IDLE After that the vehicle should send a CONN command with the credentials, which are constituted of the vehicle id and the protocol mode that will be used for the current session. The server should then do the following:

1. Check if the vehicle with the protocol mode exists in the database and if so, query data from the database
2. Do some real time check or connection check
3. If everything is fine, the server sends an OK response and transitions into the SESSION state

For the real time check, the server can choose among different options. For this thesis, only a real time check function is defined, in which the server measures the round trip time of the communication by sending random string to the client, which the client has to send back as fast as possible. This is done multiple times. Finally, the median is chosen and checked against the maximum round trip time.

We can find an example communication extract for the IDLE state in 3.5: When the server responds with OK and with the state SESSION, the vehicle can change state.

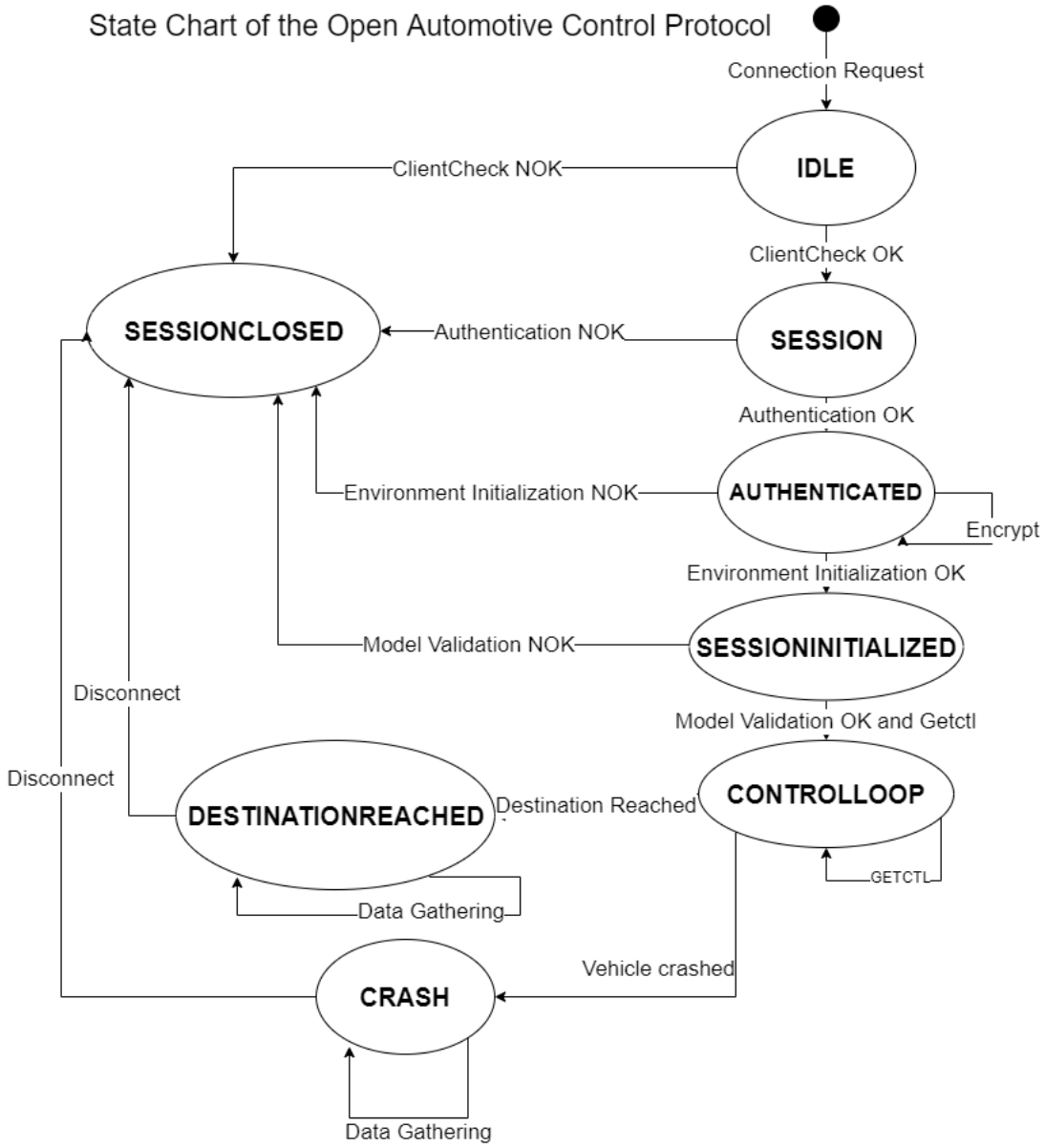


Figure 3.4: The OACP State Chart

```
Client:
{
  "CMD": "CONN",
  "STATE": "IDLE",
  "CREDENTIALS": {
    "VEHICLEID": "a0535dc5-338c-4b38-bcc4-9ea14676cd72",
    "MODE": "LIVE"
}
Server:
{
  "STATE": "IDLE",
  "RES": "TRY",
  "REALTIMECHECK": {
    "Nonce": "827617669"
}
Client:
{
  "CMD": "CONN",
  "STATE": "IDLE",
  "REALTIMECHECK": {
    "Nonce": "827617669"
}
Server:
{
  "STATE": "SESSION",
  "RES": "OK"
}
```

Figure 3.5: Example listing of a communication in the IDLE state

SESSION When the systems are in the state SESSION, the client should try to authenticate to the server. So the client should send an AUTH command.

