Master Thesis

Master’s Degree in Automotive Engineering

SOFTWARE ARCHITECTURAL DESIGN FOR SAFETYIN AUTOMATED DRIVING SYSTEMS

REPORT

Author:	Yassine El Kabdani Haddouzi
Director:	Dr.-Ing. Núria Mata Burgarolas
Supervisor:	Dr. Manuel Moreno Eguílaz
Call:	June 2022

ETSEIB

Escola Tècnica Superior
d’Enginyeria Industrial de Barcelona

UPC

ETSEIB

Abstract

Self-driving cars have generated a revolution in the automotive industry. Nonetheless, ensuring safety in the absence of a supervising driver and verifying safe vehicle behaviour in various contexts are two of the main challenges for autonomous driving systems that need to be addressed in the near future.

Due to their complexity, Autonomous Driving Systems (ADS) cannot be solved in a straightforward way without being properly structured. Therefore, it requires a well-defined architecture to guide its development. In addition to providing modularity and scalability, the proper architecture provides a maintainability system.

To help overcome some of the challenges, this master thesis develops an architecture for an ADS that adapts its behaviour to the context by switching between different operational modes, with the aim to standardize and ease the development process. The work was divided in four parts. First, the safety standards for the development of an autonomous functions have been analysed. Second, the system’s requirements were derived from a widely adopted automotive standard. Third, a logical architecture has been proposed and instantiated for an automated parking system. Finally, the architecture has been implemented in a simulation environment for its proper validation.

This work has shown that the architecture modelled in AUTOSAR and the generated Run-Time Environment is capable of adapting its behaviour to the context by executing the mode switch. In addition to meeting the safety requirements for a safe autonomous parking system. The interface created with the simulation environment allows future works to benefit from it for the development and testing of actual developed autonomous systems.

Key words: AUTOSAR, Mode Manager, Autonomous Driving Systems, Software Architecture.

Contents

GLOSSARY ______ 9

1. PREFACE ______ 11

1.1. Origin of problem 11
1.2. Motivation 11
1.3. Previous requirements 12

2. INTRODUCTION ______ 13

2.1. Background 13
2.2. Objectives 13
2.3. Scope 14

3. STATE OF THE ART ______ 15

3.1. Systems engineering 15
3.2. Software architecture 16
3.2.1. What is a software architecture? 17
3.2.2. Reference Software Architecture 18
3.3. AUTOSAR 18
3.3.1. Platform Software 19
3.3.2. AUTOSAR Classic Platform 20
3.3.3. Virtual Functional Bus 21
3.4. Safety standards for automated systems 21
3.4.1. V-Model 22
3.4.2. SAE Automated driving levels 23
3.4.3. ISO-26262 24
3.4.4. ISO/TR-4804 24
3.4.5. ASAM OpenODD Concept Paper 25
3.4.6. ISO/PAS-21448 27

4. DEVELOPMENT ______ 28

4.1. System Requirements for ADS Mode Manager 29
4.2. Logical architecture for safety 31
4.3. Detailed design 33
4.3.1. Perception 33
4.3.2. Localization 34
4.3.3. Drive Planning 34
4.3.4. AS Motion 35
4.3.5. ODD Handling 35
4.3.6. ADS Mode Manager 39

4.4. Vehicle modelling 46
4.4.1. Radar sensor 47
4.4.2. Camara sensor 48
4.4.3. Lidar sensor 49
4.4.4. Semantic lidar sensor 50
4.4.5. GNSS sensor 50
4.4.6. IMU sensor 51
4.5. Architecture model in AUTOSAR 51
4.5.1. Vehicle configuration 52
4.5.2. Perception 55
4.5.3. Localization 56
4.5.4. Drive Planning 57
4.5.5. AS MOTION 57
4.5.6. ODD Handling 58
4.5.7. ADS Mode Manager 58
5. RESULTS ________ 60
5.1. Run-Time Environment 60
5.2. Integration environment 62
5.3. Developed prototype (MVP) 64
6. CONCLUSIONS ________ 69
6.1. Future works 69
ACKNOWLEDGEMENT ________ 71
A. ENVIRONMENTAL AND SOCIAL IMPACT ________ 72
B. BUDGET ________ 73
B.I Equipment 73
B.II Human resources 73
B.III Total budget 73
C. TIME PLANNING ________ 74
D. CARLA SIMULATOR ________ 75
D.I What is CARLA 75
D.II Weather Operation Design Domain 78
E. AUTOMATED PARKING SYSTEM DEFINITION ________ 81
BIBLIOGRAPHY ________ 85

