UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Tècnica Superior d'Enginyeria
de Telecomunicació de Barcelona

SOFTWARE ARCHITECTURAL DESIGN FOR SAFETY IN AUTOMATED PARKING SYSTEM

A Master's Thesis

Submitted to the Faculty of the

Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona

Universitat Politècnica de Catalunya

by

Roger Tremosa Hernández

In partial fulfilment

of the requirements for the degree of

MASTER IN ELECTRONIC ENGINEERING

Advisor: Dr.-Ing. Núria Mata & Dr.-Ing. Luis Javier de la Cruz Llopis

Barcelona, June 2022

Title of the thesis: Software Architectural Design for Safety in Automated Parking System

Author: Roger Tremosa Hernández

Advisor: Dr.-Ing. Núria Mata & Dr.-Ing. Luis Javier de la Cruz Llopis

Abstract

The automotive industry has seen a revolution brought about by self-driving cars. However, one of the main challenges facing autonomous driving systems is ensuring safety in the absence of a supervising driver and verifying safe vehicle behaviour under various circumstances.

Autonomous Driving Systems (ADS), due to their complexity, cannot be solved straightforwardly without proper structure. Thus, they need a well-defined architecture to guide their development with requirements that involve modularity, scalability, and maintainability among other properties.

To help overcome some of the challenges, this master thesis defines and implements in a simulated environment an automated parking system that complies with industrial and safety standards. The work has been divided into four parts. Firstly, the safety rules for the development of an autonomous function have been analysed. Secondly, the use cases and system requirements have been defined following the needs of the automated parking system. Thirdly, the system has been implemented in the simulation environment with a structure based on a widely adopted automotive standard. The final result is the software architecture of an autonomous vehicle with automated parking functionality. This concept has been validated within the virtual environment together with the integration of the AUTOSAR runtime environment, which the communication between components and mode switching functionality in the CARLA simulation environment.

The result of this project shows the benefit of integrating architecture and simulation, thus easing the development and testing of future autonomous systems.

Key words: Autonomous Driving Systems, automated parking, safety standards, CARLA simulator.

Acknowledgements

First and foremost, I would like to express my gratitude to Dr.-Ing. Núria Mata, who trusted me from the beginning and gave me the opportunity to get involved in a completely new world for me, such as autonomous driving. Thank you for your patience, your words of encouragement and your enthusiasm for every achievement along the way. I am glad to know that we still have a long way to explore together as we work to make autonomous system safer.

I would also like to thank Joao-Vitor Zacchi for his continuous support during the whole period assisting me and for all the ideas provided to solve the problems that have come up.

Special thanks to Dr.-Ing. Luis Javier de la Cruz Llopis for directing the thesis from Barcelona and for being always available for any need that arose during the development of the thesis.

I want to thank the Fraunhofer Institute for Cognitive Systems IKS for the opportunity to write this master thesis in the Cognitive Software Systems Engineering Department and the "Bayerisches Staatsministerium für Wirtschaft, Landesentwicklung und Energie" for the funding under the project "Unterstützung des thematischen Aufbaus des Instituts für Softwaretechnik Kognitive Systeme".

Last but not least, I want to express my gratitude to my new ERASMUS friends and my family for making me feel at home while I have been living away from it. Special thanks Yassine El Kabdani, who has been working side by side with me in the development of this project and its implementation.

Revision history and approval record

Revision	Date	Purpose
0	01/04/2022	Document creation
1	08/06/2022	Document revision
2	27/06/2022	Document revision
3	01/07/2022	Document revision

Written by:		Reviewed and approved by:	
Date	20/06/2022	Date	01/07/2022
Name	Roger Tremosa Hernandez	Name	Núria Mata Burgarolas & Luis Javier de la Cruz Llopis
Position	Project Author	Position	Project Supervisor

Table of contents

List of Figures

List of Tables

1 Preface

Open systems (of systems) run and interact in a physical world with unforeseeable uncertainties. Cognitive systems are software-intensive technical systems that imitate cognitive capabilities of human behaviour by processing the environment data, predicting upcoming changes, adapting to the context, while ensuring that safety is preserved.

Fraunhofer Institute for Cognitive Systems IKS researches and develops methods and technologies that enable intelligent, autonomous systems to respond reliably and safely to unexpected or previously unknown situations. In doing so, they work at the interface between science and industry to bring innovative concepts for cognitive systems into practical application[1].

1.1 PreMotivation

My short professional experience in a consulting company working in software architecture and at Ingenia (Novanta group) developing servo motors for robotics have been ideal as a basis to awaken my curiosity for the automotive world. I have never had the pleasure of working directly with automotive engineering and even less with autonomous driving, except for an optional subject in the master's degree. Directly thanks to this subject I had the chance to enjoy the opportunity to develop this thesis at Fraunhofer IKS.

My main motivation for this master thesis has been to deepen into the automotive industry and to understand what approaches are being used to develop an autonomous vehicle. Along with this, to learn how to manage the development of vehicle systems using a systems engineering approach in order to improve design quality, robustness and resource optimisation.

1.2 Previous requirements

This thesis focuses on the design of engineering systems and autonomous driving systems. No special academic qualifications are required, but a background in automotive, software and engineering are recommended.

2 Introduction

The project has been developed at the Fraunhofer Institute of Cognitive Systems, specifically in the department of Cognitive Software Systems Engineering, where one of the research focus is the modelling and research on autonomous vehicle systems.

In the automotive industry, research and development departments are increasingly focusing on the concepts for autonomous driving systems. It is true that many companies already have their first own autonomous functions in series vehicles and that this trend is increasing constantly, however there is still a long way to go to achieve full autonomy for all possible known and unknown environments. Understanding the needs of ADS is the first step towards developing a new function/system for an autonomous vehicle.

The results of a survey show that about 10,000 traffic accidents occur due to entering or leaving parking spots in 12 million traffic accidents[2]. In fact, the actual number of accidents is much more than the number of this traffic report. Because of the large number of traffic accidents and requirements of drivers, automatic parking system draws an infinite attention in the automotive engineering domain.

Regarding the above statistics, an automatic parking system can optimise the density of traffic and the distribution of parked vehicles on the roads.

2.1 Goals

The main goal of this thesis is to define, develop and implement in a simulated environment an automated parking system fulfilling industry and safety standards. The software architecture of the Automated Parking System (APS) developed in this work is based on the guidelines proposed in the ‘Road vehicles — Safety and cybersecurity for automated driving systems — Design, verification and validation’, ISO/TR 4804, 2020 [3].

This project has been developed simultaneously with another master thesis focusing on the design and implementation of the autonomous system (AS) mode manager component [4]. The software architecture has been modelled in AUTOSAR using the mode management methodology described in the AUTOSAR standard. The software architecture has been instantiated for the APS function developed in this master thesis.

The final result is the software architecture of an autonomous vehicle with automated parking functionality. The concept has been validated in a virtual environment. The generated AUTOSAR run-time environment (RTE) has been integrated with the environment and the autonomous vehicle implemented by the CARLA simulator.

2.2 Scope

The scope of this work is to learn how to specify and model automated parking systems using first a system engineering classical approach, and second established industry safety standards to model the system architecture. Then, the integration an implementation of the

designed system inside a simulation environment proceeds. It is not in the scope of this master thesis the implementation of the AUTOSAR run-time environment.

Chapter 3 provides an overview of all relevant topics needed to develop this thesis, namely systems engineering, safety standards, automated parking methodologies and simulation environments. Chapter 4 focuses on the description of the proposed automated parking system and on its development. The results of the project and the integration with the run-time environment are presented in chapter 5. The breakdown of the costs is specified in the budged in chapter 6. An overview of the benefits inside environment and social impact are featured in chapter 7. Finally, the conclusions and future works can be seen in chapter 8.

3 State of the art of the technology

Our first step has been to review the current state of the art of the research areas that we will be working on. The topics we cover in this section include system engineering, automotive safety standards, parking manoeuvres for automated vehicles, and simulation environments.

3.1 Systems Engineering

Systems engineering is about studying and understanding reality as it is with the aim of optimizing and improving complex systems. It can be applied to any type of system; it is not committed to a concrete field. For example, it can be applied for studying the human digestive system or an informatics system, as an example of two different fields without anything in common[5].

Systems Engineering provides facilitation, guidance and leadership to integrate the relevant disciplines and specialty groups into a cohesive effort, forming an appropriately structured development process that proceeds from concept to production, operation, evolution and eventual disposal[6].

In this thesis, we are going to focus on electronic systems, more specifically on automotive embedded systems and their control in terms of standards for the automated parking function.

During system design there is always implicitly described all the information that is needed, for example: what functionality the system provides, how the system interacts, how many components are needed to realise our system, how the interaction between components is, what part of the functionality each component implements, etc. This set of information allows us to draw the hierarchy of the system, not only at the top level, and to detail the requirements to fulfil the total implementation.

Developing a project based on a good system architecture provides a robust foundation. This implies that all the relationships with the different factors that make up the system are well defined and connected to each other. It allows all team members to know what, where and how things have to be done in a clear and simple way. In addition, everyone uses the same language, which improves communication and project management tasks, reducing the chance of failure.

Systems engineering allows a challenging large system to be decomposed into smaller and easier-to-solve subsystems that can be classified and prioritised. The main benefits of it are increasing the performance of the overall system, reducing hidden project costs, improving the quality of the system platform, and allowing the system to be upgraded and expanded in a quick and easy way.

Systems engineering is also an iterative process. First, it is sufficient to define the project architecturally. Subsequently, at the component level, with the further refinement of

requirements, communication and implementation details, the design gets more detailed. All in all, systems engineering provides a guided path for the successful design and implementation of an engineering project.

In this report we will clearly see how we use this type of engineering for our development and its division into small subsystems.

3.2 Safety standards for automated systems

Before proceeding with the development of the architecture for automated driving systems, it is convenient to perform an overview of the different standards that currently exist and could be of interest to guarantee safety, which is one of the main objectives of this work.

Currently there are several standards for safety applicable to the automated parking function. For the development of this project the following have been considered:

- V-Model
- SAE Automated driving levels
- ISO 26262
- ISO / TR 4804
- ASAM

The subsequent sections briefly explain each of them and what topics they intend to standardize.

3.2.1 V-Model

The V-Model is an approach model that was developed by commissioning of the State of Germany for planning and implementing system development projects. It considers the entire lifecycle of a system nicely fitting the line of thinking in systems engineering[7]. The V-model is a graphical representation of a system's development lifecycle, and it is used to produce rigorous development lifecycle models and project management models. It is also known as Verification and Validation model. A graphical description of the V-Model is provided in Figure 1.

The left side of the "V", called design phase, represents the decomposition of requirements, and creation of system specifications. Just in the middle of the process, at the vertex of the "V", we have the implementation of the project to be developed. When this phase is completed, we continue our way upwards where the test plans are put into action. The right side of the "V", called testing phase, represents the integration of each design part and their validation.

The main advantages of this model are: simple and easy to use, planning and designing activities happens well before implementation, saves a spot of time (hence higher chance of success over the waterfall model), proactive defect and avoids the downward flow of the defects. The disadvantages are very rigid and lack of flexibility. Also, the project is developed during the implementation phase and no early prototypes are produced. if any

changes occur midway, then the requirements and test documents need to be updated.

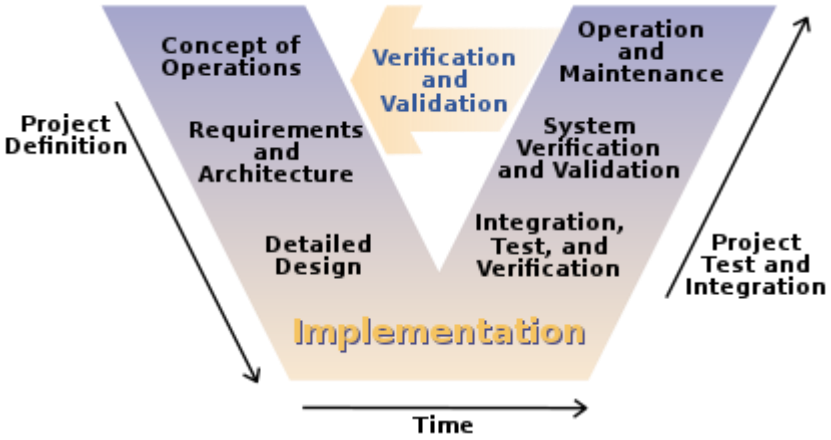

Figure 1. The V-model of the systems engineering process[8].

In this thesis we follow this model, the project flow is easily recognisable and comparable with the V-Model.

3.2.2 SAE Automated driving levels

As described by the Society of Automotive Engineers (SAE), transferring total control from humans to machines is a step-by-step process on a scale of 0 to 5 levels. Level 0 means no automation and level 5 means full-time performance by an automated driving system under all road and environmental conditions.

The standard SAE J3016 [9] defines the multiple levels of driving automation. The Standard provides descriptive and broad information about this evolution but does not provide strict requirements.

SAE classifications are designed to clarify the role of a human driver, if any, during vehicle operation. An environmental monitoring agent is the first discriminant condition. For levels 0-2 of automation, a human driver monitors the environment; while for levels 3-5 of automation, the vehicle monitors the environment.

As another discriminant criterion, dynamic driving task (DDT) fallback mechanisms are also taken into consideration. Intelligent driving automation systems (levels 4-5) embed the responsibility for automation fallback constrained or not by operational domains, while for low levels of automation (levels 0-3) a human driver is fully responsible. Figure 2 shows the remaining classification factors used to define each level [9], [10].

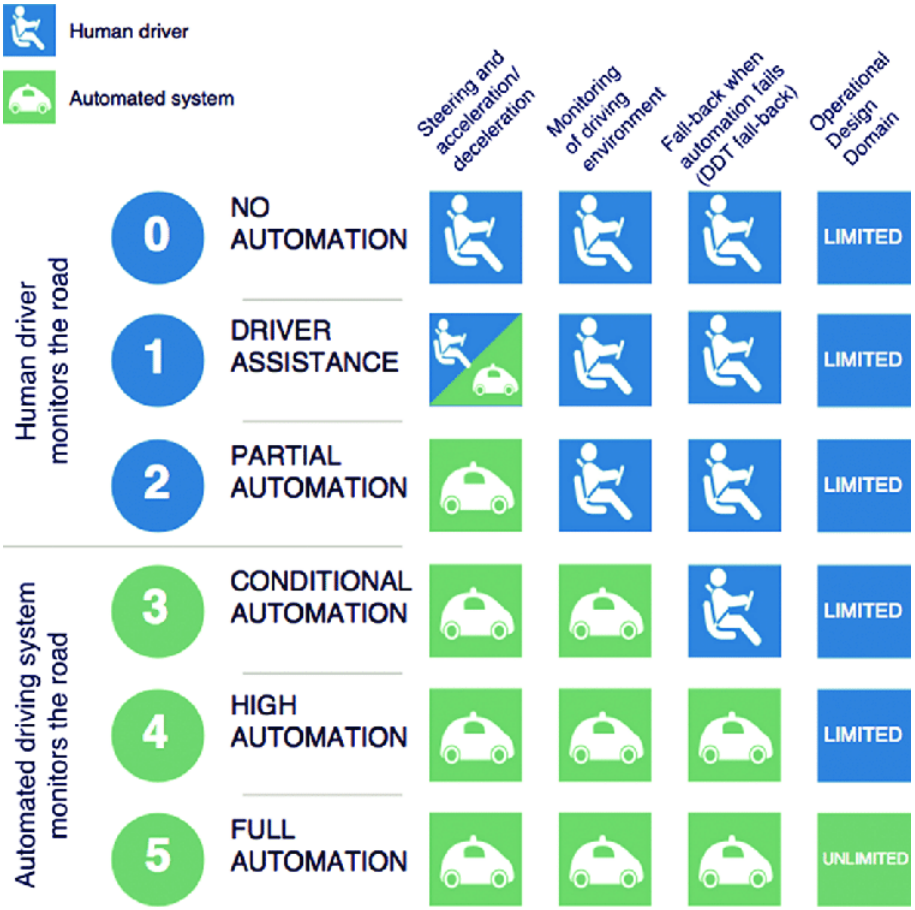

Figure 2. Automated driving levels[10].

3.2.3 ISO 26262

The ISO 26262 titled “Road vehicles – Functional safety” is a standard based on the functional safety for the electrical and electronics systems that the vehicles have installed. The scope of this standard was extended for passenger cars to all road vehicles except mopeds[11].

First edition of ISO 26262:2011 is based on current knowledge of automotive systems (such as steering, braking, airbag systems, etc.). However, it does not fully address very complex, distributed systems or how to meet availability requirements. It is necessary to interpret the second edition further in order to resolve some of these issues.

The aim of functional safety is to prevent accidents or component failures as a result of changes in inputs, hardware or environmental conditions. Currently, it is unclear how autonomous vehicles shall behave if an accident cannot be avoided, or how risks can be minimized[10].

In the ISO 26262 functional safety applies only to the static context in which the system is developed. As this standard is only applicable when the environment conditions are well known, it is not applicable to autonomous systems because the environment is no longer static. It is impossible to study all the possible scenarios. In this way, the standard cannot

ensure a total functional safety of autonomous systems, since it cannot know in detail the conditions they will face.

This is address in the ISO PAS 8800 “Road vehicles – Safety and artificial intelligence” [12], currently under development that will be publicly available specification in late 2023.

3.2.4 ISO/TR 4804

This ISO/TR document titled "Road vehicles - Safety and cybersecurity - Design, verification, and validation" [3] outlines a framework for developing, verifying, validating, producing, and operating automated driving systems that are focused on safety and cybersecurity. According to SAE J3016:2018[9], the technical report discusses verification and validation methods for automated driving systems that focus on levels 3 and 4 (or lower if necessary).

This technical report aims to provide a generic strategy for addressing the risks associated with automated vehicles. It is possible to use this basic approach as a starting point for safe automated driving, but it does not describe a complete and safe product.

This standard supplements existing safety standards and publications. This document offers a more technical overview of recommendations, guidance, and strategies for avoiding unreasonable risk and cybersecurity related threats, emphasizing the importance of safety by design[3].

An automated driving safety and cybersecurity framework is presented, along with recommendations for developing, verifying, validating, producing, and operating the systems. Automotive and transportation industry stakeholders can benefit from it.

3.2.5 ASAM

The Association for Standardization of Automation and Measuring Systems, ASAM in short, is an incorporated association under German law. Its members are primarily international car manufacturers, suppliers and engineering service providers from the automotive industry. The association coordinates the development of technical standards, which are developed by working groups composed of experts from its member companies[13].