The server should in response first get the secret key for the vehicle and the protocol mode. After that it should generate a random byte array and an initial value (nonce) and send back the AUTH command with a CHALLENGE field, which contains the initial value and the random string base64 encoded.

The client should then send back a message with a CHALLENGERESPONSE field that contains the CHALLENGE and a CIPHER field which contains the encrypted version base 64 encoded. After that the server can encrypt the random value with the secret key of the vehicle and validate that the two cipher are the same.

If the validation passes, the server should send back an OK response with the PREDRIVE state.

We can find an example communication extract for the SESSION state in 3.6: When the server responds with OK and with the state PREDRIVE, the vehicle can change state.

PREDRIVE This state is the main state before the actual control happens. In the state PREDRIVE the vehicle should initialise the session environment, and optionally request the encryption of the communication stream for the new trip.

For the encryption, the client should send an ENC command with an ENCRYPTIONDATA field, that contains all data that the server would need to encrypt the stream. The server should implement many possible encryption schemes. In our implementation we used AES-256 with Cipher Block Chaining and PKCS7 for the padding mode, and it is the only accepted encryption scheme. The definition of possible encryption schemes, depends on the service provider. This architecture allows the client to implement an encryption function that suits for that particular vehicle or to possibly skip the encryption part if the vehicle is a simulator and a hacker has no possibility to do harm.

The server should respond with an OK response and an unchanged state. If the server had an internal problem, it should respond NOK and the client could try another encryption scheme.

If no other drive is going to happen, the client can close the session with the DCONN command.

In figure 3.7 we can look at an example of the encryption messages. For the initialisation, the user can specify what information should be passed to the server from the vehicle in every session. For example the user can specify that before the control

```
Client:
{
  "CMD": "AUTH",
  "STATE": "SESSION"
}
Server:
{
  "STATE": "SESSION",
  "RES": "TRY",
  "CHALLENGE": {
    "PLAIN": "Ob1/45zPP9OKKEATondSh1/Ch5mlxcohG4ayTKg9kwM=",
    "IV": "tZdTMWw8IfAYssBas0s+BQ=="
  }
}
Client:
{
  "CMD": "AUTH",
  "STATE": "SESSION",
  "CHALLENGERESPONSE": {
    "CHALLENGE": {
      "PLAIN": "Ob1/45zPP9OKKEATondSh1/Ch5mlxcohG4ayTKg9kwM
        =",
      "IV": "tZdTMWw8IfAYssBas0s+BQ=="
    },
    "CIPHER": "+A92izSetCko2VCEE4nwBF+mkijCy8itKdH6MaEKPR0="
  }
}
Server:
{
  "STATE": "PREDRIVE",
  "RES": "OK"
}
```

Figure 3.6: Example listing of a communication in the SESSION state

```
Client:
{
  "CMD": "ENC",
  "STATE": "PREDRIVE",
  "ENCRYPTIONDATA": {
    "IV": "AAAAAAAAAAAAAAAAAAAAAA==",
    "CIPHERMODE": "CBC",
    "PADDINGMODE": "PKCS7"
  }
}
Server:
{
  "STATE": "PREDRIVE",
  "RES": "OK"
}
```

Figure 3.7: Example listing of the ENC command

loop, the vehicle should send some information regarding the errors that are at this time recorded in the vehicle, or a humidity measurement. Based on that data, the engineers could possibly test some application or analyse the data to develop new features.

The client can initialise the environment with the command INIT and the field INITVALUES. The field INITVALUES is basically a dictionary, a collection of key-value pairs. The client can send multiple INIT commands to the server. In case of the same key in different requests, the value of the last key should be applied.

In response to that the server should send an OK response and not change state.

When the client has finished the initialisation, it should send an ENDINIT command. The server should then validate that the required key-value pairs are created. If not, the server should send an NOK response with a MISSING field, that is basically a list with all of the keys that are not initialised but have to be initialised to start the control loop. When the client then sends these keys and they pass the validation of the server the server should send an OK response with the state SESSIONINITIALIZED.

We can find an example communication extract for the PREDRIVE state in 3.8. When the server responds with OK and with the state SESSIONINITIALIZED, the vehicle can change state.

SESSIONINITIALIZED In the SESSIONINITIALIZED state, the vehicle can send the DRIVE command to finally start the control loop. The vehicle has to provide the mathematical model of it in the VEHICLEMODEL field, in order for the server to

```
Client:
{
  "CMD": "INIT",
  "STATE": "PREDRIVE",
  "INITVALUES": {
    "Position": "Munich",
    "Target": "Augsburg"
  }
}
Server:
{
  "STATE": "PREDRIVE",
  "RES": "OK"
}
Client:
{
  "CMD": "ENDINIT",
  "STATE": "PREDRIVE"
}
Server:
{
  "STATE": "PREDRIVE",
  "RES": "NOK",
  "MISSING": [
    "Humidity"
  ]
}
Client:
{
  "CMD": "INIT",
  "STATE": "PREDRIVE",
  "INITVALUES": {
    "Humidity": 0.8
  }
}
Server:
{
  "STATE": "PREDRIVE",
  "RES": "OK"
}
Client:
{
  "CMD": "ENDINIT",
  "STATE": "PREDRIVE"
}
Server:
{
  "STATE": "SESSIONINITIALIZED",
  "RES": "OK"
}
```

```
Client:
{
  "CMD": "DRIVE",
  "STATE": "SESSIONINITIALIZED",
  "VEHICLEMODEL": {
    "MODELSTATES": [
      "x1",
      "x2",
      "x3",
      "x4"
    ],
    "MODELINPUT": [
      "u1",
      "u2"
    ],
    "DIFFEQ": [
      "xx1=x2",
      "xx2=x3",
      "xx3=x4",
      "xx4=x3+x2*3+u1+u2"
    ]
  }
}
Server:
{
  "STATE": "CONTROLLOOP",
  "RES": "OK"
}
```

Figure 3.9: Example listing of a communication in the SESSIONINITIALIZED state

validate that the mathematical model is the same like the engineer specified in the registration system. The VEHICLEMODEL field consists of three fields. The MODELSTATES, the MODELINPUT and the DIFFEQ fields. The model states and model input fields are variable names to use within the DIFFEQ field.