List of figures

Figure 1. Differences between AUTOSAR platforms [4]. 20
Figure 2. Main layers of the AUTOSAR Classic platform architecture [16]........ 20
Figure 3. Communication between the SWCs in a VFB [7]........ 21
Figure 4. V-model of the systems engineering process [19]. 22
Figure 5. Identification of SAE J3016 levels of driving automation [22] 23
Figure 6. ODD Taxonomy according to BSI PAS 1883 [29]........ 26
Figure 7. Example definition for an ODD taxonomy [30]. 26
Figure 8. Relation of the ADS mode manager with the APS........ 28
Figure 9. Classical software architecture "sense - plan – act" [26]........ 31
Figure 10. Example of safety-based architecture of the intended functionality [26]. 32
Figure 11. Safety-based architecture for minimal autonomous driving systems. 33
Figure 12. Defined parking types (parallel and angular types) [36] 35
Figure 13. Types of parking spot in CARLA: Car park and street parking........ 41
Figure 14. ADS Mode Manager state machine........ 45
Figure 15. Positioning of the sensor modules for the modelling of the vehicle hardware. 47
Figure 16. Preview of the virtual vehicle composition with the components defined........ 53
Figure 17. Components and mode interfaces for the architecture. 54
Figure 18. Architecture composition of the virtual vehicle with all the connections. 55
Figure 19. Provided ports for the sensor modules (Perception component). 56
Figure 20. Ports and runnable configuration for the Localization component........ 56
Figure 21. Ports and runnable configuration for the Drive Planning component. 57
Figure 22. Ports and runnable configuration for the AS Motion component. 58
Figure 23. Ports and runnable configuration for the ODD Handling component. 58
Figure 24. Ports and runnable configuration for the ADS Mode Manager........ 59
Figure 25. RTE files generated for the modelled architecture. 60
Figure 26. Generated Code-Frame components........ 62
Figure 27. TCP socket communication protocol. 63
Figure 28. Generalist sequence diagram of the APS integrated to the architecture. 65
Figure 29. Detailed sequence diagram for the activation of the PM_Forward manoeuvre. ... 66

Figure 30. Detailed sequence diagram for the activation of the safe mode. 67
Figure 31. Gantt diagram presenting the project evolution. 74
Figure 32. CARLA sponsors 75
Figure 33. CARLA Simulator System Architecture Pipeline. 76
Figure 34. Cloudy road junction on map 5 of CARLA simulator. 77
Figure 35. Sunny day. 78
Figure 36. low ambient lighting. 79
Figure 37. Sunny day with water deposits. 79
Figure 38. Night ambient with water deposits. 80
Figure 39. Foggy day. 80

List of tables

Table 1. System requirements applied to the ADS mode manager for APS. 29

Table 2. Constrains applied to the ADS mode manager for APS 30

Table 3. Supported ODD's for the APS definition and based on the PAS 1883:2020 [38]. ... 36

Table 4. Selected ODDs for the implementation of the APS in CARLA simulator. 37

Table 5. CARLA functions to define specific ODDs. 39

Table 6. Definition of the Vehicle Modes. 40

Table 7. Definition of the Automated Parking System Modes. 40

Table 8. Definition of the Parking Location Modes. 40

Table 9. Mode switch for defining the manoeuvre functions for the Drive Planning. 42

Table 10. Mode switch for defining the manoeuvre functions for the Drive Planning. 42

Table 11. ODD mapping with vehicle mode group. 43

Table 12. ODD mapping with the APS mode group. 44

Table 13. Definition of the Boolean variables. 45

Table 14. Selection of the APS Mode based on the Parking Location Mode. 46

Table 15. Instance variables for the radar module defined in the CARLA simulator [50]. 48

Table 16. Instance variables for the camera module defined in the CARLA simulator [53]. .. 48

Table 17. Instance variables for the LIDAR module of CARLA simulator [56], [57]. 49

Table 18. Instance variables for the Semantic LIDAR module of CARLA simulator [59]. 50

Table 19. Instance variables for the GNSS module defined in the CARLA simulator 51

Table 20. Instance variables for the IMU module defined in the CARLA simulator 51

Table 21. Description of the ETAS tools. 52

Table 22. Definition of the repeated components prototypes. 52

Table 23. Function manoeuvre codification for the TCP. 64

Table 24. Equipment costs. 73

Table 25. Human resources costs. 73

Table 26. Total budget of the thesis. 73

Table 27. Uses cases for the APS. 81

Table 28. Constrains for the APS. 82

Table 29. System requirements for the APS 83

Glossary

ADS: **A**utomated **D**riving **S**ystem. Hardware and software that drives the vehicle (applicable only to levels 3, 4, 5).