The relevant standard for this project is the OpenODD standard which is defined in the ASAM Simulation domain. This standard is not finalized; it is still a concept that will serve as a basis for the development of the future standard. The main objective is to provide a format capable of representing the Operational Design Domains (ODD) defined for Connected Automated Vehicles (CAVs) for simulation based testing[14].

An ODD must be valid for the entire lifecycle of the vehicle, as it is part of its safety and operating concept. The ODD chosen for the system greatly impacts the design of that function, both its capabilities and its respective validation. ODD is basically used to specify the functionality of connected automated vehicles, specifically the environment in which the CAV must be able to operate. The environment includes all traffic participants, the weather

conditions, the infrastructure, the location, the time of day and everything else that has an impact in one way or another on automated driving. An ODD always defines a well-closed and bounded region of the entire ontology, as seen in the Figure 3.

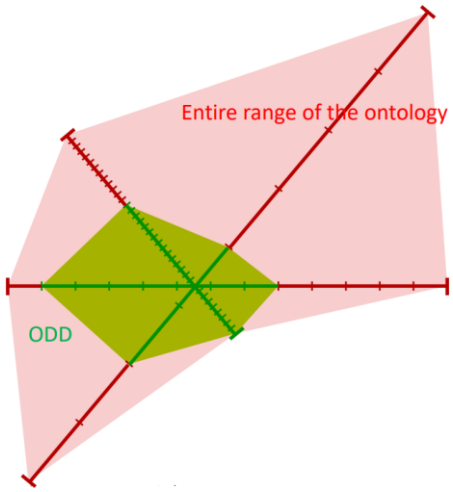

Figure 3. Range of the ontology[14].

In this context, a standard to regulate and define all possible ODDs is necessary. This is where the ASAM OpenODD concept project initiative arises, which is being developed by all relevant automotive stakeholders. The ODD should be represented in a way that can be easily used for simulation and different machine processing environments. In order to be able to use an abstract ODD description of the vehicle for simulations and post-processing, the format shall meet the following requirements: searchability, exchangeability, extensibility, machine readability, measurability and verification and finally human readability[9].

A formal definition of ODD is found in the standard SAE J3016 (2018) which states that “Operating conditions under which a given driving automation system or feature thereof is specifically designed to function, including, but not limited to, environmental, geographical, and time-of-day restrictions, and/or the requisite presence or absence of certain traffic or roadway characteristics"[9].

ASAM’s philosophy is not to do something that another standards organization is doing or to define new standards that contradicts the current ones, but only to fill the gaps that currently exists. In this case, the ASAM OpenODD standardization aims to complement the activities of BSI (BSI PAS 1883, which provides the taxonomy for ODD) and ISO (ISO 34503 which uses the taxonomy to provide a high-level definition format for ODD). All three standards are in contact to avoid contradictions.

The standard provides all possible attributes of the ODD that may be required by an automated driving system, and also gives different examples of what the taxonomy should look like.

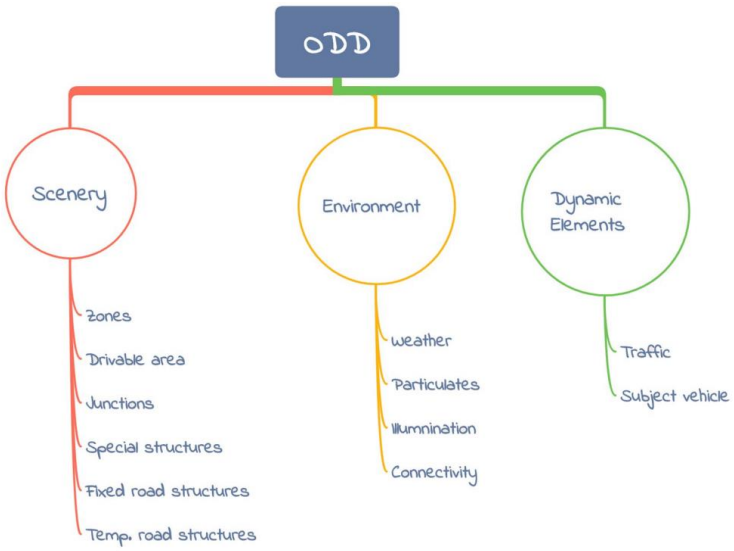

Figure 4. ODD taxonomy according to BSI PAS 1883[14].

3.3 Automated parking

In this section we focus on the factors necessary to execute a parking manoeuvre. The essential topic for a manoeuvre is the path planning. In what follows, we will discuss how this is done and different implementation approaches.

Path planning is the means by which autonomous vehicles plan their movements and navigate through the environment. As stated in [15], there are multiple challenges in planning an autonomous vehicle's path through a dynamic environment:

- Build a map based on offline coordinates that provides a basis for the vehicle's real-world position and planned trajectory.
- Locate the vehicle's current position on the map and plan a short-term trajectory through these points. There may be multiple candidate points for the vehicle's next step. The best candidate should be decided based on the positions of obstacles detected by the vehicle sensors.
- Find the best vehicle heading and acceleration to ensure a safe trajectory, possibly also considering comfort (favouring smoother trajectories with less abrupt accelerations).

Path planning is an important subject to be considered in the automatic parking system. According to the acquisition of environmental information around the vehicle, trajectory planning can be divided into two categories[2]. Firstly, a global trajectory planning method based on known environmental information. Second, a local path planning method that is a key task for the motion planners of autonomous vehicles since it commands the vehicle across its environment while avoiding any obstacles. To perform this task, the local path planner generates a trajectory and a velocity profile, which are then sent to the vehicle's actuators[16].

For the scope of our project, we develop a local planner. Typical parking control theories have been evaluated as fuzzy control, neural network control, linear inequalities-based control method, etc. Also, the typically proposed types of path curves such as circular arcs with straight lines, clothoid curves, arc tangent curves, and cubic polynomial curves. The triangular function was also used by some researchers.

Though other geometries of the parking path may be calculated based on simpler mathematical equations, a fifth-degree polynomial curve with continuous curvatures is going to satisfy the path's smoothness and flexibility better. Since more different variables regarding initial and final conditions are involved in the calculation, it is better to choose a higher degree curve. A fifth degree polynomial is the least polynomial curve that can present parking motion[17].

For the reason explained above, we follow the quintic polynomial planning to generate the trajectory that will allow us to park forwards and park backwards. Of course, for each type of parking, it is not the same type of trajectory/driving and must be adapted by adjusting some parameters. After that, we will follow an optimal trajectory in a Frenet frame to recalculate during run-time the path to avoid obstacles.

3.3.1 Quintic polynomial path planning

The method presented in [17] and [18] has been chosen for our forwards path planning in the parking manoeuvre simulation. This method is based on the idea of quintic control function [19].

If you use two quintic polynomials along x axis and y axis, you can plan for two dimensional robot motion in x-y plane:

$$x(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5$$

$$y(t) = b_0 + b_1 t + b_2 t^2 + b_3 t^3 + b_4 t^4 + b_5 t^5$$

Planning the local path is to determine the coefficients of the fifth polynomials, $a_0 \sim a_5, b_0 \sim b_5$, by solving six simultaneous equations satisfying boundary conditions, see below:

$$\begin{aligned} x(t=0) &= X_s \\ \dot{x}(t=0) &= V_{xs} \\ \ddot{x}(t=0) &= A_{xs} \\ x(t=T) &= X_e \\ \dot{x}(t=T) &= V_{xe} \\ \ddot{x}(t=T) &= A_{xe} \end{aligned} \qquad \begin{aligned} y(t=0) &= Y_s \\ \dot{y}(t=0) &= V_{ys} \\ \ddot{y}(t=0) &= A_{ys} \\ y(t=T) &= Y_e \\ \dot{y}(t=T) &= V_{ye} \\ \ddot{y}(t=T) &= A_{ye} \end{aligned}$$

Where the boundary conditions are:

- T: the time interval traveling along the path.
- X_s, Y_s: the position components at t = 0.

- V_{xs}, V_{ys}: the velocity components at t = 0.
- A_{xs}, A_{ys}: the acceleration components at t = 0.
- X_e, Y_e: the position components at t = T.
- V_{xe}, V_{ye}: the velocity components at t = T.
- A_{xe}, A_{ye}: the acceleration components at t = T.

Based on the assumptions of the [18], the boundary conditions of initial time (t = 0) can be obtained, e.g. the current localization and motion values of the vehicle. The values of velocity and acceleration at the end of the path are predefined (t = T). Parameter T represents the time required for automated guided vehicle to travel along the local path. This parameter is inversely related to acceleration. Too much acceleration could take away from the comfort of the manoeuvre, therefore the maximum acceleration is the one that sets the minimum route time (T).

The calculations of interest for the travelling along the local path, after x(t) and y(t), are the velocity (v_s) and the desired curvature (k):

$$v_s = \sqrt[2]{\dot{x}^2 + \dot{y}^2}$$

$$k = \frac{a_d}{v_s^2}$$

$$where\ \ a_d = \left| a - \frac{v}{|v|}\left(a\frac{v}{|v|}\right)\right|;\quad where\ v, a\ are\ vectors\ from\ (x, y)$$

With these parameters it is then also possible to calculate the rotational speed of the wheels and the steering.

Going into more detail, this method has a restraining trajectory error method implemented. In practice, the trajectory gets errors due to calculations caused by wheel slippage, the actual non-flat shape of the ground, etc. and is compensated at each driving calculation iteration in real time, see Figure 6.

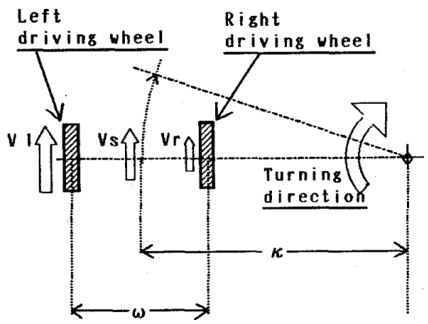

Figure 5. Calculation for traveling[18].

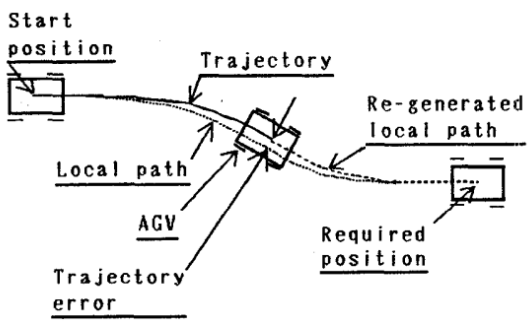

Figure 6. Restraining the trajectory error[18].

3.3.2 Optimal trajectory in a Frenet frame

The method presented in [20] has been chosen for recalculate during runtime the path to avoid obstacles.

The Frenet Frame method, which asserts invariant tracking performance under the action of the special Euclidean group $SE(2) \coloneqq SO(2) \times \mathbb{R}^2$, is a well-known approach in tracking control theory. This method is used to combine different lateral and longitudinal cost functionals for diverse tasks, as well as to simulate human-like highway driving behaviour.

The trajectory type that this control follows has many possibilities but, in our case, it will be a quintic polynomial too (explained in the previous section).

First, we must know the change of coordinates in order to understand the behaviour. It is a change from Cartesian coordinates to Frenet coordinates, see images Figure 7 and Figure 8.

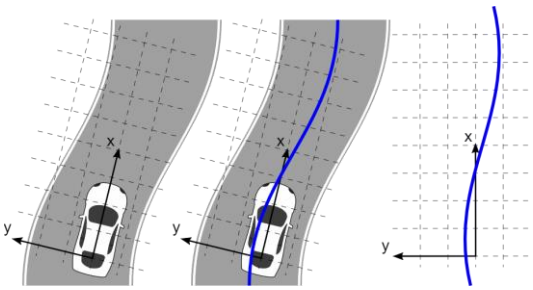

Figure 7. Cartesian coordinates.

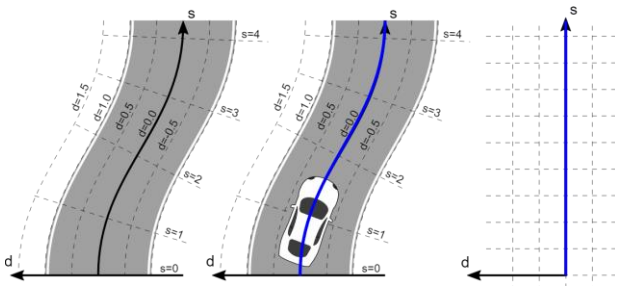

Figure 8. Frenet coordinates.

The next step is to determine de trajectory start state. Once this point and the main trajectory are known, several trajectories are generated within the maximum range in the d-axis at each run-time and in a limit range in the s-axis. These trajectories have a cost calculation referenced according to the lateral trajectory (controlling the steering) and longitudinal trajectory (controlling the speed) considering physical magnitudes. This cost has the objective of finding the most comfortable and beneficial trajectory. Then, static and dynamic collision checking takes place. Each set of trajectories is evaluated, and their total costs are modified according to whether they avoid static and dynamic collisions. After that, the collision-free trajectory with the lowest conjoint cost functionals of each active trajectory is compared to the other ones, and the trajectory with the smallest initial jerk value is finally put through to the tracking controller. Finally, the coordinates are change to the cartesian initial ones. Further mathematical explanations in [20] and [21].

The example below shows a plot of an optimal trajectory. The crosses are the obstacles, the blue line is the main path, the green lines are the possible paths, the red lines are the no collision-free paths, and the blue circles are the waypoints of the selected path. All this in an instant of time t.

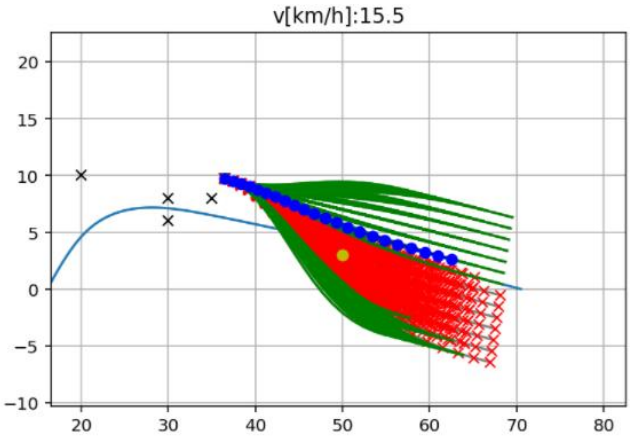

Figure 9. PythonRobotics, optimal trajectory with Frenet frame example in cartesian coordinates.

3.4 Simulation environment and virtual integration

As an emerging technology, automated driving has been the subject of major research efforts over the past decades. Planning is an essential topic in the field of automated driving. It is a critical part of realising driving autonomy, embedded with perception and execution within the system architecture. It requires different software tools for its development, validation and execution.

In this section of the state-of-the art we focus on different simulation software environments based on the classical sense-plan-act paradigm from robotics.

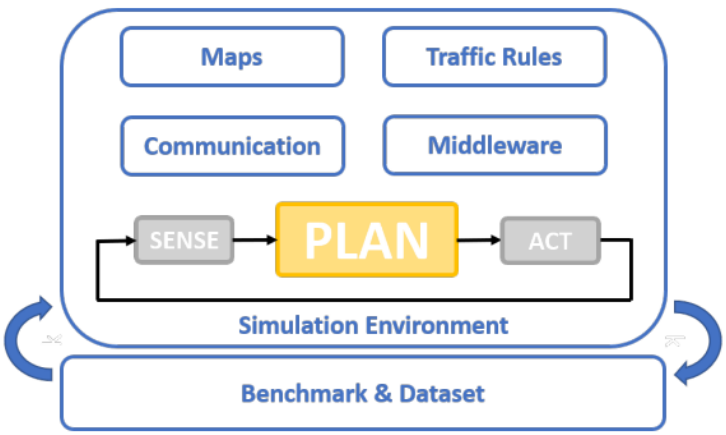

Figure 10. Tools supporting planning[22].

A simulator must provide multiple information for the planning stack: road network, traffic data for route planning, sufficient data for perception to generate a scenario for behaviour planning, and environmental features for trajectory planning. As an automated driving system is a highly complex and coupled system, sensor data is also needed for sense and act. See Table 1 for the comparison between different simulator software's than complain with.

Table 1. Comparison of simulators supporting planning methodology[23].

Simulator	Group	V2X	TF	DM	SE	VI	VD
Carla [75]	graphics	i	+	+	+	+	+
Cognata [76]	graphics	o	+	+	+	++	o
LGSVL [77]	graphics	i	+	+	+	+	+
Gazebo [78]	robotics	o	o	o	+	o	+
USARSim [79]	robotics	o	o	o	+	-	o
AirSim [80]	robotics	o	o	o	+	+	+
MORSE [81]	robotics	o	o	o	+	-	o
TORCS [82]	racing	o	o	+	o	o	+
SynCity [83]	AD	o	o	o	++	++	+
PreScan [84]	AD	++	+	+	++	-	o
Righthook [85]	AD	o	+	++	+	+	+
SCANeR [86]	AD	+	+	+	+	+	+
VTD [87]	AD	i	+	+	++	++	+
Autono Vi-Sim [88]	AD	o	+	-	+	+	+
rFpro [89]	AD	o	i	i	+	+	++
Vissim [90]	traffic	i	++	+	--	--	-
Sumo [59]	traffic	i	++	+	--	--	--
Aimsun [91]	traffic	i	++	+	--	--	--
CarMaker [92]	VD	++	+	+	+	+	++

About the Table1: open-source software's in bold, the symbols means: (--) very poor, (-) poor, (o) irrelevant, (+) good, (++) very good, (i) some efforts to implement, TF – Traffic flow simulation, DM – Driver model for non-ego objectives, SE – detail and variety of sensors, VI – detail of the rendered graphics, VD – detail of vehicle dynamics.

For this thesis, the simulator chosen is CARLA simulator. The main reason for choosing this simulator is that it is open source software. Furthermore, based on the above comparison and looking at all open source simulators, we can easily see that CARLA performs well in all specifications. It is true that it is not suitable for V2X, but we will not need it in our thesis. A more detailed explanation of the software can be found in appendix A.

4 Development

The case study of this master thesis is an Automated Parking System (for now on **APS**), which provides automated parking and pick up functionalities to the users (Park now and Get car, respectably in section 4.1). In contrast to parking assistance systems, where the driver is only supported by the system, but it is still responsible for the parking manoeuvre, APS is a fully automated function with SAE autonomous level 4.

The focus of this work is to design a complete automatic parking system using a systems engineering approach. We define the vehicle components, the environment and system management components. Subsequently, simulate and recreate this function in the CARLA simulator.