This makes sure that the control inputs sent from the server are valid for that vehicle, and that no other model can be used from the user without registering it. The server can respond with OK or NOK . In 3.9 we can see an example of the communication in the SESSIONINITIALIZED state. When the server responds with OK and with the state CONTROLLOOP, the vehicle can change state.

CONTROLLOOP The state CONTROLLOOP is the main state of the protocol. In this state the actual control of the vehicle gets executed. The vehicle should send a GETCTL command with a STATES, OBSTACLES, OPT and a TARGET field.
The STATES field should contain the current state of the vehicle, like specified in the DIFFEQ field of the VEHICLEMODEL field.
The OBSTACLES field should contain a list with obstacle hyperrectangles which are described by a lower and an upper bound coordinate for each state. This field specifies the hyperrectangles that are considered "bad states" for the vehicle to be.
The TARGET field should contain a hyperrectangle which is describes by a lower and an upper bound coordinate for each state. The target specifies the hyperset that the vehicle should eventually reach.

The server should delegate the computation to a symbolic control platform and receive the valid input for the current state. After receiving the input sequence for the current state, the server should filter the input sequence to fulfill the current optimisation function, like specified in OPT.

The server should then send an OK response with a CONTROLINPUTS field, that is a list of dictionaries, which specify for every sampling time of the actuator, which inputs to apply. This is the main control loop of the protocol. In figures 3.10 and 3.11 we can see a typical control loop. The protocol includes a CRASH command, which notifies the server that a crash happened. The server should then take appropriate actions to handle the situation, but it is not specified what exactly. This also depends on the current mode of the protocol.

When the vehicle finally arrives at the destination, an ENDDRIVE command should be sent. The server should respond with an OK response and a PREDRIVE state.

CRASHED In the CRASHED state, the server could be programmed to do a standard action or to gather data. This is left unspecified.

SESSIONCLOSED In the SESSIONCLOSED state the server and the client should just close the connection and release resources that were hold by the systems.

3.5 Implementation of the OACP

In the last section we wrote the specification of the Open Automotive Control Protocol. In this section we will discuss the implementation of a server and a client vehicle that will use the protocol. We will also implement an OACPProtocol library, that handles the message stream. We will use the C# programming language and .NET Core.

```
Client:
{
  "CMD": "GETCTL",
  "STATE": "CONTROLLOOP",
  "STATES": {
    "x1": "5",
    "x2": "2.496886617366581",
    "x3": "2.123372836753095"
  },
  "OBSTACLES": [
    {
      "x1_lb": "0.6501399766684627",
      "x1_ub": "0.11321297686006215",
      "x2_lb": "0.4400519980933788",
      "x2_ub": "0.24529668741434751",
      "x3_lb": "0.0176468321552513 13",
      "x3_ub": "0.8623260048109105"
    },
    {
      "x1_lb": "0.1154283012976608",
      "x1_ub": "0.1269495662331369",
      "x2_lb": "0.18270469854888738",
      "x2_ub": "0.9389724770448382",
      "x3_lb": "0.5601018202284388",
      "x3_ub": "0.3445666182467304"
    }
  ],
  "TARGET": {
    "x1_lb": "0.28212875574445384",
    "x1_ub": "0.7231408715554216",
    "x2_lb": "0.3148305724603526",
    "x2_ub": "0.10455548801250047",
    "x3_lb": "0.38975663077371836",
    "x3_ub": "0.732221383098693"
  }
}
```

Figure 3.10: Example listing of the control loop clientside

```
Server:
{
  "STATE": "CONTROLLOOP",
  "RES": "OK",
  "CONTROLINPUTS": [
    {
      "u1": "0.7",
      "u2": "1.3"
    },
    {
      "u1": "3.43",
      "u2": "1.23"
    },
    {
      "u1": "4.3",
      "u2": "1.5"
    }
  ]
}
```

Figure 3.11: Example listing of the control loop serverside

3.5.1 The OACPProtocol library

The protocol library should contain functionality regarding the transmission of messages, encryption of stream, the state, command and response objects and a message class that models an OACP message. Ideally, in order for the server and the client to use the library, the library should give us the capability to:

1. Read protocol field names
2. Create messages
3. Write messages to the stream
4. Read messages from the stream
5. Encrypt the stream

We will start with th creation of a .NET standard project.

First we will create static classes that contain constant strings which will be used for the name of our protocol fields. The classes which we will create are:

- OACPState
- OACPCommand

- OACPResponse

We will use these classes for convenience, in order to have the field names of the protocol in a centralised class, so that we do not have to use different names for the same object. For example to access the state PREDRIVE we will use the name OACPState.PREDRIVE.