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

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

AP: AUTOSAR **A**daptative **P**latform.

API: Application **P**rogramming **I**nterface.

APS: Automated **P**arking **S**ystem.

AUTOSAR: Standardized **AUT**omotive **O**pen **S**ystem **Ar**chitecture.

ASAM: **S**tandardization of **A**utomation and **M**easuring **S**ystems.

CAV: Connected **A**utomated **V**ehicles.

CP: AUTOSAR **C**lassic **P**latform.

DDT: **D**ynamic **D**riving **T**ask. Driving the vehicle on a road. This includes two important sub-tasks: vehicle movement (acceleration, braking, steering), and OEDR (defined below). Example: Driving for at least several minutes on a highway while staying on the road, observing the actions of other road users, and manoeuvring to avoid crashes.

DMIPS: **D**hrystone **M**illion **I**nstructions **P**er **S**econd is a unit that evaluates performance based on the time it takes to execute a Dhrystone instruction program on a processor and process it.

ECU: Electronic **C**ontrol **U**nit.

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

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.

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

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

MRC: **M**inimal **R**isk **C**ondition. A Fallback puts the vehicle in an MRC to reduce the risk of a crash instead of attempting to continue a trip.

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

OEDR: **O**bject and **E**vent **D**etection and **R**esponse. Monitoring the driving environment including other road user actions to recognize the need for and execute a response.

OEM: Original **E**quipment **M**anufacturer.

ODD: **O**perational **D**esign **D**omain. Conditions the ADS is designed to handle.

RTE: Run-**T**ime **E**nvironment.

Runnable: Capable of being run.

SAS: Safer **A**utonomous **S**ystems.

SDL: **S**pecification and **D**escription **L**anguage.

SWC: Software **C**omponent.

VFB: Virtual **F**unctional **B**us.

1. Preface

1.1. Origin of problem

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].

The main challenge is that manufacturers of autonomous driving systems must guarantee that their products are safe for their customers and for the intended use. Many OEMs are making huge steps towards autonomous driving, but according to Safer Autonomous systems organization, 70% of people would still not take the risk of getting into a driverless vehicle, because safety is not guaranteed in every situation in which the vehicle operates [2].

The SAS consortium, consisting of several leading European research institutes and private industry companies, has been created and has aligned its efforts to solve significant challenges that could threaten the safety of autonomous systems. Fraunhofer IKS is one of the partners in the SAS consortium.

1.2. Motivation

Innovations in the automotive domain have been driven by electric/electronic systems and software in the last 20 years. Autonomous vehicles implement the capability to emulate human behaviour and replace the brain of the driver. This demands software systems, that we can trust to provide the autonomous functionality in a reliable and safe way.

The increasing, demanding requirements on safety, environmental protection and comfort have also increased the number of electronic systems and changed the whole structure of these systems. This makes that more than 90% of all innovations are in electronics and software systems, and up to 40% of a vehicle’s development costs are derived from the electronics in the vehicle, and around 50 to 70% of the electronics costs are related exclusively to software [3]. However, the required development processes, methodology to ensure safety and technologies are still in their early stage.

This master thesis is a great opportunity to learn about the developments and standards related to autonomous systems. The current and future challenges have wakened my interest in discovering how to design an autonomous system from scratch, which is a totally innovative aspect of the automotive industry.

My main motivation for this master's thesis has been to extend the knowledge acquired in the automotive master's degree. More specifically, build the expertise required for designing, developing and integrating autonomous vehicle functions. This includes learning how to structure and solve a complex cyber physical system and managing its states, with the aim of improving the overall quality, reliability and resource efficiency.

1.3. Previous requirements

For the development of the project, no specific knowledge of standards or tools is required, but it is recommended to have some notions of systems engineering in order to have the ability to develop and design complex systems, such as autonomous systems.