The Automated Parking System functionality has been modelled taking into account the different situations that our system will face. From a systems engineering point of view, this methodology starts with the definition of the main use cases, so that we identify the needs of the system. This guides us in differentiating the main operational modes and dependencies on the context that influence on the parking functionality and, therefore, have to be considered in the system architecture.

4.1 Overview and system under design

The automated parking system designed in this master thesis consists of the autonomous vehicle for parking management, where the functionalities for parking and unparking the vehicle are implemented. The components involved are the autonomous vehicle, a dedicated mobile app and an external cloud services. The external cloud service component is out of the scope in this thesis but is needed to behave the system. The autonomous vehicle and the dedicated mobile app are the system under design for the APS, see Figure 11. The system is further described in the next three sections where the use cases, constraints and system requirements are covered. As mentioned, the external cloud service is a part of this study but is not part of the system itself.

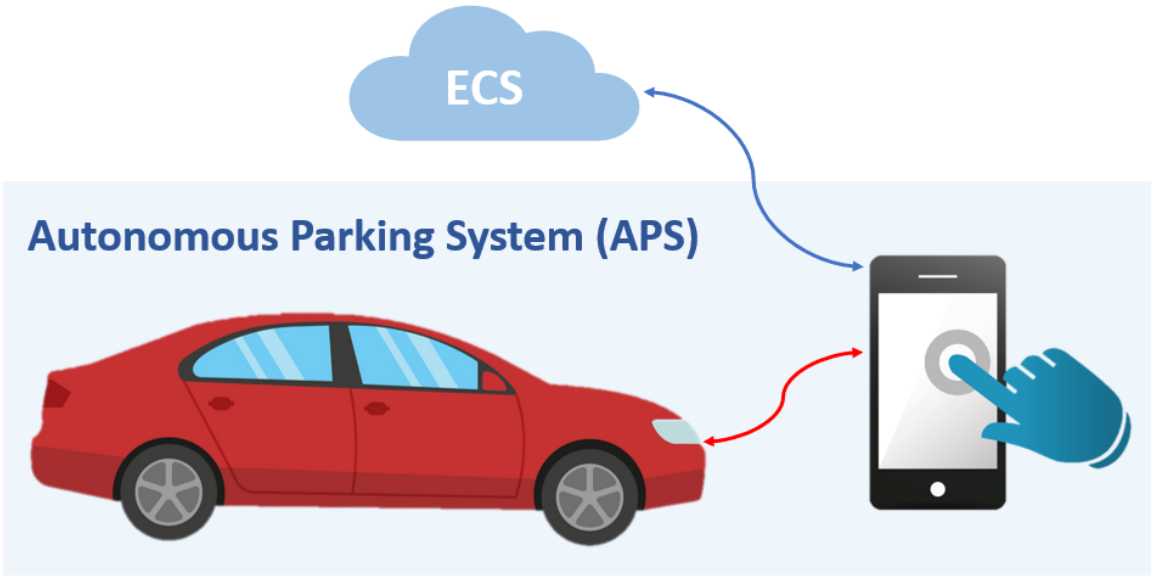

Figure 11. System under design.

We define briefly how the system under design works:

- APS parking functionality. When parking is requested, the user has to get out of the vehicle and through the mobile application (from now on APS App) search for the desired spot and activate the parking functionality.
- APS unparking functionality. When it is requested to pick up the vehicle, the user must activate the unparking functionality through the APS app indicating where he wants to receive the vehicle and when (it can be immediately).

In the Figure 12 we have a small simulation of the start of the APS parking functionality. The user gets out of the red vehicle and wants to park in the parking area that can be seen in the background of the image.

Figure 12. CARLA simulator in an example case.

4.1.1 User cases

As mentioned in the previous section, we use a systems engineering approach and start first defining use cases, see Table 2.

Table 2. Uses cases

USE-CASE	Description	Realized
UC-1	APS is an automated parking system for a vehicle that replaces the driver for the driving manoeuvre of the vehicle.	**SYS-4, SYS-6, SYS-11**
UC-2	APS supports the following parking manoeuvre: ○ Parallel parking ○ Angular parking ○ Perpendicular parking	**SYS-2, SYS-4, SYS-15**
UC-3	APS supports the following drivable area types: ○ Indoor parking ○ Outdoor parking ○ Urban road ○ Interurban road	**SYS-3, SYS-4, SYS-15**
UC-4	The user manages the APS function through a dedicated application on a mobile phone (APS App).	**SYS-1, SYS-5**
UC-5a	The APS App: - Shows the available parking space. - Reserves the parking space selected by the user. - Activates APS for parking when selecting “Park now” - Activates APS for unparking when selecting “Get car” - Allows to stop the APS function. - Activates “safe mode” if the parking manoeuvre has not been completed. Notifies if the result of the parking manoeuvre: completed or “safe mode”.	**SYS-1, SYS-5, SYS-7, SYS-10**
UC-5b	The APS App Communicates with the external cloud service (ECS). The ECS: - manages parking spots, including reservations - provides location of parking spots	
UC-6	The APS App shows the following information: ○ The status of the function: active or inactive. ○ The menus to select the parking spot. ○ The notifications about the APS status: parked, safe mode or stop.	**SYS-1, SYS-5, SYS-7, SYS-10**
UC-7	The APS App offers a "Park now" button to start the vehicle park function. This function allows the user to select the desired parking spot and start the manoeuvre.	**SYS-10**
UC-8	The user selects the parking spot and accepts the parking spot to start the parking manoeuvre of the vehicle. The APS function will not proceed if the user does not select any spot or rejects the selection.	**SYS-3, SYS-10**
UC-9	If there is no parking space in the area, the user will not be able to select any parking spot, and therefore will not be able to start the APS functionality. In this case the vehicle will never take autonomous control.	**SYS-1**

UC-10	The APS App offers a "Get car" button to start the vehicle unpark function. This function allows the user to set the meeting point and pick up time.	**SYS-10**
UC-11	The ECS provides the exact address and coordinates of the parking spot to the vehicle. Both parking or pick up locations are displayed and recorded in the APS App.	
UC-12	APS drives to the parking spot provided by the APS App.	**SYS-1, SYS-5, SYS-8, SYS-9, SYS-11, SYS-12**
UC-13	The user is liable for persons and objects left inside the vehicle. The APS function assumes that there are no users inside the vehicle, but it is not their responsibility to comply with it.	
UC-14	When the APS detects an obstacle that prevents it from finishing parking (e.g., a pedestrian walking in the spot) it will not complete its manoeuvre until the obstacle leaves the space.	**SYS-6, SYS-9, SYS-12**
UC-15	APS goes to “safe mode” manoeuvring if any of the following situations happen during the parking manoeuvre: ○ Collision risk with any element outside the vehicle. ○ The user stops the manoeuvre through the APS App to abort the APS. Other risks are excluded and referenced as constrains.	**SYS-4, SYS-6, SYS-8, SYS-9, SYS-12**
UC-16	APS finishes the parking manoeuvre when the vehicle is completely parked in the parking spot selected by the user.	**SYS-1, SYS-5, SYS-7**
UC-17	APS locks the vehicle whenever the system is active, or the vehicle is parked.	**SYS-14**
UC-18	APS announces the parking manoeuvre successfully finished by blinking the emergency lights and sending a notification through the APS App. This notification contains a confirmation and the coordinates of the parked vehicle.	**SYS-1, SYS-5, SYS-7**

4.1.2 Constrains

For simplicity and satisfaction on safety demands, this work assumes the following constrains:

Table 3. Case study constrains

CONSTRAINS	Description	Realized
CON-1	Misuse of the APS function is not considered.	**UC-1, UC-7, UC-10, UC-12, UC-13, UC-17**
CON-2	Technical failures of the vehicle, sensors, actuators are not to be considered during the parking manoeuvre.	**UC-12, UC-15**
CON-3	The vehicle is able to fulfil the parking manoeuvre and return to the initial location with sufficient amount of energy.	**UC-12**
CON-4	Indoor and outdoor parking areas only provide perpendicular parking spots.”	**UC-2**
CON-5	The parking floor for street is 0.	

CON-6	Failures in the connection between the vehicle and the mobile app are not considered.	**UC-6, UC-15**
CON-7	The ECS providing free parking spots is always available.	**UC-5a, UC-5b, UC-9**
CON-8	The ECS reliably provides parking spaces.	**UC-8, UC-9, UC-11**
CON-9	Once a parking spot has been selected, it is reserved and will not be offered to another vehicle.	**UC-5, UC-8, UC-14, UC-16**
CON-10	The vehicle always fits in the assigned parking spot. The vehicle has standardised measurements in order to be able to be compared with parking spaces.	**UC-5a, UC-5b, UC-11 UC-16**
CON-11	The user must be reachable at any time during APS operation, starting when the user selects "Park now" / "Get car" in the APS App until received the terminated notification.	**UC-9, UC-15, UC-18**
CON-12	The drivable areas must have traffic lane type for identifying the parking area spot. The APS can't park in a scenario where the surroundings cannot be recognised (e.g., a land car park).	**UC-3**
CON-13	The APS must respect the traffic rules (e.g., speed limit).	

4.1.3 System requirements

The full functionality must be guaranteed. The following requirements shall be fulfilled by the system under design, which, as is known, is composed of the autonomous vehicle and the mobile application.

Table 4. System requirements.

SYS. REQUIREMENTS	Description
SYS-1	The system shall provide the state of the parking manoeuvre to the APS App.
SYS-2	The system shall support the following parking modes: ○ Parallel mode ○ Angular mode ○ Perpendicular mode
SYS-3	The system shall be able to drive to the location provided by the APS App.
SYS-4	The system shall calculate and manage all possible routes in real time. If needed, the system aborts the manoeuvre when high risk detected.
SYS-5	The system shall communicate its status to the APS App in real time.
SYS-6	The system is the only one that can change the route of the parking manoeuvre once started the APS.
SYS-7	The system shall send a parking manoeuvre completed to the APS App to notify that the vehicle is parked correctly and safe.
SYS-8	The APS vehicle automatically recognises obstacles while manoeuvring into or exiting a parking spot.
SYS-9	The APS must avoid collisions with any dynamic or stationary object while manoeuvring into or exiting a parking spot.
SYS-10	The APS app shall provide: ○ Localization of the spot ○ Park or unpark mode

	○ Type of parking mode ○ Angle of parking, in angular parking mode case
SYS-11	The parking speed is limited to 10 km/h forward and backwards. the speed can be decreased up to 5 km/h in the last metres of the manoeuvre. However, this speed limit must conform to local regulatory requirements, such as internal law and technical guidance.
SYS-12	The APS shall abort if any collision occurs.
SYS-13	The APS shall abort if the user stops the function via APS App and follow the new route provided by the APS App.
SYS-14	APS shall lock the vehicle whenever the system is active, or the vehicle is parked.
SYS-15	The APS App shall manage the parking location modes specified below: - Street_parallel (spot on the street) - Street_angular (spot on the street) - Parking_outdoor (spot inside car park) - Parking_indoor (spot inside car park) - Parking_road (spot on the road)

4.2 Modelling of the APS function

The system needs to be modelled to fulfil the use cases and system requirements described above. Considering that the vehicle is autonomous and the APS function safety relevant, we model our function making use of the industry standards described in section 3.2. The ISO TR-4804[23] is used for the modelling of the logical architecture, whereas the ODD's is based on the ASAM specification.

4.2.1 Logical architecture for safety

The ISO technical report 4804 deals with safety for autonomous driving systems[3]. It defines that the overall system is considered safe according to its capabilities.

During the time the system is nominally operating, the performance of the system can be understood using the classical sense-plan-act design paradigm of the robotics and automation literature. In this model, sensing and perception (including localization), planning and control, and actuation and stability provide a general, implementation-independent view of the automated driving system. Figure 13 shows it at a very high level.

Figure 13. Sense - plan - act design paradigm.

The capabilities are divided into fail-safe capabilities (FS) and fail-degraded capabilities (FD). Based on the allocation of capabilities to the basic functions for sense – plan – act, it is possible to allocate requirements for elements that the automated vehicle is reasonably safe as depicted in Figure 14.

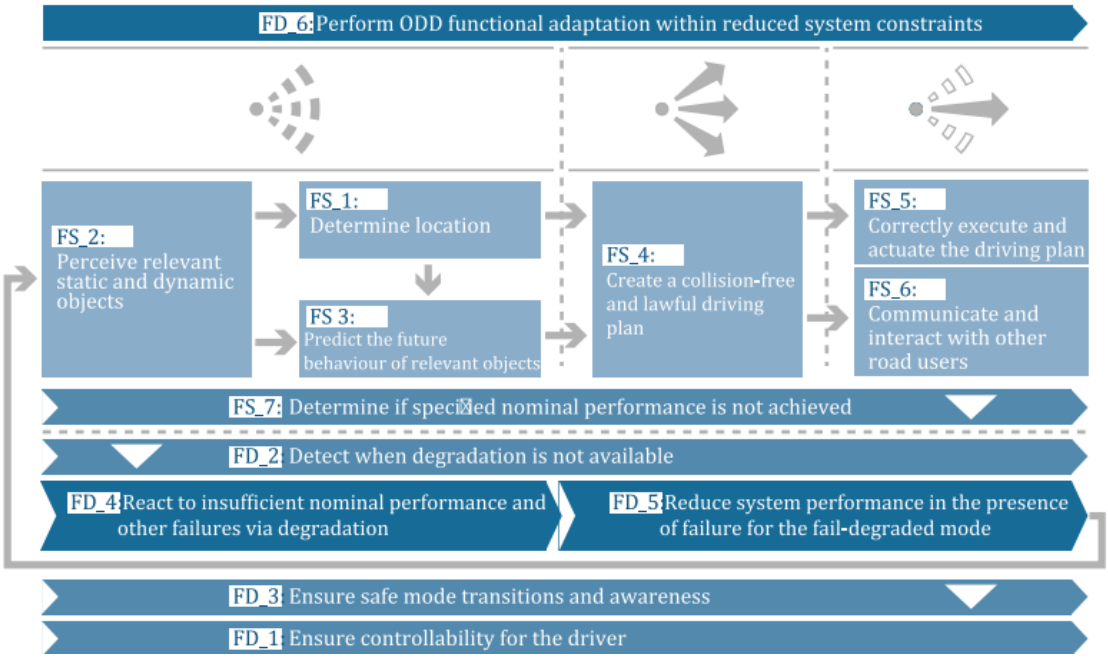

Figure 14. fail-safe & fail-degraded capabilities.

The classical architecture discussed above is outdated in the presence of the autonomous systems that are currently being developed, this architecture is only useful in a highly controlled environment. If the system's context is continuously changing, a more complex structure is needed. The environment is dynamic, and it is impossible to define all the contexts in which an autonomous vehicle will operate, therefore ISO 26262 is not applicable.

In order to ensure safety, the system should check and monitor if everything around it behaves as expected, look at the system's state, and manage the system to ensure safety. In any context (ODD) where it is defined, the system must work safely. As a result, the system is more complex than it already is, but it is also safer and more reliable.

To develop this kind of systems and explore new architectures, it is important to see the recommendations and standards defined by some organizations, such as [3].

The ISO technical report 4804 proposes a generic architecture for an autonomous vehicle. In this thesis we focus on a reduced scope of the proposed logical architecture. Our minimal autonomous driving system for automated parking is aligned with the Sense - Plan - Act paradigm and the capabilities defined in the standardisation. However, we consider only the set of components shown in Figure 15: perception, localization, path planning and vehicle motion.

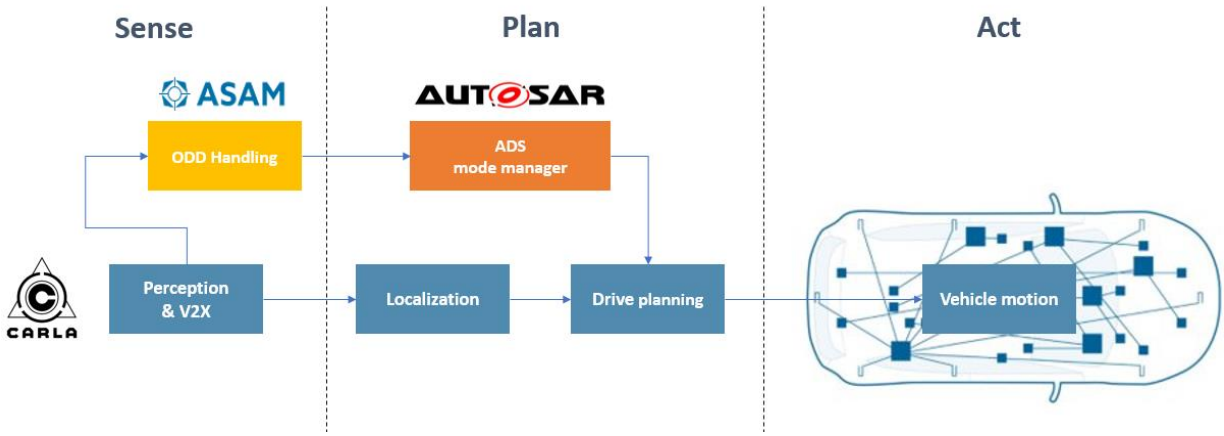

Figure 15. Safety-based architecture for the minimal autonomous driving systems.

To define the ODD's we have used the ASAM standard. Because of the safety requirements of the ADS mode manager, the architecture of the vehicle has been modelled using the standardized AUTOSAR architecture. The software architecture and the implementation of the ADS mode manager have been developed as part of another master thesis [4]. Although they are out of the scope of this thesis, the code has been jointly integrated and tested together with the implemented parking manoeuvrers in the CARLA simulator.

4.2.1.1 Components

Now let's talk about each of the components of the architecture with a focus on the role they play in our development.

The first component is the **Perception**, responsible for the identification of the environment around the vehicle sensors that produces objects (data). Perception is where the various inputs from the on-board sensors and the optional V2X information are captured to generate the actual world model.

The model includes a central lidar sensor, front and rear camera sensors, central GNSS sensor for localization and two radar sensors on each side at the front, centre and rear part of the vehicle. All these sensors are necessary to satisfy safe autonomous driving and will be used to drive the vehicle whenever necessary in our APS function. Additionally, some of them are essential for parking manoeuvres. The model also includes a V2I infrastructure for the connexion between the vehicle and the APS App described in section 4.1.