Next we will create the OACPMessage class, which will be used to create a message, and read the properties from it in a standardised way. The OACPMessage class, should contain all possible fields of the protocol. The value of a non-existent field of a message should be null. We will attach public fields to OACPMessage objects, that map to the protocol fields like specified in the last section. We will also implement convenience response properties, like OACPMessage.OK or OACPMessage.NOK, that create an empty OACPMessage response object and a Fluent API like interface to create and add specific fields to the message object. The functions will be named Add<FieldOfProtocol> and return the current message object for a direct use after the function. For example to create the connection message we could use: OACPMessage().AddCommand(conn).AddCredentials(credentials).AddState(state). We will also create a constructor, that accepts a string object in json format, deserialises the string and returns the corresponding OACPMessage object.

Finally, we will create the class OACPStream, which abstracts away the sending and reading of OACPMessages. The OACPStream class is responsible for the communication part. The constructor accepts a base stream, which is the stream that we will use as an underlying communication layer and a function pointer to a function that accepts a stream way argument and returns void. This function can be used for logging purposes. Every time someone sends or receives a message, an event will be raised, notifying the party that a communication event has happened. The corresponding event handler will then be called, which can log the message.

The base stream has to use a reliable protocol. For this matter, we will use TCP/IP for the network and transport layer. TCP/IP opens a bidirectional communication channel between two endpoints. It can be thought as a stream of data.

The stream should accept a function pointer, which, when called, returns the current state of the active party. This allows the client and server to focus on the message itself instead of taking care of sending the current state every time. This function, will be called when the party initiates a write request. The given OACPMessage will then be appended with the current state of the sending party.

There are many details that have to be considered in network programming. For example, the read function accepts a buffer to fill and an integer argument that specifies the maximum number of bytes to read from the stream. The received bytes can be smaller than the maximum receivable bytes. In order to write messages, we need to tell the

stream how many bytes we should read in order to construct the whole message.

There are three different approaches for size control of messages:

1. Fixed size messages
2. Control character ending of messages
3. Variable length messages with fixed header

We will use variable length messages with a fixed 4 byte header as an integer, that specifies the length of the message. The byte order should be network order (big endian). The message follows immediately.

The stream has to support symmetric encryption. Different algorithms can be used. Most symmetric ciphers operate on blocks. In case of an encrypted stream, the first four bytes represent still the unencrypted header. To build such a stream we will use different stream classes.

The input to our write function is the OACPMessage that needs to be sent through the network. There are two different paths that the message can follow.

Unencrypted stream Sender: First, we serialise the message object to a string. Then, we convert the string to a byte array with ASCII. After that we calculate the length of the byte array and send the header as integer. Finally, we send the whole byte array after the header.

Client: First we read four bytes from the stream and compose the integer value n. After that we read n bytes from the stream and decode the bytes in ASCII. Finally, we call the constructor of the OACPMessage that accepts a json string to get the message object.

Encrypted stream Sender: First, we serialise the message object to a string, like in the unencrypted stream. Then we convert the string to a byte array. The byte array gets fed into the cryptostream, that encrypts the whole byte array. Because of the nature of block ciphers, we have to flush the stream in case that the byte array is no multiple of the block size. The padding scheme should be known to the stream, based on the encryption parameters. The encrypted byte array gets saved into memory with the memorystream class. After that, we calculate the length of the encrypted byte array and send the header. Finally, we write each encrypted byte to the stream.

Client: Like in the unecrypted stream we first read the first four bytes and compose the length of the message. The client knows that the stream is encrypted. After that we read n bytes from the stream and save them into an internal buffer. This buffer gets read by the cryptostream class, which decrypts the encrypted byte array. The decrypted

bytes are then decoded with ASCII. Finally, the string gets passed to the constructor of the OACPMessage class, which creates a new OACPMessage object that we can use. The stream should support different encryption schemes. For our convenience we will only implement AES 256.

The stream should throw exceptions in case of errors. We will set the read and write timeout of the underlying socket to 10000 ms.

The structure of the main Read and Write functions of the stream are shown in figure 3.13.

3.5.2 The OACPServer

The OACPServer is the software application that will listen for incoming connections and handle the communication between the client and the controller.

The server will reference the protocol library and use the classes and methods of that library to communicate with the client.

The server should be able to accept multiple clients, to connect with the database and query data about every vehicle, to connect with the symbolic controller and exchange data, to log the messages of every session, to handle errors etc.

The entry point of our application is the OACPServer class. The OACPServer class can be instantiated with a port and an optional logfolder. The port is used for the endpoint creation in order to listen for incoming connections. The logfolder is used as a directory, where OACP sessions are saved.

After the server object is created, the only public function that is supported id the start method. The start method starts a tcp listener on the specified endpoint. When a new client connects to that endpoint, a new thread, that handles the session, is created. The main thread can then accept another client and instantiate a new session.

An OACP session is modeled by the OACPServerSession class. The OACPServerSession class accepts a stream, an optional logger, and a symbolic controller factory. The logger will be called in every message event like specified in the protocol library. We get the stream from the TcpClient that connected to our server. The symbolic controller factory is a class that implements the ISymbolicControllerFactory interface, that specifies that every symbolic controller factory should have a method that returns a concrete instance of that symbolic controller.