2. Introduction

2.1. Background

The project has been developed at Fraunhofer Institute for Cognitive Systems, specifically in the department of Cognitive Software Systems Engineering, where one of the research focus areas is the engineering for safe intelligence systems and the modelling of resilience architectures for autonomous systems.

Autonomous driving systems are a core field of robotics research, and many universities, research institutes, and companies are now working on autonomous driving technology. Interest in automated driving technology has increased exponentially over the past few years driven by the goal of reducing road fatalities, improving traffic conditions and the introduction of new mobility concepts. When doing so, the safety of automated driving vehicles is one of the most important factors [4].

The continuous evolution of automotive technology aims to deliver even greater safety benefits than earlier technologies. One day, automated driving systems, which some refer to as automated vehicles, may be able to handle the whole task of driving when we do not want to or cannot do it ourselves [5].

The purpose of this work is to design and engineer resilience software systems which are open systems. This means that they perceive the environment, can identify relevant objects and predict their possible trajectories. However, the main goal is to adapt the functionality of the system to the actual context that is facing, ensuring in this way the safety of the system. This concept is called safe intelligence.

With the interaction of automated vehicles with the environment and with other systems, the design of the software is becoming increasingly complex. To facilitate the evolution of today's vehicles to automated systems, it is necessary to extend existing standards, define new standards and apply standardized architectures to manage the increasing complexity.

2.2. Objectives

The goal of this thesis is to design and model a software architecture in AUTOSAR for a minimal autonomous driving system that adapts its behaviour to the context by switching between operational modes. The communication between the software components, including the mode management, shall be implemented by the AUTOSAR Run-Time Environment (RTE). The software architecture will be instantiated in a case-study for an

Automated Parking System.

2.3. Scope

This master thesis focuses on electronic systems, specifically on automotive embedded systems and their control in terms of standards for autonomous driving systems (ADS).

It is in the scope of this project the modelling of the software architecture using the Automotive Open System Architecture “AUTOSAR”, with special emphasis on the use of the standardized mode-management mechanisms. The architecture will be instantiated to an Automated Parking System in order to verify the correct functionalities of the system.

The concept shall be validated in a virtual environment. The generated AUTOSAR Run-Time Environment (RTE) will be integrated with the environment and the autonomous vehicle implemented by the CARLA simulator. It is not in the scope of this master thesis the implementation of the parking manoeuvrer, which has been specified and developed in [6].

This document is structured as follows. Chapter 3 starts with the state-of-the-art of the safety standards for the automated driving systems. Chapter 4 focuses on the description of the proposed autonomous system and on its development. The implementation results and verifications through simulations are presented in chapter 5. Finally, chapter 6 summarizes the conclusions and future work.

3. State of the art

This chapter provides an overview of the most important techniques, methodologies and standards used for the development of this master thesis. The goal is to help the reader with background information to understand the context of the project and the results.

This state of the art covers the following aspects: systems engineering (Section 3.1), software architecture (Section 3.2), AUTOSAR [7] as a standardized software architecture (Section 3.3) and safety standards for automated systems (Section 3.4).

Systems engineering is crucial for the development of this project. The development of such a complex system as the autonomous vehicle, requires proceeding using a structured approach, dividing the system into small subsystems to successfully manage the complexity.

For subsystems to be implemented in software, a clean and well-defined architecture must be designed. It is important to know about the applicability of the different architectures, how they are defined, how they are documented, how they are referenced, etc. This provides the necessary skills to decide which type of architecture fits best to the system under development.

Finally, it is important to be aware of all the standards that are relevant for the development of the project so that the modelling is done according to these standards. The focus will be on AUTOSAR and on the relevant safety standards for autonomous systems since the main goal of this work is to ensure safety in the architecture.

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 since it is not committed to a specific field. For example, it can be applied for studying the human digestive system or computer system, as an example of two different fields without anything in common [8].

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 [9].

During system design, all the information that is needed is explicitly described, e.g., what functionality the system provides, how the system will interact with other systems, how many different components are needed to implement the system, how the interaction among components is, what part of the functionality each component implements, etc. This information allows the hierarchy of the system to be drawn, not only at the top level, but also in detailing the requirements to meet the overall 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 must 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. Its main benefits 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.

The process of systems engineering is iterative. To begin, the project must be architecturally defined. The design becomes more precise at the component level as the requirements, communication, and implementation details are polished further. Overall, systems engineering provides a roadmap for successful engineering project design and implementation.