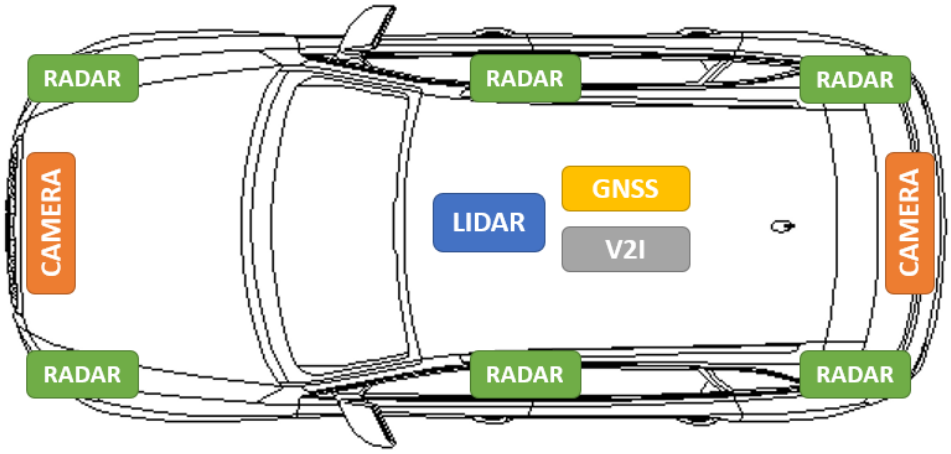

Figure 16. Sensor's location on the vehicle for the APS.

The **Localization** component focuses on the identification of the vehicle's environment. It is important that the automated driving vehicle localises itself appropriately, using information from environmental sensors, and identify objects in the environment. Furthermore, possible linking of additional a-priori information from outside the on-board perception performance (e.g., through mapping information, referencing detected events to a unique coordinate system), and considering egomotion information to predict whether the automated vehicle is about to exceed an ODD limit.

For the implementation in the simulator, it is very important to know the location of the vehicle. Also, knowing the environment for the localization is relevant and is acquired through ground truth, not from sensor data.

The **Path Planning** component sets out the manoeuvre to be performed to achieve the next driving step. This component must handle location information, follow traffic rules, consider egomotion and adapt the functionality according to the switches provided by the mode manager. The automated driving system obeys traffic rules for the drive planning element to produce a lawful driving plan unless crash avoidance manoeuvres can be prioritized over traffic rules to prevent collisions.

The model features two different parts in its motion, which are the autopilot and the parking manoeuvre. Autopilot mode is responsible for driving the vehicle to the parking spot. In this part, we use route planner of CARLA's autonomous pilot. For the parking manoeuvre mode, we have to define the type of parking, the different possible manoeuvres and the type of path planning.

As shown in Figure 17 to Figure 19, we distinguish three types of parking: vertical, parallel and angular. We define the parking angle as the angle between the parking spot and the side of the road. Therefore, parallel parking corresponds to 0°, perpendicular parking to 90° and angular parking can typically vary between 30°, 45°, 60° and 75°[24]. As manoeuvres we define two types of parking manoeuvres, two types of unparking

manoeuvres and one emergency manoeuvre to ensure a safe mode. All manoeuvres have been developed as part of this master thesis and are explained in section 4.3.3.3.

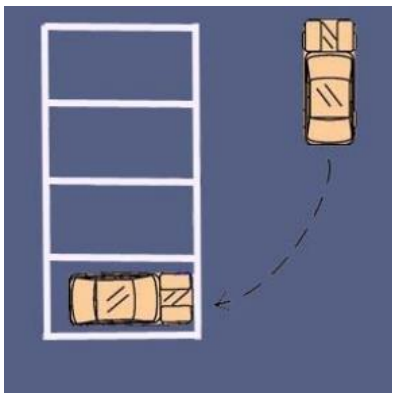

Figure 17. Perpendicular parking [25].

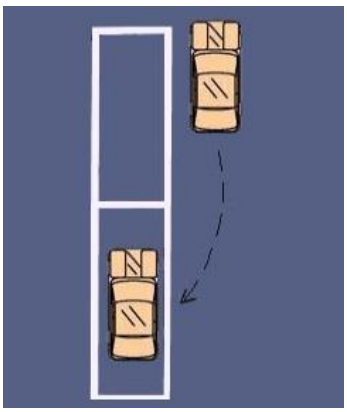

Figure 18. Parallel parking [25].

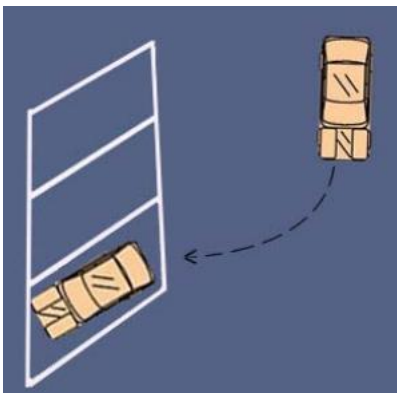

Figure 19. Angular parking [25].

The last component is **Vehicle Motion** and refers to its translation along and rotation about all three axes (e.g., longitudinal, lateral, and vertical). Rotations of a vehicle around these three axes correspond to angular momentum of the car body in roll, yaw, and pitch[26]. Shortly, the movement of the vehicle to achieve the path planed.

Using the CARLA simulator, the control of the actor (vehicle under test) is defined by the constants acceleration, braking and steering according to the function and mode of operation.

4.2.1.2 ADS Mode Manager

The **ADS Mode Manager** fulfils the task of safely switching between manual driving modes and the different automated driving modes. For the activation of an automated driving mode, this means getting all information to check if all prerequisites such as ODD are fulfilled (e.g. if the automated vehicle is on the right type of road, check weather conditions)[3]. This module is out of scope of this master thesis. The AS Mode Manager has been designed and implemented in the my co-worker's thesis[4].

4.2.1.3 ODD Handling

The **ODD Handling** component in Figure 15 is in charge of managing in which ODD the vehicle is driving. In this thesis, we only implement the operational domains that are supported by the CARLA simulator and apply them to the APS. For details, see section 4.2.2.

4.2.2 Operational Design Domain

The first step in establishing the capability of an automated driving system is the definition of its the operational design domain. The ODD represents the operating environment within which an ADS can perform the DDT safely, as we have seen in section 3.2.5. We follow

the PAS 1883:2020 standard which provides requirements for the minimum hierarchical taxonomy for specifying an ODD to enable the safe deployment of an ADS. The ODD comprises the static and dynamic attributes within which an ADS is designed to function safely. This PAS is applicable to Level 3 and Level 4 ADS[27].

Table 5, which follows the ASAM standardisation and PAS 1883, defines the ODDs of our APS focusing on the parking manoeuvres. The capability of the different attributes defines whether they are applicable for the simulator and whether they are relevant for our function.

Table 5. Supported ODDs for the APS.

		Attribute	Sub-attribute	Capability
Scenario	Zones	Geo-fenced areas		Yes
		Traffic management zones		No
		School zones		No
		Regions or states		No
		Interference zones		No
	Drivable area	Type	Indoor parking	Yes*
			Outdoor parking	Yes
			Shared space	No
			Motorways	No
			Urban roads	Yes
			Interurban roads	Yes
		Line type	Bus Lane	No
			Traffic lane	Yes
			Cycle lane	No
			Emergency lane	No
			Road lane	Yes
		Direction of travel - Only left-hand traffic		Yes
		Geometry - Longitudinal plane	Up-slope	Yes
			Down-slope	Yes
			Level plane	Yes
		Surface type	Loose (gravel, sand, etc.)	No
			Segmented	No
			Uniform (Asphalt)	Yes
	special structures	Pedestrians' crossings		Yes
		Bridges		No
		Rail crossings		No
		Tunnels		No
	fixed road structures	Buildings		Yes
		Streetlights		Yes
		Street furniture		No

		Vegetation		No
Environmental conditions	weather	Wind		Yes
		Rainfall		Yes
		Snowfall		No
		Sunny		Yes
	Illumination	Day		Yes
		Night / low ambient lighting		Yes
		Cloudiness	Clear	Yes
			Partly cloudy	Yes
			Overcast	Yes
		Artificial illumination		Yes
Dynamic elements	traffic	Parked vehicle		Yes
		Pedestrians		Yes
		Presence of special vehicles		Yes
		On road vehicles		Yes

*Yes: To integrate it in our simulation it is necessary to create a new map, which is out of scope of this thesis.

4.3 Detailed design

After the system description and the modelling of the architecture, now we can proceed with the detailed design of the APS function. From now on, the architectural model of the function will be adapted to the terminology of the CARLA simulator. We will cover in different sub-sections the CARLA environment, the functions, how they work in detail and how to change ODD's.

4.3.1 CARLA environment

CARLA is the simulation environment on which we integrate the APS for testing and validation. As already seen in appendix A, the simulation runs in a simulation world. In each simulation world, a single map is loaded with the possibility of placing different buildings, vehicles, pederasts, etc.

4.3.1.1 CARLA world and sensors

CARLA has 10 default maps created for the use of the simulator. Map 5 has been chosen for the APS simulation. The main reason is that it has space for parallel on-street parking and an outdoor car parking area for perpendicular parking. Figure 20 is the road layout of map 5, where the green area marks the on-street parking area and the orange area the outdoor parking. Figure 21 shows how the map environment looks like.

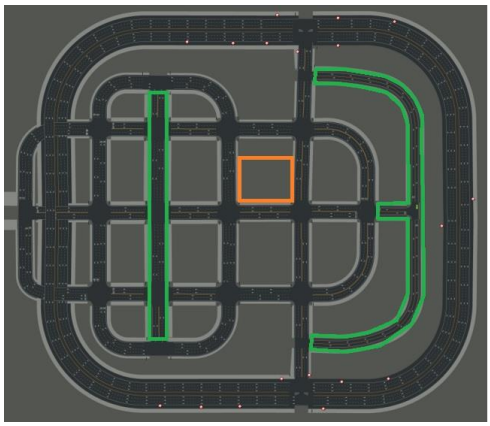

Figure 20. Map 5 layout in CARLA simulator.

Figure 21. Simulation environment in map 5.

For our implementation of the APS, we use specific CARLA classes to emulate the behaviour expected from the model. CARLA provides a number of sensors: a complete use of all sensors for our functions is out of scope, would lead to a huge workload and a new thesis. Next, we provide a short description of the Gnss sensor and the Obstacle sensor that have been used in this work.

The **Global Navigation Satellite System (Gnss) sensor** is attached to an actor to show its current position. This location is the geographical reference defined by the map definition, it shows longitude, latitude or altitude position of the actor. In this thesis, Gnss was used to emulate the GPS signal and get the localization on time of our vehicle. With this sensor we want to fulfil the APS use cases and simulate the use of parking spot and vehicle pick-up locations.

The **Obstacle sensor** is a sensor that detects obstacles during simulation. Detection depends on the view and range of the sensor. This sensor provides the distance to the detected obstacle. In this thesis, the Obstacle sensor was used to emulate the behaviour of a radar and to know how close our vehicle is with respect to other actors (vehicles, etc) to avoid collisions.

4.3.1.2 Custom CARLA classes

A Python API is used to access the CARLA simulator. We used open-source code provided by CARLA in [28], modified classes, and implemented our own classes to develop the APS function.

For instance, an individual file has been created for each of the sensor classes to be consistent with the software component types in [4]. This change includes the following sensors: camera, Gnss, IMU, LIDAR, radar and semantic LIDAR.

To better understand how our simulation's code works, we describe four custom classes and what they are used for.

Waypoint class

The first custom class is for waypoints. A Carla.Waypoint, no open-source code, is a 3D-directed point in the CARLA world that represents a directed point in the road. Each waypoint contains a Carla.Transform which states its location on the map and the orientation of the lane containing it. It also contains the variables road_id, section_id, lane_id and s corresponding to a road. These waypoints only exist and can be used along the routes defined on the map. This is a major drawback because it limits their use, and this is the reason why we must create our own class to allow us to use waypoints freely. Our waypoint class only contains the location and orientation.

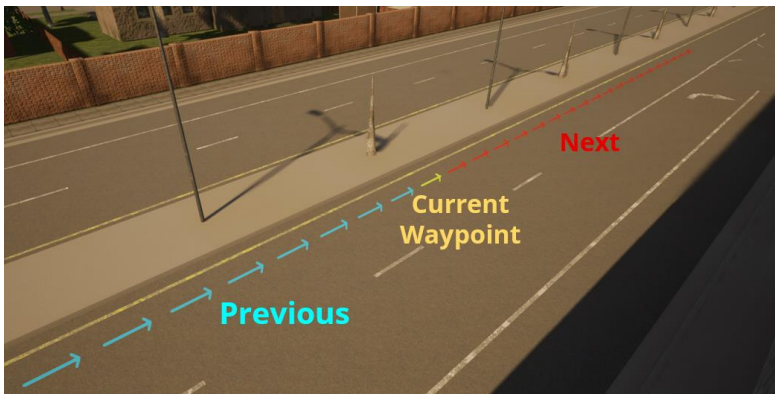

Figure 22. Waypoint in the road.

Controller class

The second custom class is the controller class, which is engaged in the vehicle's motion. Carla.VehicleControl class manages the basic movement of a vehicle using typical driving controls such as throttle, steer, brake, hand brake, reverse, manual gear shift, gear. This class will be the output of the controller. The vehicles follow a route defined by waypoints and the controller is in charge of controlling these variables. To manage the movement of the vehicle it is necessary to calculate the variables brake, accelerator and steering. Always drive with automatic gears. The control is divided in longitudinal and lateral types.

For longitudinal control we implemented a PID Controller. This controller will take the desired speed as the reference and outputs throttle and brake. A PID controller consists of three components:

- Kp: A pure gain Kp that scales the vehicle acceleration based on the speed error which ensures that the vehicle is accelerating in the correct direction with the magnitude proportional to the error.
- Ki: The integral term Ki sets up the output based on accumulated past errors, which ensures the steady steed errors are eliminated for ramp referencing.
- Kd: The derivative term Kd dampens the overshoot caused by the integration term.

The controller follows the next equation where the parameters are desired speed (v_d), vehicle speed (v), acceleration (u), proportional gain (k_p), integral gain (k_i), derivative gain (k_d).

$$u = k_p(v_d - v) + k_i \int_0^t (v_d - v)\, dt + k_d \frac{d(v_d - v)}{dt}$$

It is necessary to convert the acceleration from the PID controller into throttle and brake commands in order to complete the longitudinal control. Positive outputs correspond to throttle, while negative outputs correspond to break.

For lateral control is the same idea but controlling the steering of the vehicle instead of the speed. This steering is based in the rotation between the next reachable waypoint and the vehicle. The variables desired speed (v_d), vehicle speed (v), acceleration (u) from the previous formula are changed into desired steer (s_d), vehicle steer (s), steering (u). Positive outputs correspond to right steer, while negative outputs correspond to left steer.

Agent class

An agent in CARLA allows a vehicle to follow a random and endless route or to take the shortest route to a given destination. Agents obey traffic lights and react to other obstacles on the road. Parameters such as target speed, braking distance, following behaviour and others can be modified.

From the default agent, we have created an agent that contains the possibility to move all over the map (inheritance of the mother class behaviour agent), move inside the car park and manoeuvre for parking. As explained above, waypoints are only defined on the roads on the map, and we must create our own paths to drive into the car park and manoeuvring. More detail on this path planning will be given in section 4.3.3.3. In addition, a safe mode has been defined in the APS functionality and we must include it in the agent. An overview of the parking agent through pseudocode is shown below.

```
from my_controller import VehiclePIDController as MotionController
class CustomAgent(BehaviorAgent):
    # variable initialisation
    my_car_controller = MotionController()
    others
    # functions
    set_autopilot_destination(destination):
    set_parking_destination(destination):
    set_manoeuvre_destination(destination):
    run_step_autopilot():
        return run_step from BehaviourAgent inheritance class
    run_step_parking(target_speed = 10 (km/s)):
        if distance(my_vehicle & next_waypoint) > 0.5 m:
            return motion_stop()
        else:
            return controller.run_step(target_speed, next_waypoint)
    run_step_manoeuvre(target_speed = 5 (km/s)):
        if distance(my_vehicle & next_waypoint) > 0.2 m:
            return motion_stop()
        else:
            return controller.run_step(target_speed, next_waypoint)
    set_waypoints_queue(waypoints_queue):
```

```
    save waypoints_queue
done_autopilot():
    return boolean if it's done
done_parking():
    return boolean if it's done
done_manoeuvre():
    return boolean if it's done
emergency_lights():
    set emergency light state active
motion_stop():
    return control with max brake
safe_mode():
    emergency_lights()
    motion_stop()
```

SpotsMap class

This class has been created to manage all the parking spots that we have inside the simulation map, town 5, as seen in Figure 20. When instantiating the SpotsMap, a percentage of occupancy of the parking spaces is selected to occupy them randomly. Within this class there is another class called Spot that contains the information of the location and if the spot is occupied. The spot management functions are:

- get_all_spots
- get_occuped_spots
- get_free_spots
- get_street_free_spots
- get_parking_free_spots

4.3.2 APS functions

As described in section 4.1 and 4.2.1, we need to implement functions to park and unpark the vehicle for different types of parking spots. These functions must be able to reach their destination in autonomous driving and proceed with the parking (and unparking) manoeuvres. These functions are the following:

Table 6. Forwards parking manoeuvre description.

Function name	**PM_Forwards**	
Input	Carla.Localization(x,y,z)	Parking spot
	Float: angle	Angle of parking
Description	Parking manoeuvre for perpendicular and angular type of parking. If the angle is 90° it corresponds to perpendicular, otherwise to angular.	

Table 7. ForwardBackwards parking manoeuvre description.

Function name	**PM_ForwardBackwards**	
Input	Carla.Localization(x,y,z)	Parking spot
Description	Parking manoeuvre for angular parking type. This manoeuvre needs to drive in reverse.	

Table 8. Backwards unparking manoeuvre description.

Function name	**UM_Backwards**	
Input	Carla.Localization(x,y,z)	Pick-up point
	Float: angle	Angle of parking
Description	Perpendicular and angular type of unparking manoeuvre. This manoeuvre needs to drive backwards out of the parking spot.	

Table 9. BackwardForwards unparking manoeuvre description.

Function name	**UM_BackwardForwards**	
Input	Carla.Localization(x,y,z)	Pick-up point
Description	Parallel type of unparking manoeuvre. This manoeuvre leaves the parking spot in a forward direction.	

Table 10. Safe mode function description.

Function name	**Safe_Mode**
Description	Safe mode manoeuvre to avoid collisions or when the APS is aborted for any reason. This manoeuvre is activated automatically when necessary.
Behaviour	When this function is activated, the vehicle must stop and turn on the emergency lights (4 flashers, rear brake and dipped headlights). Out of simulation, but within the system under design, the user is informed through the APS App consequently.