The OACPServerSession class models a session with the client. In a high level the execution of the session is as follows:

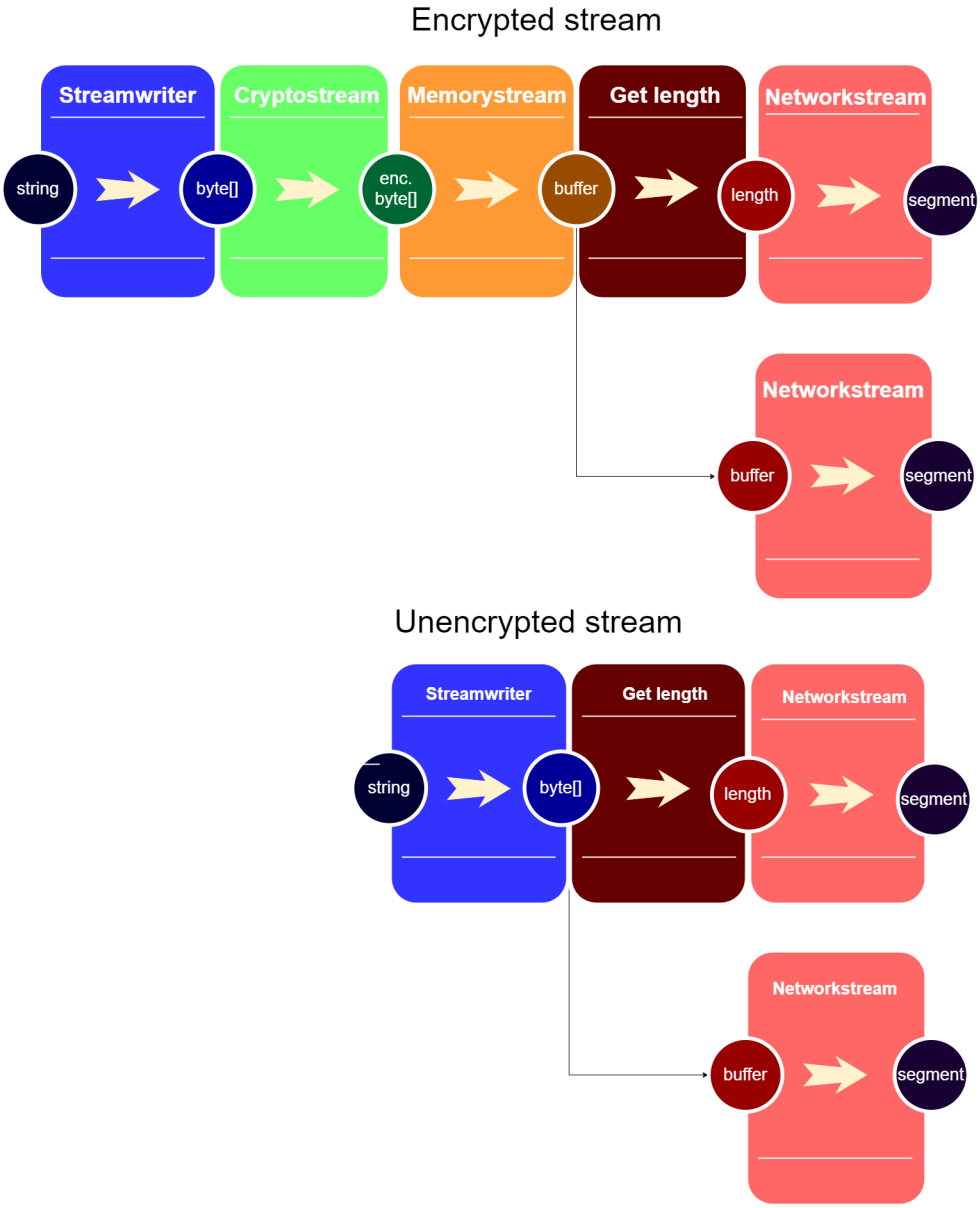


Figure 3.12: Stream pipeline of the OACPStream class

```
0 Verweise
public OACPMessage Read()
{
    //Read the header
    int bodyLength = ReadOACPHeader();

    //Read the body - Encrypted or not, we dont care since the stream is handling it in deserializing method
    byte[] body = ReadOACPBody(bodyLength);

    //Deserialize the body to an OACPMessage object
    OACPMessage msg = DeserializeToOACPMessage(body);

    //Fire the read event
    OnMessageEvent(msg, OACPStreamWay.Read);

    return msg;
}

0 Verweise
public bool Write(OACPMessage msg)
{

    //General processing of the message
    msg = ProcessMessage(msg);

    //Get the byte[] to write
    var body = SerializeOACPMessage(msg);

    //Write the header
    WriteOACPHeader(body.Length);

    //Write the body
    WriteOACPBody(body);

    //Writing EVENT
    OnMessageEvent(msg, OACPStreamWay.Write);

    return true;
}
```

Figure 3.13: Structure of the OACPStream class

1. Update session configuration, based on flags of the last run if not absent
2. Accept a request message
3. Handle the request, produce flags, produce a response
4. Handle state transition
5. Send the response back

After creating a new OACPServerSession in the new thread, we call the Run() method of that session. The run method is the execution loop of the OACP session, which executes the steps that we previously defined.

The state machine is implemented as a dictionary. The key of the dictionary is the current state of the session. The value of each key-value pair is an object of a class that extends the base class StateHandler.

The class StateHandler is a base class that models the basic properties and methods of each state of the protocol. The StateHandler class accepts a session parameter, which should be the current session object. This object is used for communicating between states, controlling the stream, setting encryption flags etc. Basically it is used as a communication mechanism between states and the session. The StateHandler class has also a list of string with the admissible commands of the current state and a transition table which is a dictionary that accepts an OACPCommand string and returns a function to handle the transition in the current state. Also, a DefaultHandle() method is implemented, which when called, returns an OACPMessage with a list of admissible commands, based on the admissible commands list of that class. Finally, convenience functions are implemented, which allow a class to specify the state which they should run, modify the underlying transition table and admissible command list, called SetState and AddTransition.