This master thesis makes use of systems engineering for the development and decomposition into small subsystems.

3.2. Software architecture

Software architecture is key for the development of future-oriented software-based systems. In this section we define what an architecture is, so that at the time of design and development, it can be described in the optimal and correct way. For complex systems, such as autonomous driving systems, the most efficient approach is to model the architecture first, then determine the detailed requirements for the components to be implemented.

The software architecture of a system represents the structure of the system itself and it also provides a logical explanation of its high-level behaviour [10]. In this case, a system consists of a set of components that fulfil a specific function and it may also be a set of functions. In other words, software architecture is a necessary tool for developing effective software since it provides a stable framework on which the software can be developed.

The quality, performance, maintainability, and overall success of the described system are all directly influenced by the architecture decisions to be made. Different patterns and principles of high-level architecture have been developed and are used in modern and complex systems, which are called architectural styles. It is clearly stated in [10] that the architecture that composes a system is not frequently limited to a single architectural style, but it is usually the combination of different styles that form complex systems.

The software architecture directly exposes the system external structure, while covering up the system implementation details. Likewise, it focuses on how the system elements and components interact with one another.

There is no simple and structured way to determine whether an architecture is correct or wrong. The architecture itself must have a finality, it has to fulfil a purpose and to meet the defined objectives.

Some golden guidelines for the engineering process for defining architectures are as follows. First, the process must be managed by a single person or a small group of architects. Second, the architect must respect to the architectural quality features. Third, everything should be documented and, moreover, from several perspectives. Finally, review the architecture again, from the initial concept through the most recent extensions [11].

3.2.1. What is a software architecture?

The software architecture consists of a scheme of boxes and lines of the system that intends to solve the problems for which it has been thought, the boxes are the elements of the system while the lines are the relationships and information exchange between these elements.

A more formal definition of software architecture is: “Software architecture of a system is the set of structures needed to reason about the system, which comprises software elements, relations among them, and properties of both” [12]. In other words, a software architecture is a system abstraction in which the architecture defines the elements and how they connect to one another, but not the internal details.

For a variety of reasons, architecture is an essential component of engineering. The first is that it provides as a gateway for communication between stakeholders. It also serves as a manifestation of the system's most essential design decisions and it works as a transferable and reusable system abstraction.

When defining a specific architecture, the structure being considered should be made explicit and well defined from the start. These can be classified into three categories.

- Modular structures: The system is divided into small units known as modules. This is a static way of looking at the system since it divides the responsibilities of the development teams.

- Component structures and connectors: They focus on how elements interact with each other during execution.
- Mapping structures: They focus on mapping software structures into structures that do not contain any software.

3.2.2. Reference Software Architecture

A Reference Software Architecture (RSA) is a software architecture where the structures, the elements and relations provide templates for concrete architectures in a particular domain or in a family of software systems [13]. The main goal of the RSA is that application software can be specified and implemented independently on where the software is going to be executed.

The RSA provides, on the one hand, a list of functions and their interfaces (APIs) and, on the other hand, the interactions with each of the functions that are outside the scope of the reference architecture.

In the automotive domain two, RSAs have been standardized by the AUTOSAR partnership: the AUTOSAR Classic Platform and the AUTOSAR Adaptive Platform.

3.3. AUTOSAR

In automotive embedded systems, application software has a strong interaction with hardware. The software is embedded in small computers with limited resources distributed all over the vehicle, known as electronic control unit (ECUs). Vehicle functions are becoming more and more complex, especially in autonomous vehicles. Furthermore, real-time and safety requirements shall be fulfilled for this increasingly complex of systems.

AUTOSAR is a standardized **AUT**omotive **O**pen ***S***ystem **AR**chitecture. It is a global development partnership of vehicle manufacturers, suppliers, service providers and companies from the automotive electronics, semiconductor and software industry [7].

AUTOSAR aims to improve complexity management of highly integrated E/E architectures through an increased reuse and exchangeability of SW modules between OEMs and suppliers. It defines the exchange formats and description templates to enable the configuration process of the basic software stack and the integration of the application software into the ECUs. AUTOSAR is the global established standard for software and methodology enabling open E/E system architectures for future intelligent mobility supporting high levels of dependability, especially safety and security [14].