Important details for these functions are highlighted. The angle is an important variable for the path planning in the manoeuvre and to ensure that the vehicle is parked in the right orientation. It is not relevant to know on which side of the road the parking space is located, with the location of the vehicle and the location of the space we already know where it is. If it is a two-way street, we will always arrive on the side of the road closest to the parking space, avoiding having to invade the oncoming lane. The simulation map does not allow driving inside an indoor car parking. Hence, we assume that the parking spot is always located on the ground floor.

4.3.3 Three phases of the APS functionalities

The implementation of the manoeuvre functions of the previous section follows a workflow consisting of three phases.

Our vehicle under test (VUT) is an agent that has the capabilities of navigating the map, driving into the car parking area and manoeuvring for parking. These three capabilities are the three phases that the APS follows to proceed with its functions: autopilot, car parking route and parking manoeuvre. For unparking, the autonomous vehicle follows a reverse workflow: unparking manoeuvre, car parking route and autopilot to the final destination.

An example of these three phases for perpendicular parking in a car parking area is shown in the next figure: blue path for the autopilot, orange path for driving the car in the parking and red path for the parking manoeuvre. All three phases are implemented in the parking agent's functions.

Figure 23. Example of the three motion planning phases for parking inside the car park.

In the case of the parallel parking manoeuvre (and unparking) we have the autopilot and manoeuvre phases, as we do not have to drive into the car park.

Figure 24. Example of two motion planning phases for street parking.

4.3.3.1 Autopilot phase

In this phase we take advantage of the autonomous pilot provided by CARLA's open-source code. Without going any further, we only have to indicate the location where we want to go and the VUT drives in autonomous vehicle to the place. It follows the traffic rules, the traffic lights and avoids collisions. We do not take care of the lidar sensors and cameras for this drive; hence we will not attach them to the VUT in the simulation.

4.3.3.2 Car parking route phase

To move the autonomous vehicle within the car parking area of our simulation map, we must create the route to reach our destination. To achieve this, we have created a graph-based method. The car parking coordinates are connected, the streets are given unique directions, and a direct graph is then generated. In the Figure 25 we see an aerial photo of the car parking area of map 5, where the orange lines mark the directions of each lane. These lanes are extracted from the implemented graph-based method.

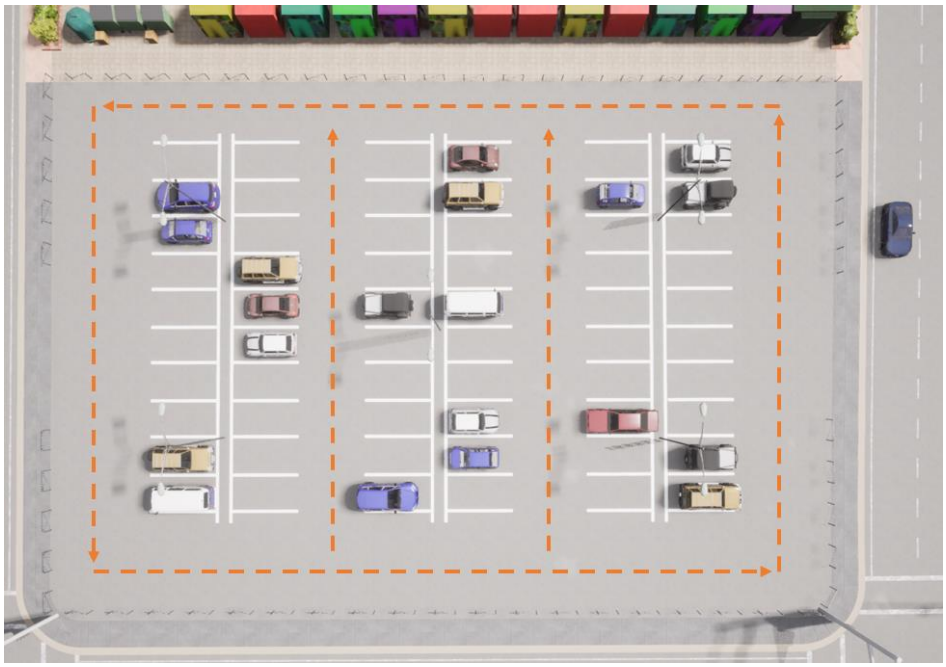

Figure 25. Car Park overview of map 5 with road directions.

To achieve this functionality, another custom class has been implemented in our project. In this class, we generate first a graph with the parking coordinates; a function to compute the shortest path in the graph has been implemented. This function, as a path planning method, returns the list of waypoints between the start point and the end point. These points should not be contained in the graph itself as we have also implemented an algorithm to translate the coordinate we pass through the function to the nearest coordinate on the graph. Figure 26 shows the routes of the car park (Figure 25). Each value within the blue circles corresponds to a coordinate on the map. Note that the arrows do not have the same direction in both figures, but this is due to having a different order of coordinates in CARLA's world.

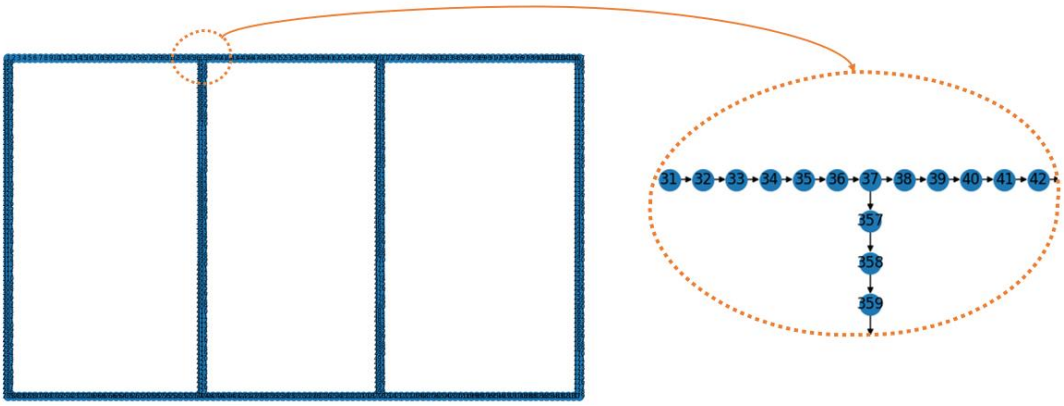

Figure 26. Directed graph of the car park.

To drive the car, we use the functions of our parking agent. First, we generate the trajectory inside the car park as explained above, and then iterate this trajectory (list of waypoints) to move the vehicle. If we detect any kind of obstacle through obstacle detector sensors (radar emulators), we stop the motion until we can continue.

4.3.3.3 Parking (and unparking) manoeuvre

The last phase is the parking manoeuvre. Here we apply the methodology explained the state of the art, in section 3.3, where we deal with a path planning of the fifth grade path because it is the one that best fits the desired manoeuvres.

For the reason explained in section 3.3, we follow the quintic polynomial planning to generate the trajectory that will allow us to park forwards and park backwards. Of course, for each type of parking, it is not the same type of trajectory/driving and must be adapted by adjusting some parameters. After that, although the optimal trajectory in a Frenet frame was the most promising approach, the integration into CARLA showed us that it would be demanding on the implementation side. For that reason, we decided to not use this approach for the path planning.

Thanks to the vehicle controller that follows a series of waypoints for driving (see section 4.3.1.2) we only need to focus on path planning.

Once we have the VUT ready to park, we calculate the reference path it should follow. The path is based on a quintic polynomial path planning. For the calculation of this path, we must set boundary conditions. In our implementation we have used a library called PythonRobotics[29], which contains the implementation of the quintic polynomial path planner. We provide the location of the VUT as the starting point, the location of the spot as the end point, the maximum acceleration and jerk, and the initial and final velocities and accelerations. These last variables we could think that they should be 0 in both cases, but for reasons of optimizing the path design we have to set the velocities to 1 m/s, because otherwise the orientation of the points is not respected.

Furthermore, for the trajectory adjustment, the maximum jerk is the most restrictive variable, hence we must study its behaviour. The Figure 27 is an example when the car is ready to manoeuvre towards the car park (orange cross destination).

Figure 27. Previous scene of parking manoeuvre.

Figure 28 is the representation of the trajectory of this example with different values of maximum jerk for the polynomial trajectory calculation. To choose the optimal value we should focus on having as realistic path as possible without crossing into other parking spaces. For large values the VUT may violate other parking spaces and collide with parked vehicles, and for too small values the VUT makes an atypical trajectory and deviates too much in the opposite direction of the parking space. Maximum jerk values between 0.01 m/s^3 and 0.1 m/s^3 are ideal, choice subject to testing in the simulation. A jerk value of 2 m/s^3 models the behaviour of an aggressive driver, while the comfortable jerk value of 0.9 m/s^3 refers to normal driving behaviour[30]. Our manoeuvre respects these values and proceeds to a smooth driving behaviour.

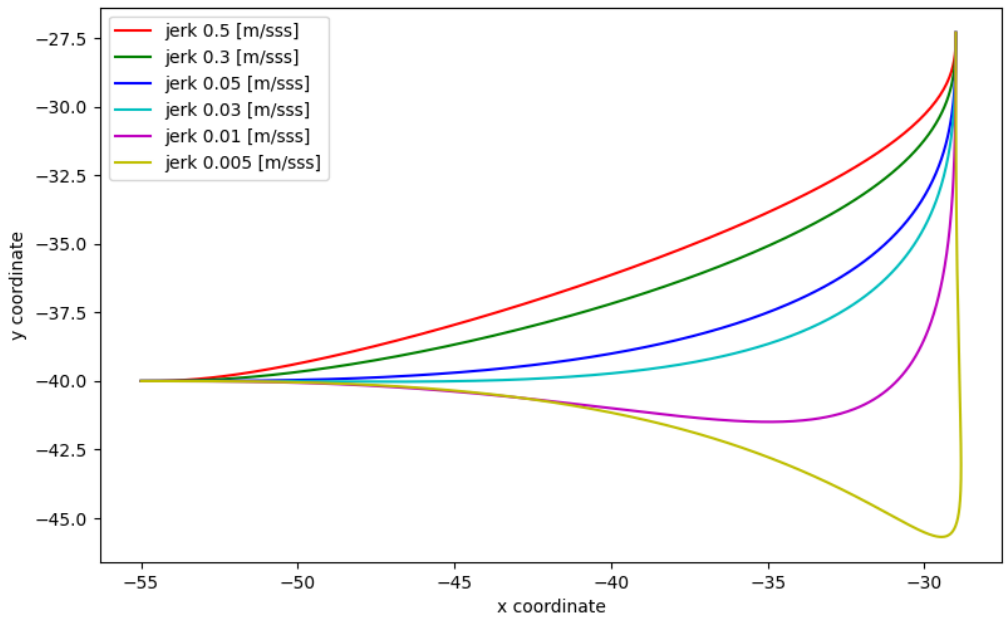

Figure 28. Quintic polynomial representation of the perpendicular parking path with different max jerk values.

Other interesting plots are the figures for the speed, yaw, acceleration and jerk of the example in Figure 27 with a maximum jerk of 0.01 m/s^3 . The time base is not scaled in a real simulation, rather scaled for experimentation and pre-study of trajectories.

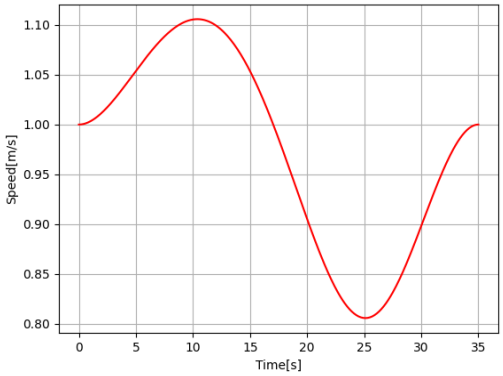

Figure 29. Speed vs time study.

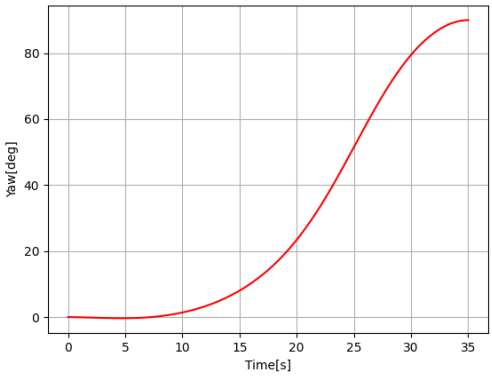

Figure 30. Yaw vs time study.

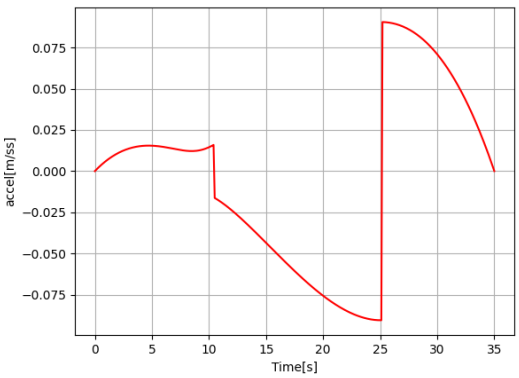

Figure 31. acceleration vs time study.

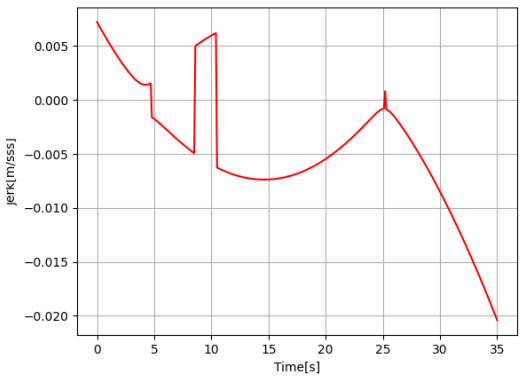

Figure 32. jerk vs time study.

In the appendix section 0 we can see the study for the parallel parking manoeuvre.

The flow that follows a manoeuvre is summarised in the following pseudocode. We first generate the reference path to follow (path planning) and then we iterate to move the vehicle. If we detect any kind of obstacle during parking through obstacle detector sensors (radar emulators), we stop the motion until we can proceed. We do not detail the distance because there are different casuistry depending on the position of the vehicle and the position of the sensor. This code is implemented inside the parking agent.

```
# set manoeuvre destination
target speed = 5 km/h
waypoints_queue = quintic_polynomials_planner(vehicle location, parking spot)
# do the manoeuvre
while (parking not done):
    if (distance with obstacle detected less than permitted):
        run step of the controller motion
    else:
        stop motion
stop motion and finish the APS functionality
```

4.3.4 Implementation of the functionality

The following pseudocode aims to show in more detail how the 5 different functions are implemented in the simulator. important, this is not the final flow of this implementation because it is not integrated with the AUTOSAR run-time environment. This implementation can be found in section 5. According to which APS functionality the simulation follows some steps or others. You can see that the functions PM_Forwards, PM_ForwardBackwards, UM_Backwards, UM_BackwardForwards are differentiated, but within each one you can see how the safe mode function is ready to act when necessary.

```
#Inicializations
simulation_world = world()
simulation_world.load_map(Town_5)
simulation_world.load_weather()
simulation_world.spawn(parked_vehicles)
VUT = ParkingAgent()
sensors = list of obstacle_detector_sensor
VUT.attach(sensors)
VUT.spawn(location)
state = (autopilot, parking, manoeuvre)
APS_function = (PM_Forwards, PM_ForwardBackwards, UM_Backwards,
UM_BackwardForwards)

#Main behaviour
switch (APS_function):
    case PM_Forwards:
        VUT.set_autopilot_destination(parking_location)
        state = autopilot
        while True:
            if any critical situation:
                VUT.safe_mode()
            else:
                switch (state):
                    case autopilot:
                        if (VUT.done_autopilot())
                            VUT.set_parking_destination(parking_lot)
                            state = parking
                        else
                            VUT.run_step_autopilot()
                    case parking:
                        if (VUT.done_parking())
                            VUT.set_manoeuvre_destination(parking_lot)
                            state = manoeuvre
                        else
                            VUT.run_step_parking()
                    case manoeuvre:
                        if (VUT.done_manoeuvre())
                            VUT.motion_stop()
                            break
                        else
                            VUT.run_step_manoeuvre()
    case PM_ForwardBackwards:
        VUT.set_autopilot_destination(parking_location)
        state = autopilot
        while True:
            if any critical situation:
                VUT.safe_mode()
            else:
                switch (state):
                    case autopilot:
                        if (VUT.done_autopilot())
                            VUT.set_manoeuvre_destination(parking_lot)
                            state = manoeuvre
```

```
                    else
                        VUT.run_step_autopilot()
                case manoeuvre:
                    if (VUT.done_manoeuvre())
                        VUT.motion_stop()
                        break
                    else
                        VUT.run_step_manoeuvre()
case UM_Backwards:
    VUT.set_manoeuvre_destination(parking_lane_location)
    state = manoeuvre
    while True:
        if any critical situation:
            VUT.safe_mode()
        else:
            switch (state):
                case manoeuvre:
                    if (VUT.done_manoeuvre())
                        VUT.set_parking_destination(parking_exit_location)
                        state = parking
                    else
                        VUT.run_step_manoeuvre()
                case parking:
                    if (VUT.done_parking())
                        VUT.set_autopilot_destination(pick_up_location)
                        state = autopilot
                    else
                        VUT.run_step_parking()
                case autopilot:
                    if (VUT.done_autopilot())
                        VUT.motion_stop()
                        break
                    else
                        VUT.run_step_autopilot()
case UM_BackwardForwards:
    VUT.set_manoeuvre_destination(street_lane_location)
    state = manoeuvre
    while True:
        if any critical situation:
            VUT.safe_mode()
        else:
            switch (state):
                case manoeuvre:
                    if (VUT.done_manoeuvre())
                        VUT.set_autopilot_destination(pick_up_location)
                        state = autopilot
                    else
                        VUT.run_step_manoeuvre()
                case autopilot:
                    if (VUT.done_autopilot())
                        VUT.motion_stop()
                        break
                    else
                        VUT.run_step_autopilot()
```

4.3.5 Consistency with the ODD's

In order to be consistent with the defined ODD's, and with the master thesis in [4], we have a function in the simulation to be able to change the simulation environment. Moreover, we are able to get this environment. This is important to finally integrate this implementation with the AUTOSAR generated RTE from [4]. These functions are used to communicate the simulation environment / context, see in section 5.

Table 11. CARLA functions to define specific ODDs.