The session object, first checks for the current state and selects an object of the local state machine dictionary that is of type StateHandler. After that, the session calls the corresponding Handle() method of that object, and passes the current request message as argument. Tha Handle() method is implemented in the StateHandler base class. The handle method just checks the transition table for a key with the value of the command of the current message. If present, it delegates work to the corresponding method. If not, the DefaultHandle() method is called, which returns a list of admissible commands.

For every state, there is a corresponding <State>Handler class. The subclasses should first set the right state for the current class. After that they should modify the transition table in order to function properly. For every admissible command, a corresponding method should be implemented called <Command>Handle that accepts an OACPMessage as parameter. Every Handle method returns a string-OACPMessage tuple, that

represent the next state, and the response to that message.

For example, let's take a look at the PreDriveHandler class. We first set the current state to OACPState.PREDRIVE. After that we specify the transitions that are valid in this class, and the corresponding command handlers, that handle the various commands. Every handle function should be responsible for one and only one command. The handle functions can read and modify the current OACPServerSession object. For example, the InitializeHandle method reads the initialisation dictionary from the message and updates the SessionDictionary property of the OACPServerSession object. If the initialisation was successful, the method returns the current state, and an OK response.
The EndInitHandle method, first checks if all keys that need to exist, are present in the current SessionDictionary. If so, then the method returns the state SESSIONINITIALIZED and an OK response. If not, the current state with a NOK response is returned and an additional field MISSINGKEYS, specifiying which keys need to be passed in order to continue. Another important detail of the application is the registration of

```
public (string state, OACPMessage response) Handle(OACPMessage message)
{
    if (TransitionTable.ContainsKey(message.OACPCommand))
    {
        return this.TransitionTable[message.OACPCommand](message);
    }

    return DefaultHandle();
}

private (string state, OACPMessage response) DefaultHandle()
{
    return (State, OACPMessage.NOTOK.AddAdmissibleCommand(this.AdmissibleCommands));
}
```

Figure 3.14: Snippet of the StateHandler class

the currently running vehicles. The service should not accept connections with two clients that have the same id. For that reason, we save the running vehicle ids in an in-memory list. This threadsafe in-memory list is implement with the static ActiveSessions class. The IdleHandler calls the function AddVehicleId of that class and when the session finishes, we call the RemoveVehicleId with the given vehicle id. We can call the GetRunningSessions of that class, to get a string with the current running vehicle ids.

Another important class, is the OACPRepository class. The OACPRepository class handles the communication with the registration system and the database. This static class connects to the registration system with a http client, and the server credentials. In the registration system, we already created a server user with some password, and authorised the server to query every vehicle in the database. The IdleHandler, calls the

```
public class PreDriveHandler : StateHandler
{
    public PreDriveHandler(OACPServerSession session) : base(session)
    {
        this.SetState(OACPState.PREDRIVE);
        this.AddTransition(OACPCommand.DISCONNECT, DisconnectHandle);
        this.AddTransition(OACPCommand.INITIALIZE, InitializeHandle);
        this.AddTransition(OACPCommand.ENDINIT, EndInitHandle);
        this.AddTransition(OACPCommand.ENCRYPT, EncryptHandle);
    }

    private (string state, OACPMessage response) DisconnectHandle(OACPMessage msg)
    {
        return (OACPState.SESSIONCLOSED, OACPMessage.OK);
    }

    private (string state, OACPMessage response) EndInitHandle(OACPMessage msg)
    {
        if (!CheckEnvironment(this.OACPSession.SessionDictionary))
        {
            var keys = GetMissingKeys(this.OACPSession.SessionDictionary);
            Logger.Log("There are missing keys. Following keys are missing:\n");
            foreach (var key in keys)
            {
                Logger.Log(key);
            }
            return (State, OACPMessage.NOTOK.AddMissingKeys(keys.ToArray()));
        }

        Logger.Log("There are no missing keys. The session is initialized");
        return (OACPState.SESSIONINITIALIZED, OACPMessage.OK);
    }
```

Figure 3.15: Snippet of the PreDriveHandler class

LoadVehicleData method of that class, and accepts a VehicleData object, which contains information about the current vehicle, like the Id, the mathematical model, the secret key of the current protocol mode and so on.

The last important part of this application is the symbolic controller family. We created a ISymbolicControllerFactory interface, that has only one method GetSymbolicController, that should return an object that implements the ISymbolicController interface. This design pattern is called abstract factory pattern. The ISymbolicController interface defines the GetControl method, that accepts the states of the vehicle, the obstacles and a target, and returns a List of inputs to apply in order to reach the specified target. With this abstract definition we can exchange our implementation of pfaces that implements the ISymbolicController interface with another implementation easily. Details of the pfaces connection are not important for this thesis, so we will omit the details.

Finally, the static Optimizer class, is responsible for optimising the received inputs of the symbolic controller.

3.5.3 The OACPClient

The OACPClient is another software application, that should validate the function principle of the protocol. It was built only for validation and test purposes.

The functionality is trivial and not worth mentioning. The OACPClientSession contains the same state machine like the server does. The only difference is that the client is initiating the communication and the commands.

What is worth mentioning is the vehicle interface. The OACPClient, should have a connection with the client vehicle. In a real scenario, the communication task should periodically run on an ECU on that vehicle and handle the communication with the sensing and actuating ECUs, or even communicate with other tasks on the same ECU. Our client application is just a dummy application, but it still marks the interface that the vehicle should fulfill in order to use the protocol.

In order to use the protocol, we need some static data, that doesn't change during the lifetime of the vehicle, static data that does change during the lifetime of it but remains static for a session and dynamic data, that is generated during the trip. Here is a list with the data needed for the OACP:

- Vehicle ID - Static for the lifetime
- Protocolmode - Static for the lifetime
- SecretKey - Static for the lifetime
- Mathematical Model - Static for the lifetime

```
typedef struct _oacpdata
{
    int VehicleId;                          // 4 Byte Id of the vehicle
    char *ProtocolMode;                     // 4 Byte Pointer to the protocolmode string
    unsigned char SecretKey[32];            // 32 Byte secret key used for authentication and enc
    char *MathModel;                        // 4 Byte Pointer to the math model
    unsigned char *(*getInitValue)(void); // Function pointer to a function that returns an initialization value
    char *(*getCipherMode)(void);           // Function returning the cipher mode
    char *(*getPaddingMode)(void);          // Function returning the paddingmode
    char *(*initializeEnvironment)(void); // Function used to initialize the environment
    char *(*getInitValues)(char *key);      // Function returning a value in a key-value pair
    char *(*getCurrentStates)(void);        // Function returning the current state
    char *(*getObstacles)(void);            // Function returning the current obstacles
    char *(*getTarget)(void);               // Function returning the current target
} OACPData;
```

Figure 3.16: The structure of the OACPData type

- Initial value for encryption - Static during a session
- Ciphermode - Static during a session
- Paddingmode - Static during a session
- Initvalues - Static during a session
- Dynamical state - Dynamic, generated in runtime
- Obstacles - Dynamic, generated in runtime
- Target - Dynamic, generated in runtime

Instead of giving the vehicle the workload to do the protocol handling, we will revert the responsibility and let the protocol stack call the desired functions and access the needed data. In order to do so we also need some functions, that should be called from the protocol stack, for example GetStates() or GetObstacles(). This idea, gives us the opportunity to define a standardised protocol stack, that accepts as a parameter a pointer to a memory section, where we can find the data and the functions that protocol has to use in order to function properly.

It is typical for structured data to be represented as a struct in the C programming language. Because ECUs are programmed with C and C++, we will define such a struct that fully describes the protocol data. In order to use the protocol, the engineer have to define a memory section "OACPDATA", and to calibrate that memory section with the defined data. The flash extract can be downloaded from the web interface. The protocol stack should then use the data and functions when needed.

4

Conclusion and Outlook

In this chapter we will draw a conclusion and make a future outlook.

4.1 Conclusion

We are at the beginning of the connected world. With 5G technology knocking on our doors, new technologies and ideas will take over the world. The dream of autonomous cars is very close to fulfillment. In this thesis we proposed a protocol, for the remote control of autonomous vehicles, the Open Automotive Control Protocol.

The OACP satisfies our requirements. It is easy to understand, the structure is clear, there are very few keywords, the use of standardised communication patterns in text format (JSON) make it extensible and easy to parse. Built-in authentication and encryption services make sure that the users stay safe.

The developed enterprise application is a prototype and should be handled like that. However, many problems were solved.

Now, there exists a standardised interface for remote control of vehicles. The protocol is easy to use and an API written in Python 3 and C# exists, with which servers can be built.

The implementation supports logging, encryption and authentication. Multiple clients can be connected concurrently. The server disposes clients that do not fulfill the real time check. With the database coupling and the role assignment, an easy to use architecture was introduced to ease the implementation of future applications. The database acts as a central repository with which engineers can tweak their system to fit their needs. Cumbersome and errorprone implementations of ECUs can be swapped out with easy to use OACP interfaces. Dynamic data that could change for every driver or every company should be stored in the database and validated against during the startup of the protocol. This makes the protocol extensible and customisable.

4.2 Outlook

The introduced specification and implementation of the OACP works fine. However, there are various specification topics that need to be analysed separately:

- How should the vehicle behave when the connection terminates?
- Definition of various system modes and their role in the protocol
- Backup autonomous controller specification and mode switch specification

Also, a validation of the protocol and its performance should be performed including:

- Live test with a real vehicle or a simulation environment
- Time measuring of the protocol and performance analysis
- Stress test of OACP server and bandwidth analysis

Additionally, various improvements and extensions can be built upon this thesis:

- Design of a neural network that accepts a sequence of test data and builds the model of the vehicle (Easier to use for automotive companies)
- Various function hooks that can be implemented serverside, to capture, log and even modify the control input
- AUTOSAR specification and implementation of a protocolstack
- Interface specification of a remote controller to change controller implementation on the fly

There are several topics and ideas that the interested reader could research on. The only boundary is the human mind and imagination or as the famous Albert Einstein once said:

> "Imagination is more important than knowledge. For knowledge is limited to all we now know and understand, whereas imagination embraces the entire world, stimulating progress, giving birth to evolution."

Bibliography

[AKM] Matthias Althoff, Markus Koschi, and Stefanie Manzinger. Commonroad: Composable benchmarks for motion planning on roads. pages 719–726.

[Beh] Behrisch, Michael & Bieker-Walz, Laura & Erdmann, Jakob & Krajzewicz, Daniel. Sumo – simulation of urban mobility: An overview.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, Cambridge, Mass., 2008.

[dGV13] Giuseppe de Giacomo and Moshe Vardi. Linear temporal logic and linear dynamic logic on finite traces. *IJCAI International Joint Conference on Artificial Intelligence*, pages 854–860, 2013.

[Dil] Jochen Dilling. Fahrverhalten von kraftfahrzeugen auf kurvigen strecken.