Sub-attribute	Type	Object / Function	Variable
Urban roads	Getter	world.get_environment_objects	Road Lines
Up-slope	Getter	carla.IMUMeasurement	Compass
Down-slope	Getter	carla.IMUMeasurement	Compass
Level plane	Getter	carla.IMUMeasurement	Compass
Loose (gravel, earth)	Getter	world.get_environment_objects	Terrain
Uniform (Asphalt)	Getter	world.get_environment_objects	Terrain
Buildings	Getter	world.get_environment_objects	Building
Street lights	Getter	world.get_environment_objects	Traffic Light
Pedestrians crossing	Getter	world.get_environment_objects	Pedestrians
Water retentions on the spot	Getter & setter	carla.WeatherParameters	Precipitation deposits
Wind	Getter & setter	carla.WeatherParameters	Wind intensity
Rainfall	Getter & setter	carla.WeatherParameters	Precipitation
Fog	Getter & setter	carla.WeatherParameters	Fog density
Sunny	Getter & setter	carla.WeatherParameters	Sun altitude angle
Day	Getter & setter	carla.WeatherParameters	Sun altitude angle
Night / low ambient lighting	Getter & setter	carla.WeatherParameters	Sun altitude angle
Sky clear	Getter & setter	carla.WeatherParameters	Cloudiness
Partly cloudy	Getter & setter	carla.WeatherParameters	Cloudiness
Overcast	Getter & setter	carla.WeatherParameters	Cloudiness
Artificial illumination	Getter & setter	world.get_turned_on_lights / world.get_lightmanager()	Streetlight
Parked vehicle	Getter & setter	world.get_environment_objects / world.spawn()	Vehicles
Pedestrians	Getter & setter	world.get_environment_objects / world.spawn()	Pedestrians
On road vehicles	Getter & setter	world.get_environment_objects / world.spawn()	Vehicles

5 Results

This section aims to present the final result of this thesis, the software architecture of an autonomous vehicle with automated parking functionality, as mentioned in section 2.1. It integrates the results obtained from the Run-Time Environment of the modelled architecture and the implementation in the CARLA simulation environment. The RTE, which implements the communication between the application layer and the basic software services, must be generated to run the modelled architecture on hardware.

5.1 AUTOSAR Run-Time Environment

The system architecture and implementation of the AUTOSAR Runtime Environment (RTE) belongs to [4], so it is out of our scope. However, we provide a short overview on what it is about in order to understand the integration of both parts.

The key aspect of the architecture has been the design and implementation of the Autonomous Driving System (ADS) mode manager, which is responsible for ensuring safety in any context. The software architecture has been modelled in AUTOSAR using the mode management methodology described in the AUTOSAR standard. An RTE generator has been used to implement AUTOSAR Runtime Environment for the designed architecture, which provides the standardized APIs for setting/getting data and allocates the memory for the data layer.

The RTE does not only implement the communication between software components, but also the sequence how the runnables have to be executed. For instance, the *OsTask_100ms.c* file specifies the order of execution of the runnables, which will be cyclically called every 100-millisecond as specified in the configuration. The runnables must be ordered logically such that all necessary information is available before the next execution step.

The following code shows the sequence of every runnable specified in the architecture. First, the system performs the localization of the vehicle and its surroundings. Second, it collects data from the specified ODDs. Third, it updates the modes groups using all these received data. Fourth, it executes one of the five defined in section 4.3.2. Finally, it performs the motion specified by the selected function.

```
TASK(OsTask_100ms)
{
   {  GetSorrounderObjects();                }
   {  Get_ODD_Context();                     }
   {  SetCurrentModes();                     }
   if (Condition)  { PM_Forward();           }
   if (Condition)  { UM_Backward();          }
   if (Condition)  { UM_BackwardFordwards(); }
   if (Condition)  { PM_ForwardBackward();   }
   if (Condition)  { M_Safe();               }
   {  Set_VehicleMotionFunction();           }
   TerminateTask();
} /* OsTask_100ms */
```

The Rte.c file is the most interesting as it lists all the APIs that are used to write or read the internal variables defined in the architecture, consistent with section 4.3.5. These variables are the information needed to integrate it with the simulation environment. For example, the precipitation variable that is given by the CARLA simulator when reading the weather.

5.2 Integration environment

To ensure communication between the Run-Time Environment and the vehicle (simulation environment), a virtual communication protocol must be established. RTE is coded in C and simulation environment in Python, there must be compatibility between both development codes. For communication, a Transmission Control Protocol socket is used.

TCP socket is defined by the IP address of the machine and the ports it uses. In our case, the IP utilized is the Localhost, which is 127.0.0.1, because the RTE and the CARLA simulator operate on the same working station. The ports 4455 has been used, since this port is one of the used for the Transmission Control Protocol [31].

Servers and clients typically use TCP sockets. By listening on a well-known port (or IP address and port pairs), a TCP server accepts connections from TCP clients. To establish a connection with a TCP server, a TCP client must send a connection request to the server [32]. For the APS implementation, the architecture (RTE) is referred to as the server and the CARLA simulator as the client, see Figure 33. In addition, the combination of this thesis with [4] thesis can be clearly differentiated.

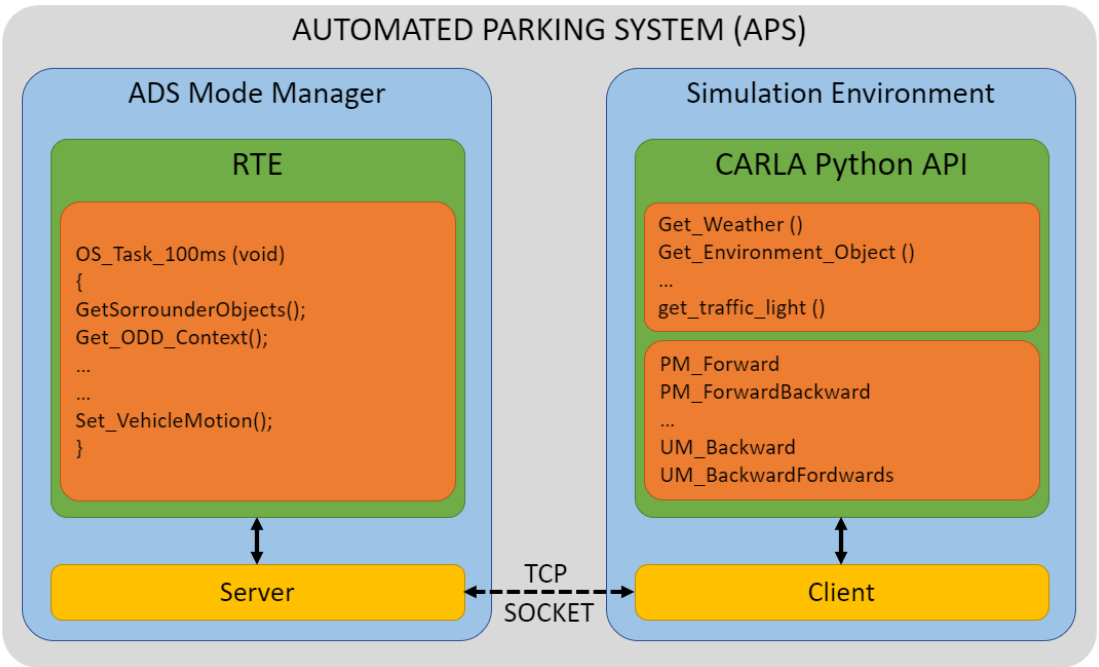

Figure 33. TCP socket communication protocol.

In order to execute the RTE tasks, an extra file has been created where the APS server is considered. As the TCP channel server, the file starts listening on the specified port to see if any client requests to establish a connection. Once the client sends the request to establish the connection, from the CARLA side, the server accepts it and both parties are connected.

The operation that follows the channel data flow is: server-client-server connection is established, the client sends the context information, the server gets the information, processes the information and provide the manoeuvre function to the simulation. All server work is the infinite loop execution of the Os_Task_100ms. The type of manoeuvre is encoded as shown in Table 12, related to section 4.3.2.

Table 12. APS functions codification for the TCP communication.

Value	Function manoeuvre
0	PM_Fordwars
1	PM_ForwardBackwards
2	UM_Backwards
3	UM_BackwardFordwards
4	Safe_Mode

5.3 Simulation environment adaptation

First, in order to integrate our work in the CARLA simulator in an easier and simpler way, we condense the implementation of the parking manoeuvres, seen in section 4.3.4, into a single function for each one. These functions will be collected by the agent, where they can then be easily called, and it looks like this:

```
class CustomAgent(BehaviorAgent):
    # variable initialisation
    ...
    # functions
    ...
    PM_Fordwards(destination)
    PM_FordwardBackwards(destination)
    UM_Backwards(destination)
    UM_BackwardFordwards(destination)
    Safe_Mode()
```

With this modification, the main file only has to choose which function to proceed according to the mode we are in.

Second, from the client side of TCP communication, the CARLA simulation, we must also emulate the mobile application. This refers to sending data from the external service, the mobile application. For this purpose, we have created a series of static constants in a settings file to simulate different situations, in consensus with the mode manager implementation on the server, see Table 13.

Table 13. Constants defined for the correct communication integration.

# PM OR UM	# APS ACTIVE OR NOT	# DESTINATION REACHED
PARKING = 0	APS_DESACTIVATION = 0	DESTINATION_NO_REACHED = 0
UNPARKING = 1	APS_ACTIVATION = 1	DESTINATION_REACHED = 1
# PARKING LOCATION MODE	**# APS FUNCTIONS**	
STREET_PARALLEL = 0	PM_FORDWARDS = 0	
STREET_ANGULAR = 1	PM_FORDWARDBACKWARDS = 1	
PARKING_OUTDOOR = 2	UM_BACKWARDS = 2	
PARKING_INDOOR = 3	UM_BACKWARDFORDWARDS = 3	
PARKING_ROAD = 4	SAFE_MODE = 4	

Finally, we create the client for the communication. The following pseudocode shows the client class we have created, where it includes a compact function for the required context information to send to the server.

```
import socket
class TCPClient:

    def __init__():
        port = 4455
        ip = "127.0.0.1"
        adrr = (ip, port)
        format = "utf-8"
        size = 1024
        client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        close_message = "200" #OK status http

    def connect():
        client.connect(adrr)

    def send(message):
        client.send(message.encode(format))

    def send_rte_message( wheather, destination_reached, compass, APS_activation,
parking_unparking, parking_location_mode):
        message = wheather.cloudiness + wheather.fog_density + wheather.precipitation
                + wheather.precipitation_deposits + wheather.sun_altitude_angle
                + wheather.wind_intensity + destination_reached + compass
                + APS_activation + parking_unparking + parking_location_mode
        send(message)

    def recive():
        return client.recv(size).decode(format)

    def close():
        send(close_message)
        client.close()
```

The TCPClient class contains the function send_rte_message(...) which includes all the necessary information to send. We transfer the weather information (cloudiness, fog density, precipitation, precipitation deposits, sun altitude angle and wind intensity), the inclination of the vehicle through the IMU information (compass variable) and the constants defined above. The constants parking / unparking and parking location mode refer to the necessary information of the APS App Mobil, the constants active / not active, destination reached / not reached are status constants, and the APS functions constants are used for decoding the message from the TCP channel to the manoeuvre in the simulation.

5.4 Developed prototype (MVP)

Once the architecture and simulation environment are fully integrated for the Automated Parking System, by including enough features to attract early clients and allow the product idea to be evaluated, the complete system is a Minimum Viable Product (MVP).

The overall concept of the MVP has been represented in a diagram to simplify the understanding of the data flow shown in Figure 34. The diagram represents the data flow of the TCP socket communication and the data flow between the blocks of the architecture. CARLA vehicle simulation flow not detailed, for further details see section 5.2.

The mode manager will only manage the different ODDs for weather changes. Appendix A.II contains several examples of weather condition contexts in the simulation environment. Information from sensors, environment objects and lights are not used in this MVP. This has been integrated to be consistent between architecture and CARLA simulator.

The components highlighted at the top of Figure 34 refer to the simulator (CARLA vehicle) and the architecture (Localization, ODD Handler, ADS Mode Manager, Drive Planning & Vehicle Motion). Since the ADS mode manager and the CARLA simulation have shared the same standards, the components defined in the safety-based architecture can be named identically, see again Figure 15. This completes the MVP with concurrency.

The CARLA vehicle component is the VUT in the simulation. Each component is described briefly; for further details get into [4]. The localisation component collects all sensor data. The ODD Handling component identifies in which context (ODD) the system is located by analysing the different information provided by the perception. The mode manager component manages the information received from the ODD handler and switches modes to satisfy safety. The drive planning component chooses the manoeuvre function according to the information from the localisation and the mode manager. The vehicle motion function component sends us the value according to the Table 12. Two examples of real simulation scenarios can be found in appendix C.

One technical problem we face is the tick rate of both servers. The TCP synchronous communication server follows a tick rate of 100 milliseconds which is set by the run-time environment task. The server where we simulate the vehicle, CARLA simulation, the tick rate is 5 milliseconds. If we violate the tick of the server we enter in undesired states, and in the case of the simulator we would lose FPS by visualizing the screen badly. As we know the integration works in an infinite loop, but the simulator does too (to be able to create each display frame in time). To solve the possible tick violation, the TCP communication will only be called every 20 times within the infinite loop in the simulation.

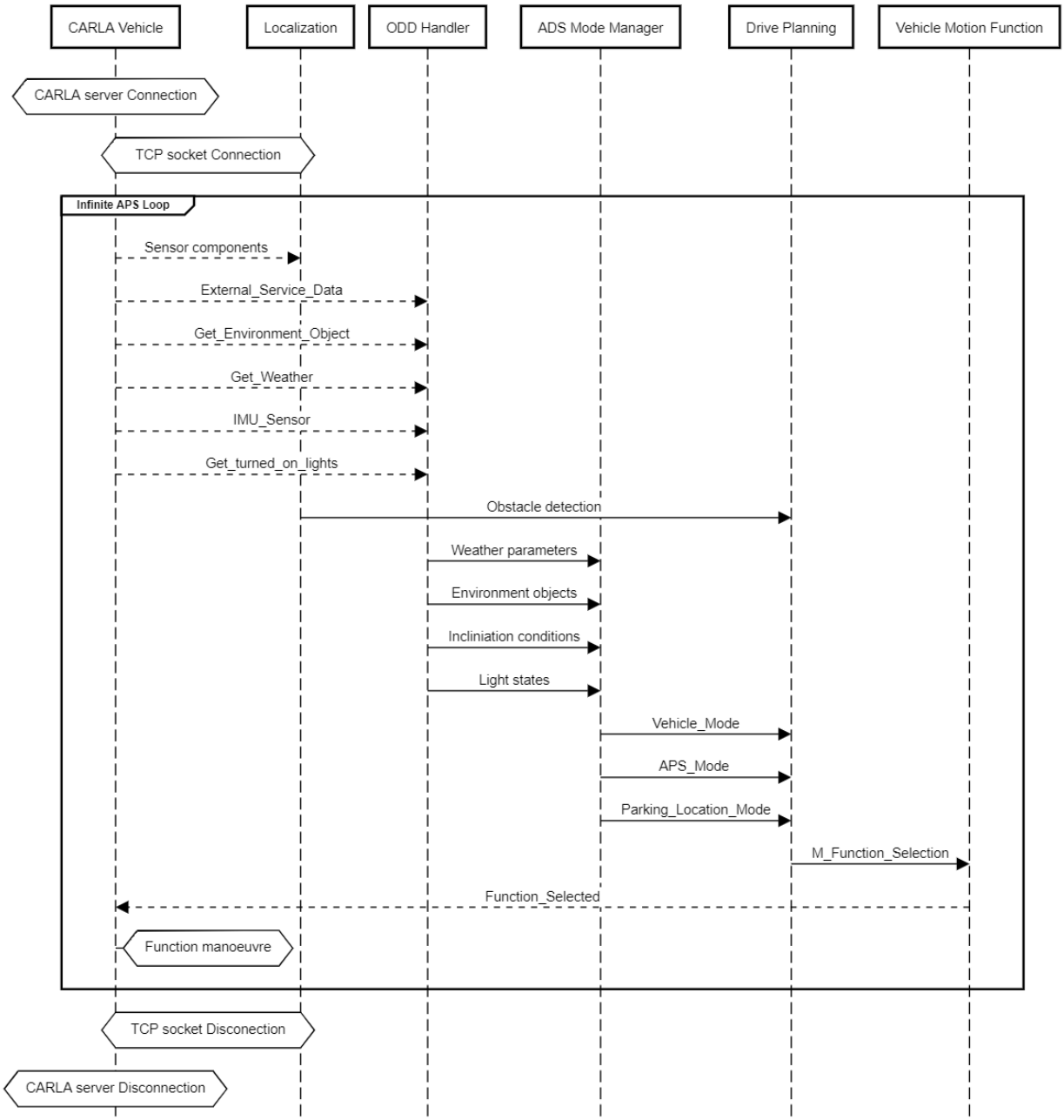

Figure 34. Generalist sequence diagram of the APS integration.

6 Budget

In this section are showed the cost related to the design of the Automated Parking System. This would be a representative case of a first employed year of a software architect engineer already formed.

6.1 Equipment

The costs of machinery and digital tools for in the project development appear in Table 14.

Table 14. Equipment costs.

Concept	Unit cost (€)	Quantity	Total (€)
Personal computer	800	0.75	600
CARLA simulator software	0	0.50	0
Working station (GPU)	3900	0.50	1950
Microsoft Office	30	0.30	9
		Total	2559

6.2 Human resources

Table 15 lists the number of working hours dedicated to the thesis and related tasks. The expense associated with the supervisory procedure is also included.

Table 15. Human resources costs.

Concept	Unit cost (€/h)	Quantity	Total (€)
Research	30	300	9000
Topic modelling	30	150	4500
implementation	30	300	9000
Writing	30	150	4500
Supervision	100	50	5000
		Total	32000

6.3 Total budget

Table 16 presents the total budget that results equipment and human resources.

Table 16. Total budget costs.

Concept	Total (€)
Research	2559
Topic modelling	32000
Total	34599

7 Environment and Social Impact

The automotive sector is one of the biggest in the world and a sector on which almost everyone is dependent in their daily lives. There are many means of transport, both public and private, but it is true that many users rely on their own vehicles to get around on a daily basis.