[Dip] Dipl.-Ing. Dirk Ebersbach. Entwurfstechnische grundlagen für ein fahrerassistenzsystem zur entwurfstechnische grundlagen für ein fahrerassistenzsystem zur unterstützung des fahrers bei der wahl seiner geschwindigkeit wahl seiner geschwindigkeit.

[DRC+] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. Carla: An open urban driving simulator.

[GBH+] Maxime Gueriau, Romain Billot, Salima Hassas, Frederic Armetta, and Nour-Eddin El Faouzi. An extension of movsim for multi-agent cooperative vehicles modeling. pages 859–860.

[HHH] Sara Haddouch, Hanaa Hachimi, and Nabil Hmina. Modeling the flow of road traffic with the sumo simulator. pages 1–5.

[HHL19] Héctor H González-Baños, David Hsu, and Jean-Claude Latombe. Motion planning: Recent developments. 2019.

[HMMS] Kyle Hsu, Rupak Majumdar, Kaushik Mallik, and Anne-Kathrin Schmuck. Multi-layered abstraction-based controller synthesis for continuous-time systems. pages 120–129.

[JK12] Jonathan A. Thompson and Kristofer Schlachter. An introduction to the opencl programming model - semantic scholar, 2012.

[Joe] Joerg Krueger. Simulation of modern traffic lights control systems using the open source traffic simulation sumo.

[JSRG] Johannes Hiltscher, S. V. N. Phanindra Akula, Robin Streiter, and Gerd Wanielik. A flexible automotive systems architecture for next generation adas.

[KZ] Mahmoud Khaled and Majid Zamani. pfaces. pages 252–257.

[LS17] Edward Ashford Lee and Sanjit Arunkumar Seshia. *Introduction to embedded systems: A cyber-physical systems approach.* MIT Press, Cambridge, Massachuetts and London, England, second edition edition, 2017.

[Mar18] Peter Marwedel. *Embedded System Design: Embedded Systems, Foundations of Cyber-Physical Systems, and the Internet of Things.* Embedded Systems. Springer International Publishing, Cham, 3rd edition 2018 edition, 2018.

[MDS95] Roger C. Mayer, James H. Davis, and F. David Schoorman. An integrative model of organizational trust. *The Academy of Management Review*, 20(3):709, 1995.

[Mic] Michael Lazar. Current cloud computing statistics send strong signal of what's ahead.

[MM] M. Khaled and M. Zamani. Cloud-ready acceleration of formal method techniques for cyber-physical systems.

[Mor18] Moritz Lipp and Michael Schwarz and Daniel Gruss and Thomas Prescher and Werner Haas and Anders Fogh and Jann Horn and Stefan Mangard and Paul Kocher and Daniel Genkin and Yuval Yarom and Michael Hamburg. [pdf] meltdown: Reading kernel memory from user space - semantic scholar, 2018.

[MPS] Oded Maler, Amir Pnueli, and Joseph Sifakis. On the synthesis of discrete controllers for timed systems. volume 900, pages 229–242.

[OAD] Bogdan Oancea, Tudorel Andrei, and Raluca Mariana Dragoescu. Gpgpu computing. *Proceedings of the CKS International Conference.*

[Pab] Pablo Alvarez Lopez, Michael Behrisch, Laura Bieker-Walz, Jakob Erdmann, Yun-Pang Flotterod, Robert Hilbrich, Leonhard Lucken, Johannes Rummel, Peter Wagner and Evamarie WieBner. Microscopic traffic simulation using sumo.

[PCY+16] Brian Paden, Michal Cap, Sze Zheng Yong, Dmitry Yershov, and Emilio Frazzoli. A survey of motion planning and control techniques for self-driving urban vehicles. *IEEE Transactions on Intelligent Vehicles*, 1(1):33–55, 2016.

[PS95] Angel Pasqual del Pobil and Miguel Angel Serna. *Spatial representation and motion planning*, volume 1014 of *Lecture notes in computer science*. Springer, Berlin, 1995.

[RH] G. Reichart and R. Haller. Mehr aktive sicherheit durch neue systeme für fahrzeuge und straßenverkehr.

[RWR17] Gunther Reissig, Alexander Weber, and Matthias Rungger. Feedback refinement relations for the synthesis of symbolic controllers. *IEEE Transactions on Automatic Control*, 62(4):1781–1796, 2017.

[RZ] Matthias Rungger and Majid Zamani. Scots. pages 99–104.

[SB18] Naresh Kumar Sehgal and Pramod Chandra P. Bhatt. *Cloud Computing*. Springer International Publishing, Cham, 2018.

[SZL16] Sonja Stockert, Andreas Zimmermann, and Markus Lienkamp. Mensch-maschine-interaktion für eine kraftstoffeffiziente, automatisierte fahrzeuglängsführung. *ATZextra*, 21(S8):16–19, 2016.

[TA09] Paulo Tabuada and Rajeev Alur. *Verification and control of hybrid systems: A symbolic approach*. Springer, Dordrecht, 2009.

[TK13] Martin Treiber and Arne Kesting. *Traffic Flow Dynamics: Data, Models and Simulation*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[Ton] Tonu Lehtla. Introduction to robotics.

[Uni15] Univ.-Prof. Dr.-Ing/ Univ. Tokio M. Buss. *Regelungssysteme 1*. Lehrstuhl für Steuerungs- und Regelungstechnik, München, 2015.

[Ver] Verband der Automobilindustrie. Automatisierung: Von fahrerassistenzsystemen zum automatisierten fahren.

[W.] W. Fastenmeier. Mensch, maschine, umwelt.