The APS function is conceived and designed to bring benefits to users, but also to bring sustainability to our planet. The main benefits are[33]:

- Safe driving. According to the US Department of Transportation, nearly 94% of fatal crashes are due to human error. That's why major automobile companies are pushing toward self-driving cars[34].
- Increase users' satisfaction. As an interesting fact, for example, with at least 293.6 million motor vehicles estimated to be on the road in the United States in 2021, parking will remain a challenge for a long time to come. With automation, drivers will not have to waste time looking for parking. In addition, the implementation of easy user interfaces, such as mobile apps, can make the user experience a convenient and memorable one that they will be willing to pay for.
- Reduce your environmental impact. With automated parking, cars don't waste time looking for a parking space and go straight to their spot, which reduces the amount of exhaust emissions compared to a driver's search.
- Save space and money. In an autonomous parking system, users no longer remain in the vehicle and can park in narrower spaces. This advantage allows to reduce the size of parking spaces and even to reuse this space.

8 Conclusions and future development

As the automotive industry shifts towards automated driving systems, the architecture of such systems shall adapt to the new challenges. This thesis presented a broad overview of the current standards that apply to the safety of autonomous systems. Different standardisations of software architectures and methods of operational design domains management have been evaluated in order to develop the autonomous parking system functionality. This functionality has been specified using a systems engineering approach.

With the goal of merging well-stablished and new technologies, an AUTOSAR modelled architecture and the implementation of an autonomous function in the CARLA simulator were combined. In contrast to the CARLA simulator, which is a simulation tool under development, AUTOSAR is a fully developed and mature industry standard.

The implemented use case, the automated parking system (APS), is based on the guidelines proposed in the document "Road vehicles - Safety and cybersecurity for automated driving systems - Design, verification and validation", ISO/TR 4804, 2020. The development in the simulator is exclusively focused on this functionality, from the setup of the map town to the path planning of the different proposed manoeuvres. However, several components of the simulator, such as the sensor instances, have been adapted to be consistent with the AUTOSAR architecture that has been developed with a generic overview. The developed system is scalable and generic. By adapting the system requirements, additional autonomous functions can be added.

The results presented in this thesis demonstrate that the implementation of a functionality based on the presented standards is affordable. Also, the simulation is able to successfully accomplish the validation of our autonomous parking system, both as a concept and as a simulation validation.

8.1 Future works

The main objective of developing an automated parking system in a simulated environment fulfilling industry and safety standards is achieved. However, there are potential improvements and extensions to this project. Some of possible future works is listed below:

- Expand the world environment with a customized map containing the three types of manoeuvres, indoor parking and outdoor parking.
- Deepen the integration of the optimal trajectory by Frenet frame in the CARLA simulator in order to reach a safer trajectory.
- Evaluate standardized robotics architectures (e.g., ROS2) against our simulation environment - AUTOSAR architecture to identify features that would improve the results of our work.
- Create a tool to generate the necessary code to adjust the vehicle in CARLA according to the specifications of the AUTOSAR architecture, e.g., sensors and their positions on the vehicle.

Bibliography

[1] Fraunhofer IKS, 'Principal site of Fraunhofer IKS'. https://www.iks.fraunhofer.de/ (accessed Apr. 01, 2022).

[2] W. Wang, Y. Song, J. Zhang, and H. Deng, 'Automatic parking of vehicles: a review of literatures', *2013*, vol. 15, p. 12.

[3] International Organization for Standardization (ISO), 'Road vehicles — Safety and cybersecurity for automated driving systems — Design, verification and validation', ISO/TR 4804, 2020.

[4] Y. El Kabdani, 'Software architecture design for safety in automated driving systems', Universitat Politècnica de Catalunya (ETSEIB), 2022.

[5] 'Systems engineering'. Accessed: May 07, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Systems_engineering

[6] 'Systems Engineering', *INCOSE*. https://www.incose.org/about-systems-engineering/system-and-se-definition/systems-engineering-definition (accessed May 07, 2022).

[7] T. Weilkiens, *Systems Engineering with SysML/UML*. Morgan Kaufmann, 2008.

[8] Wikipedia, 'V-Model'. Accessed: May 23, 2022. [Online]. Available: https://en.wikipedia.org/wiki/V-Model

[9] Society of Automotive Engineers [SAE], 'SAE J 3016-2018 - Taxonomy And Definitions For Terms Related To Driving Automation Systems For On-Road Motor Vehicles', Jan. 2014. Accessed: May 24, 2022. [Online]. Available: https://www.sae.org/standards/content/j3016_202104/

[10] A. Serban, E. Poll, and J. Visser, 'A Standard Driven Software Architecture for Fully Autonomous Vehicles', 2018 IEEE International Conference on Software Architecture Companion (ICSA-C), USA, 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8432195

[11] Wikipedia, 'ISO 26262 - Functional safety'. https://es.wikipedia.org/wiki/ISO_26262 (accessed May 27, 2022).

[12] International Organization for Standardization (ISO), 'ISO/AWI PAS 8800. Road Vehicles — Safety and artificial intelligence.', 2023. Accessed: Jun. 17, 2022. [Online]. Available: https://www.iso.org/standard/83303.html

[13] Wikipedia, 'Association for Standardisation of Automation and Measuring Systems'. Accessed: Apr. 25, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Association_for_Standardisation_of_Automation_and_Measuring_Systems#Portfolio_of_Standards

[14] ASAM e.V.all, 'ASAM OpenODD® - Concept project'. https://www.asam.net/standards/detail/openodd/ (accessed May 24, 2022).

[15] A. El Mahdawy and A. El Mougz, 'Path Planning for Autonomous Vehicles with Dynamic Lane Mapping and Obstacle Avoidance', doi: 10.5220/0010342704310438.

[16] P. Bautista/Camino, I. Cervantes, A. Barranco, M. Rodríguez, J. Prado, and F. J. Pérez, 'Local Path Planning for Autonomous Vehicles Based on the Natural Behavior

of the Biological Action-Perception Motion', Feb. 2022, doi: https://doi.org/10.3390/en15051769.

[17] S. Zhang, M. Simkani, and M. Hosseini, 'Automatic Vehicle Parallel Parking Design Using Fifth Degree Polynomial Path Planning', pp. 1–4, 2011, doi: 10.1109/VETECF.2011.6093275.

[18] A. Takahashi, T. Hongo, Y. Ninomiya, and G. Sugimoto, 'Local path planning and motion control for AGV in positioning', 1989.

[19] Wikipedia, 'Quintic function'. Accessed: Jun. 01, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Quintic_function

[20] M. Werling, J. Ziegler, S. Kammel, and S. Thrun, 'Optimal Trajectory Generation for Dynamic Street Scenarios in a Frenet Frame', pp. 1–7, Jun. 2010, doi: : 10.1109/ROBOT.2010.5509799.

[21] M. Werling, S. Kammel, J. Ziegler, and L. Gröll, 'Optimal trajectories for time-critical street scenarios using discretized terminal manifolds', pp. 347–359, 2011, doi: 10.1177/0278364911423042.

[22] K. Tong, Z. Ajanovic, and G. Stettinger, 'Overview of Tools Supporting Planning for Automated Driving', Mar. 2020.

[23] C. Pilz, G. Steinbauer, M. Schratter, and D. Watzenig, 'Development of a Scenario Simulation Platform to Support Autonomous Driving Verification', pp. 1–7, 2019.

[24] 'Parking Lot Angle Guide'. https://reliablepaving.com/blog/parking-lot-angle-guide/ (accessed Jun. 12, 2022).

[25] Y. Zhang, S. Tang, and Z. Zhang, 'A Novel Parking Control Algorithm for a Car-like Mobile Robot', Apr. 2012, doi: 10.1109.

[26] W. B. Ribbens, N. P. Mansour, G. Luecke, E. C. Jones, C. W. Battle, and E. Mansir, 'Understanding Automotive Electronics', 2003, doi: https://doi.org/10.1016/B978-0-7506-7599-4.X5000-7.

[27] 'Operational Design Domain (ODD) taxonomy for an automated driving system (ADS) – Specifi cation', ISBN 978 0 539 06735 4.

[28] 'carla github'. https://github.com/carla-simulator/carla

[29] 'PythonRobotics - quintic polynomial path planning'. https://atsushisakai.github.io/PythonRobotics/modules/path_planning/quintic_polynomials_planner/quintic_polynomials_planner.html (accessed May 20, 2022).

[30] M. Grobelna, P. Schleiß, Z. Joao-Vitor, and S. Burton, 'Dynamic Risk Management for Safely Automating Connected Driving Maneuvers', *Fraunhofer IKS*, pp. 1–8, doi: 10.1109/EDCC53658.2021.00009.

[31] X. Mertens, 'Port 4455 Attack Activity', [Online]. Available: https://isc.sans.edu/port/4455

[32] Dartmouth, 'TCP Socket Programming', Sep. 2018, Accessed: Jun. 05, 2022. [Online]. Available: https://www.cs.dartmouth.edu/~campbell/cs50/socketprogramming.html

[33] I. Todd, 'The Benefits of Automated Parking', Mar. 27, 2018. Accessed: Mar. 25, 2022. [Online]. Available: https://industrytoday.com/the-benefits-of-automated-parking/

[34] H. Kanchwala, 'Are Autonomous Cars Really Safer Than Human-Driven Cars?', Jan. 22, 2022. Accessed: Apr. 10, 2022. [Online]. Available: https://www.scienceabc.com/innovation/are-automated-cars-safer-than-human-driven-cars.html

[35] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, 'CARLA: An Open Urban Driving Simulato', p. 16, 2017.

[36] 'ASAM OpenDRIVE'. https://www.asam.net/standards/detail/opendrive/ (accessed Apr. 10, 2022).

[37] P. Pirri, C. Pahl, N. El Ioini, and H. R. Barzegar, 'Towards Cooperative Maneuvering Simulation Tools and Architecture', p. 6, Jan. 2021.

[38] 'Advanced driver-assistance system'. Accessed: Apr. 10, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Advanced_driver-assistance_system

Appendices

A. CARLA

This software is recognised by many leading companies in the industry and is highly valued and respected. Among these companies are Intel, Toyota, Samsung Europe, CVC, Valeo / KI Delta, Baselabs, various Fraunhofer institutes, and more.

Figure 35. CARLA sponsors[35]

On the other hand, the main reason for choosing this simulator is that it is open source software. Furthermore, based on the above comparison and looking at all open source simulators we can easily see that CARLA performs well in the different specifications. It is true that it is not suitable for V2X, but we will not need it in our thesis.

CARLA covers the research topics of this thesis, providing the full range of ODDs needed to develop the defined APS functions. It is very useful as it allows a fast, scalable and complete visualisation of the algorithms. This software is under active development, near five releases per year, good documentation and tutorials. Last but not least, it is widely distributed in academia and industry.

A.I What is CARLA?

CARLA is an open-source autonomous driving simulator. It has been developed from scratch to support the development, training and validation of autonomous driving systems. CARLA provides open digital assets (urban layouts, buildings, vehicles) created for this purpose that can be freely used. The simulation platform supports flexible specification of sensor sets, environmental conditions, full control of all static and dynamic actors, map generation and much more.

CARLA is based on Unreal Engine to run the simulation and uses the OpenDRIVE standard (currently 1.4) [36] to define roads and urban environments. The control of the simulation is done through an API managed in Python and C++ that is constantly growing as the project grows [35].

The Simulator

The CARLA simulator consists of a scalable client-server architecture (illustrated in Figure 36) that communicates via TCP. The client connects CARLA to the server, which with the help of the Unreal Engine 4 and the CARLA plugins runs the simulation. The simulator takes care of computing the physics and rendering the simulation scenes.

Once the client is connected to the server, it can retrieve data and send commands using scripts through the CARLA API. All functionalities are available for Python and C++. Python offers easy-to-use communication, which is what we will rely on for the simulation in this thesis.

One of the central concepts of CARLA is the world and the client. Once the client has connected to the server, it is necessary to load a simulation world in which the client can generate different actors (e.g., vehicles). From there, the client can constantly retrieve data and send commands with the help of the world object. The client contains the TM, which aims to recreate urban traffic to mimic real scenarios[37].

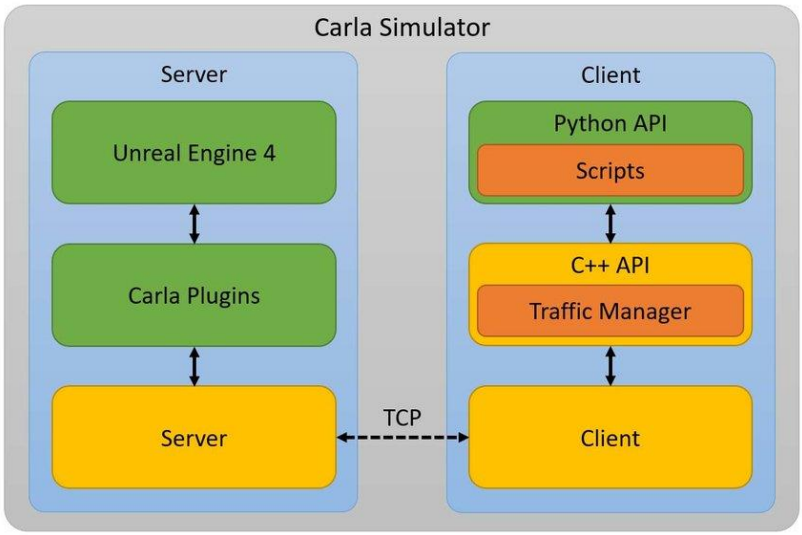

Figure 36. CARLA Simulator System Architecture Pipeline[35].

Understanding CARLA is much more than that, as many different features and elements coexist in it. Some of the most important ones are listed below:

- **Traffic manager (TM)**. An integrating system that takes control of vehicles. It acts as a driver provided by CARLA to recreate urban-like environments with realistic behaviour.
- **Sensors**. They are a specific type of actor attached to the vehicle where the data they receive can be retrieved and stored to ease the process. Vehicles rely on them to dispense information from their environment.
- **Recorder**. This feature is used to recreate a step-by-step simulation for each actor in the world. It allows access to any moment in the timeline anywhere in the world, providing a great tracking tool.

- **Open assets**. CARLA provides different maps for urban environments with weather control and a library with a wide set of actors to use. However, these elements can be customized, or new ones can be generated from scratch.
- **Scenario runner**. In order to ease the learning process for vehicles, CARLA provides a series of routes describing different situations to iterate on.

By default, CARLA runs in asynchronous mode, server runs as fast as it can. In synchronous mode the client, running your Python code, takes the reigns and tells the server when to update.

The World

The world is an object that represents the simulation. It acts as an abstract layer that contains the main methods to generate actors, change the weather, get the current state of the world, etc. For each simulation only one world exists. Every time the map is changed, the world is destroyed and a new one is created.

Figure 37. Cloudy road junction on map 5 of CARLA simulator.

Maps

A map includes both the 3D model of a city and its road definition. The road definition of a map is based on an OpenDRIVE file, a standardised and annotated road definition format. The way roads, lanes, junctions, etc. are defined determines the functionality of the Python API and the reasoning behind the decisions made.

There are eight cities in the CARLA ecosystem and each of them has two types of map, non-layered and layered. Layers refer to the objects grouped within a map (buildings, decals, stickers, foliage, ground, parked vehicles, particles, props, street lights, walls).

It is also possible to create customised maps or to use licensed maps of real cities.

Actors

Actors in CARLA are the elements that perform actions within the simulation, and they can affect other actors. Actors in CARLA includes vehicles, walkers, sensors, traffic signs, traffic lights and the spectator.

The life cycle of the actors consists of spawning, handling and be destroyed.

Sensors and data

Sensors are actors that retrieve data from their surroundings. They are crucial to create learning environment for driving agents.

The step-by-step process of a sensor within the simulator is: setting, spawning, listening and data.

There are three types of sensors:

1. Cameras: Take a shot of the world from their point of view. The types of cameras: depth, RGB, optical flow, semantic and instance segmentation and DVS.
2. Detectors: Retrieve data when the object they are attached to registers a specific event. The types of detectors: collision, lane invasion and obstacle.
3. Other: Different functionalities. Other types: GNSS, IMU, LIDAR, radar RSS and semantic LIDAR.

A.II Weather Operational Design Domains

This section shows different weather contexts where the Automated Parking System is simulated. Section 4.3.6.3 shows which modes can be selected according to the ODD selected.

Figure 38 illustrates a sunny day. All weather variables must have values of 0 to establish this ODD, except for the sun latitude, which must have a value of 75 degrees. This context is appropriate for all the defined APS modes specified, as it does not cause any problem for sensor components.

Figure 38. Sunny day.

Figure 39 depicts a night environment. In parking spaces where there is a low light level, the automated parking system cannot be performed since the system is not able to identify the surrounding objects. The APS's capacity to determine ambient light level is essential.

Figure 39. Low ambient lighting.

Figure 40 presents a sunny day with water deposits on the floor. In this condition, APS manoeuvres can only be performed when indoor park mode is activated, as this are prepared with scrapyards. Water deposits on the floor might cause line identification to be compromised.

Figure 40. Sunny day with water deposits.

Figure 41 illustrates a low light environment with water deposits on the ground. Like the previous one, it will only be possible to park indoors, as it is not affected by weather conditions.

Figure 41. Night ambient with water deposits.

Finally, Figure 42 shows a foggy day. For this ODD, the APS should only work for indoor parking lots, just like the previous two contexts.

Figure 42. Foggy day.

B. Quintic polynomial representation of the parallel parking path

In the same way as for the perpendicular parking manoeuvre, we must study the trajectory for the parallel manoeuvre. The Figure 43 is an example when the car is ready to manoeuvre towards the car park (red cross destination).

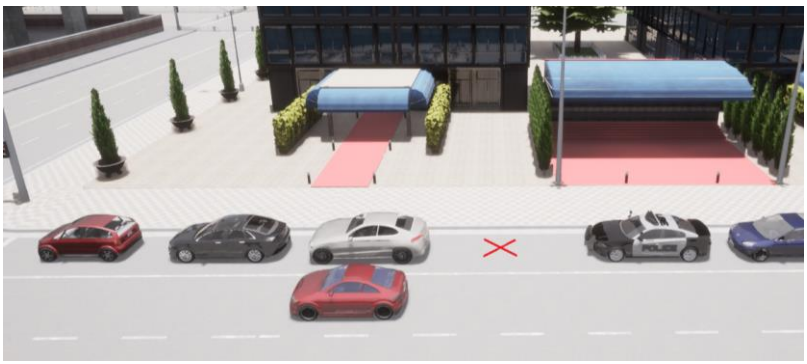

Figure 43. Previous scene of parallel parking manoeuvre.

Figure 44 is the representation of the trajectory of this example with different values of maximum jerk for the polynomial trajectory calculation. To choose the optimal value we should focus on having as realistic path as possible without crossing into other parking spaces. For smaller values the VUT may violate other parking spaces and collide with parked vehicles, and for larger values the VUT performs a more direct trajectory. Maximum jerk values between 0.5 m/s^3 and 0.8 m/s^3 are the ideal values, which can be proved in the simulation. To comply with the values referenced in [30], any value below 0.9 m/s^3 refers to normal driving behaviour.

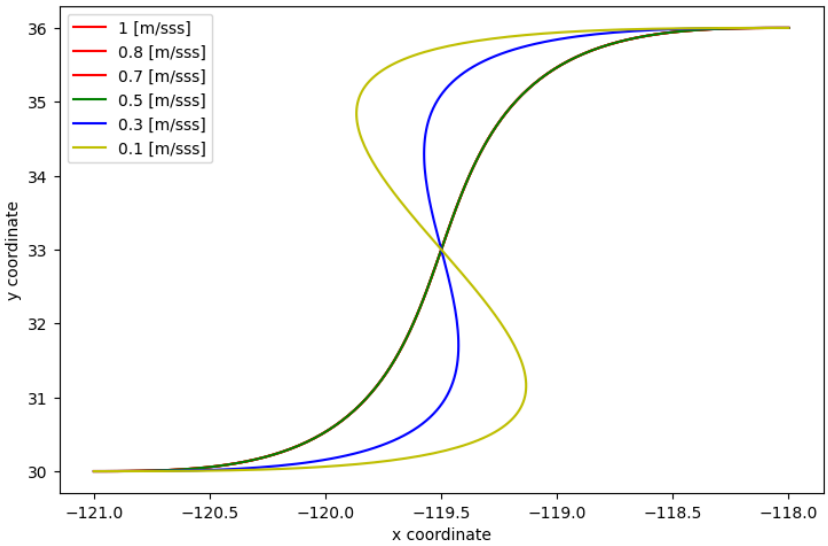

Figure 44. Quintic polynomial representation of the parallel parking path with different max jerk values.

Other interesting plots are the figures for the speed, yaw, acceleration and jerk of the example in Figure 43 with a maximum jerk of 0.8 m/s^3 . The time base is not scaled in a real simulation, rather scaled for experimentation and pre-study of trajectories.

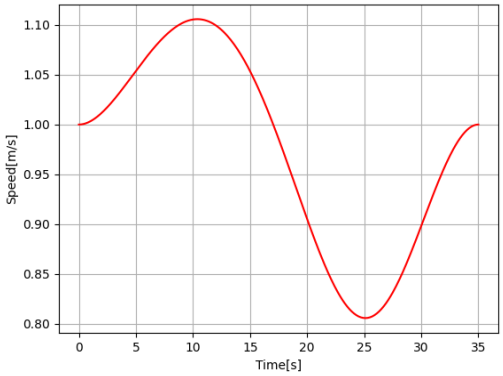

Figure 45. Speed vs time study.

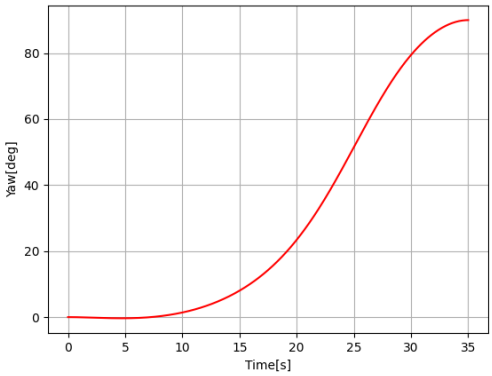

Figure 46. Yaw vs time study.

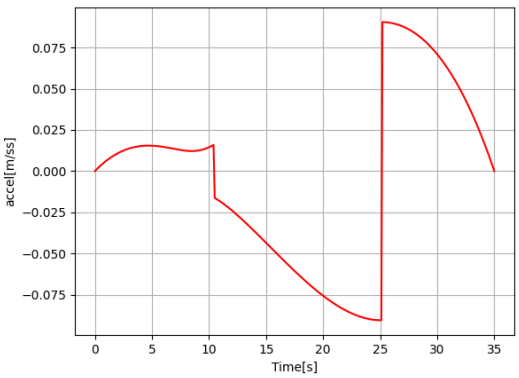

Figure 47. acceleration vs time study.

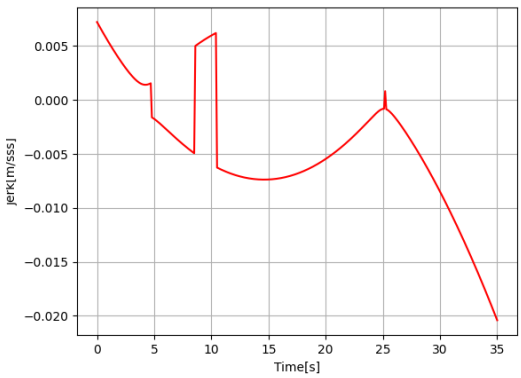

Figure 48. jerk vs time study.

C. Sequence diagrams for an example situation

In the following, some cases of simulation situations are presented to demonstrate the real data flow that the product should carry, in order to validate the MVP.

Figure 49 indicates the data transmission required to activate the PM_Forward function. The diagram shows that the weather conditions are suitable (sunny day), and no obstacle is detected. The manoeuvre can be performed without compromising safety.

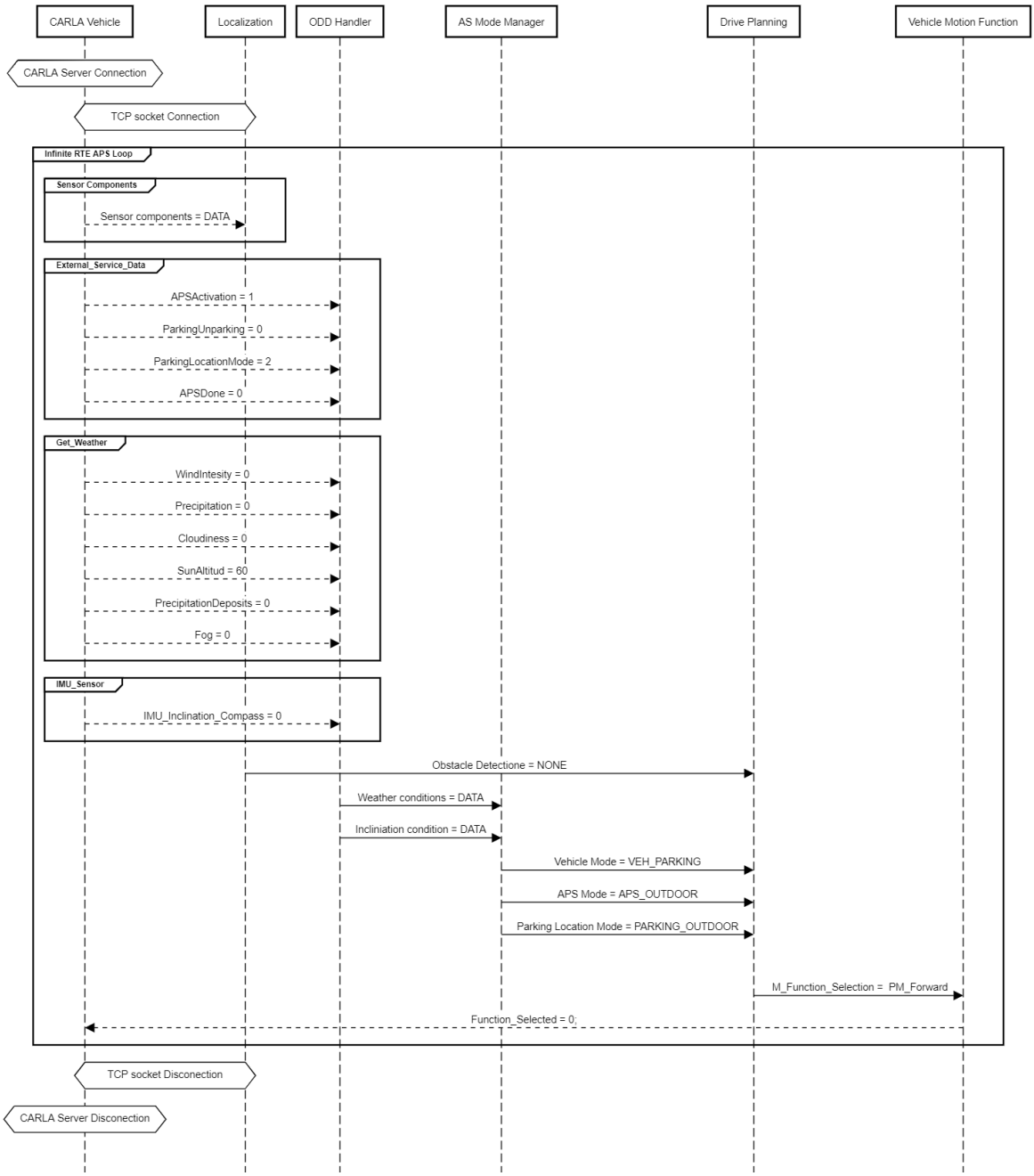

Figure 49. Case situation of the sequence diagram for the activation of the PM_Forward manoeuvre.

With the aim of ensuring safety, the MVP must react to a change in context and confirm that the specified modes are still operable under the new ODDs. In the above scenario, a storm with wind, rain and water on the ground breaks out during the chosen parking manoeuvre, see Figure 50. Due to the new conditions, the system switches to APS_SafeMode and executes the safe mode function.

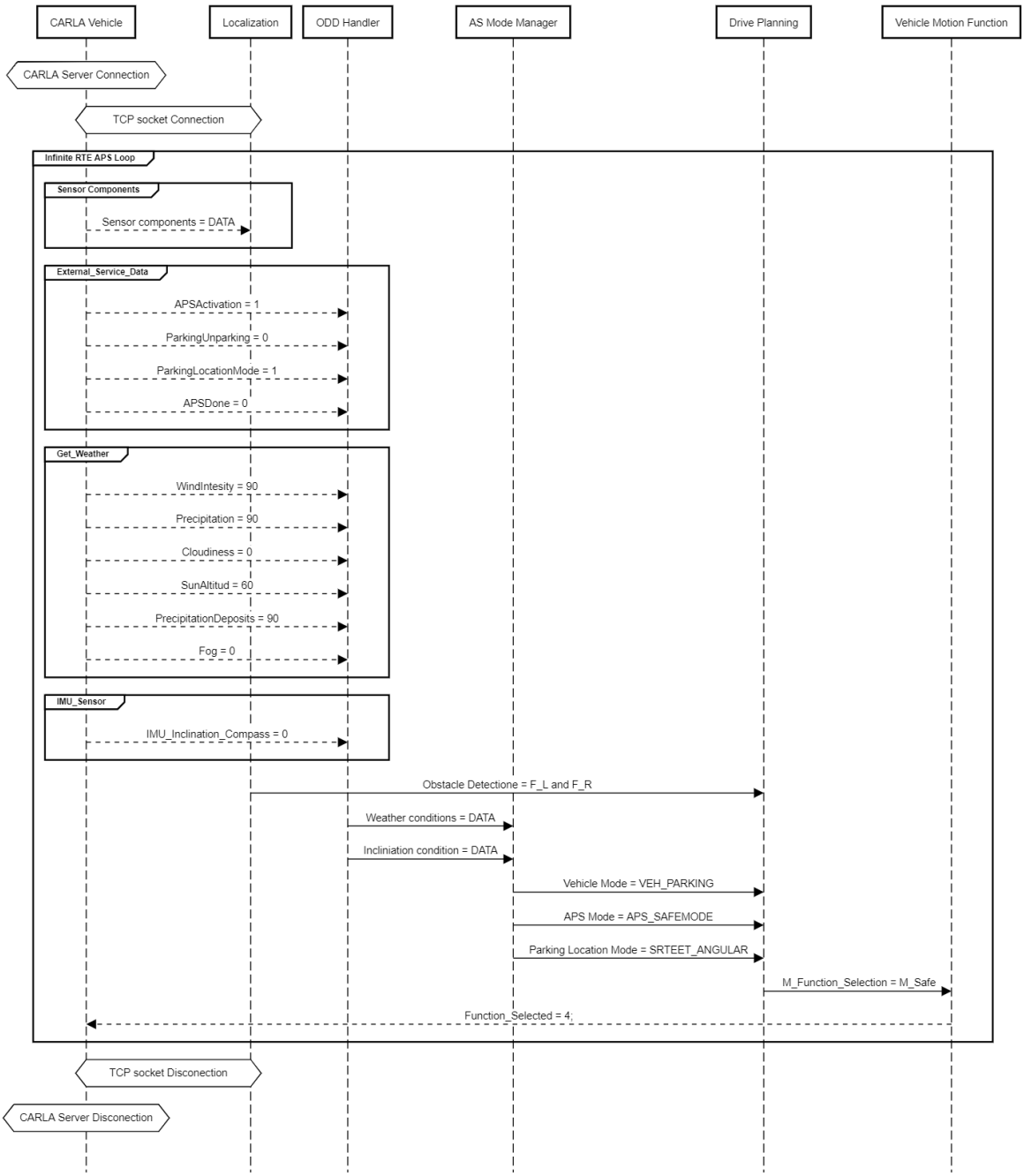

Figure 50. Case situation of the sequence diagram for the activation of safe mode.

D. Time planning

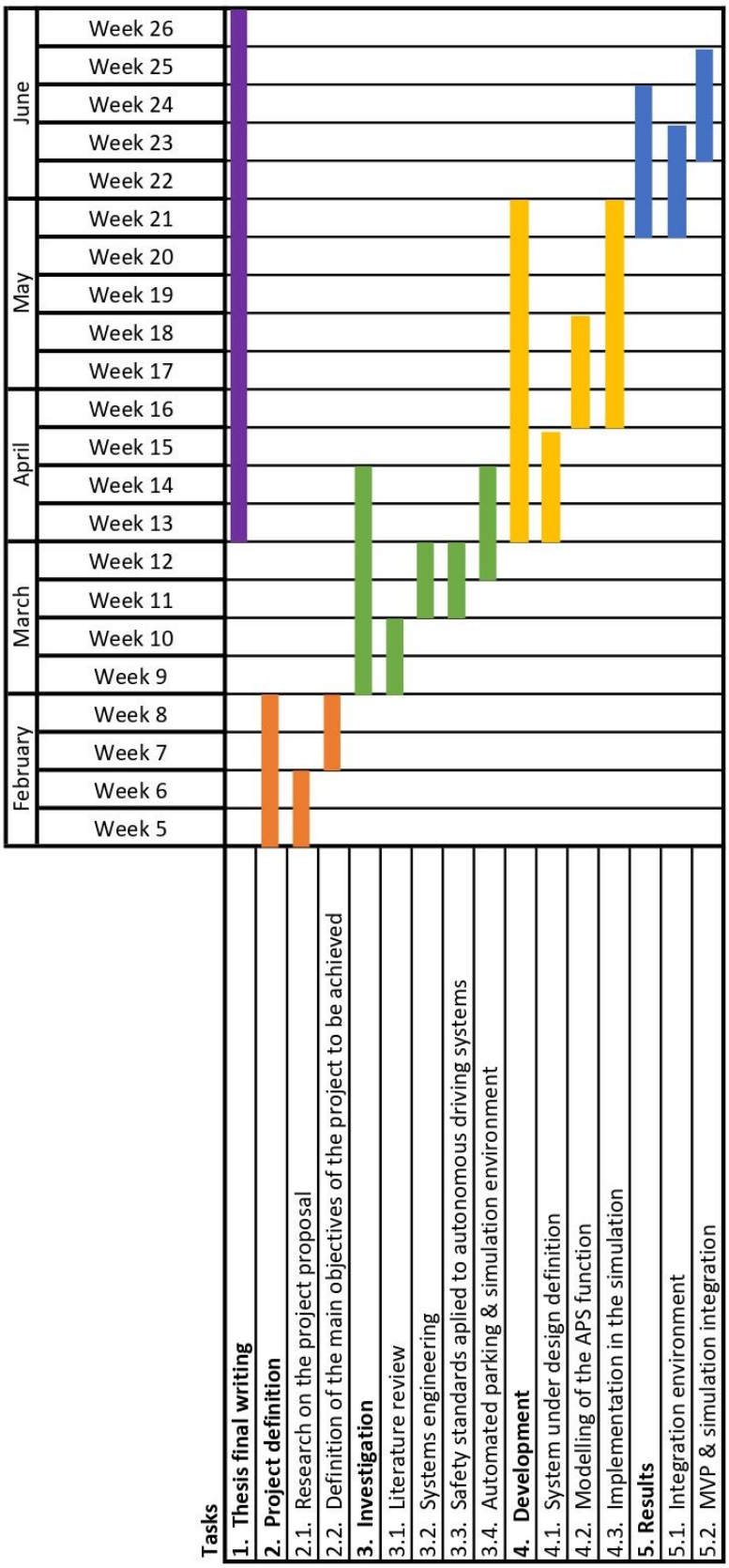

Figure 51. Gantt diagram of time planning.

Glossary

ADS: **A**utonomous **D**riving **S**ystem.

ADAS: **A**dvanced **D**river **A**ssistance **S**ystems are electronic systems that help the vehicle driver while driving or during parking[38].

AGV: Automated **G**uided **V**ehicle.

API: **A**pplication **P**rogramming **I**nterface.

AUTOSAR: Standardized **AUT**omotive **O**pen **S**ystem **A**rchitecture.

DDT: **D**ynamic **D**riving **T**ask. All of the real-time operational and tactical functions required to operate a vehicle in on-road traffic.

Egomotion: describes the actual state of the vehicle in terms of yaw rate, longitudinal acceleration, lateral acceleration and more.

Fail-safe capability: property of an automated driving system to achieve a minimal risk condition and to achieve a safe state in the event of a failure.

Fail-degraded capability: property of the item to operate with reduced functionality in the presence of a fault.

Gnss sensor: **G**lobal **N**avigation **S**atellite **S**ystem sensor.

LIDAR: **L**ight **I**maging **D**etection **a**nd **R**anging.

MVP: **M**inimum **V**iable **P**roduct.

ODD: **O**perational **D**esign **D**omain under which a given automated driving system is specifically designed to function, including, but not limited to, environmental, geographical, and time-of-day restrictions, and/or the requisite presence or absence of certain traffic or roadway characteristics.

RTE: **R**un-**T**ime **E**nvironment.

Runnable: Capable of being run.

TCP: **T**ransmission **C**ontrol **P**rotocol.

VUT: **V**ehicle **U**nder **T**est.