A standardization of the platform software common functionality of automotive embedded software has been achieved specified AUTOSAR. It includes among others, communication stacks, diagnostics services, or operating systems.

It is important to mention that AUTOSAR promotes the cooperation in the definition of standards. The goal is to define common specification language to describe the interfaces between components and communication mechanisms that guarantees a smooth integration of the software in a vehicle. At the same time, it does not prescribe how the software has to be implemented, leaving room for competition between vehicle manufacturers, ECU suppliers and software vendors.

3.3.1. Platform Software

Concerning the specification of the platform software, AUTOSAR has currently standardized two reference software architectures: the AUTOSAR Classic Platform (CP in short) and the AUTOSAR Adaptive Platform (AP in short).

In 2003 AUTOSAR started with the specification of the Classic Platform to support the development of deeply embedded microcontroller-based ECUs. The Classic Platform is well suited for functions with high real time requirements (range of microseconds) and high safety requirements. Embedded microcontroller-based ECUs require low computing power and they can process around 1000 DMIPs.

In 2016, AUTOSAR started the specification of the Adaptative Platform to support the development and integration of application software in high-performance computing platforms based on microprocessor ECUs. These ECUs provide high availability of resources and a soft real-time execution (range of milliseconds), as depicted in Figure 1. The Adaptive Platform has been developed to support the growing evolution of the connected and autonomous vehicle, including the capability to enable the increasing need for software updates.

Both platforms address different use-cases, so the AP is not replacing the CP. This means that both will coexist in an automotive ecosystem. As both platforms provide solutions to different problems, their coexistent offers great added value to the standard.

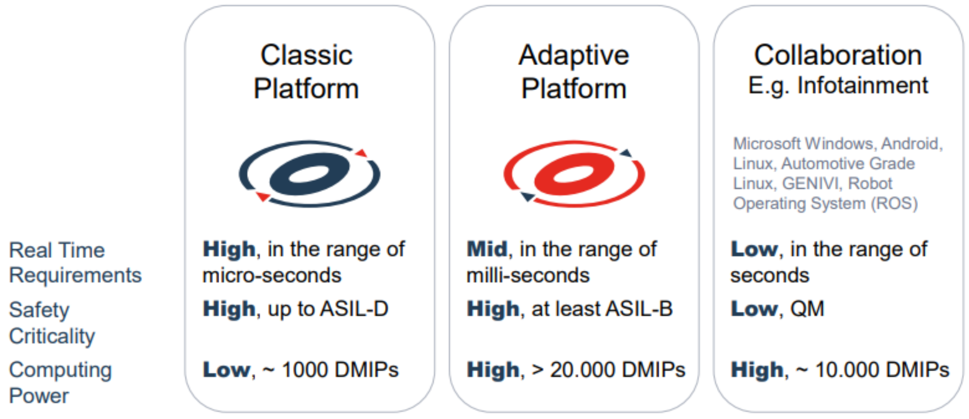

Classic Platform
Adaptive Platform
Collaboration E.g. Infotainment
Microsoft Windows, Android, Linux, Automotive Grade Linux, GENIVI, Robot Operating System (ROS)
Real Time Requirements
Safety Criticality
Computing Power
High, in the range of micro-seconds
High, up to ASIL-D
Low, ~ 1000 DMIPs
Mid, in the range of milli-seconds
High, at least ASIL-B
High, > 20.000 DMIPs
Low, in the range of seconds
Low, QM
High, ~ 10.000 DMIPs

Figure 1. Differences between AUTOSAR platforms [4].

Although the AP is better suited for the implementation of sensor data fusion and obstacle localization, the focus of this master thesis is the specification of a safe architecture for autonomous vehicles. For simplicity, the architecture will be modelled using the Classic Platform basically because of the safety requirements of the Autonomous System (AS) mode manager (see section 3.4.4).

3.3.2. AUTOSAR Classic Platform

The AUTOSAR Classic Platform architecture is a layered software architecture. As depicted in Figure 2, it provides three abstraction layers for the Basic Software (BSW): the microcontroller abstraction layer, the ECU abstraction layer and the Services layer. The Run-Time environment (RTE) abstracts the communication between the application layer and the basic software services.

The most important properties of the Classic Platform layers and modules are summarized in [15]. The application software layer is in most cases hardware independent, the RTE represents the interface for the applications, which is used to communicate with the lower levels.

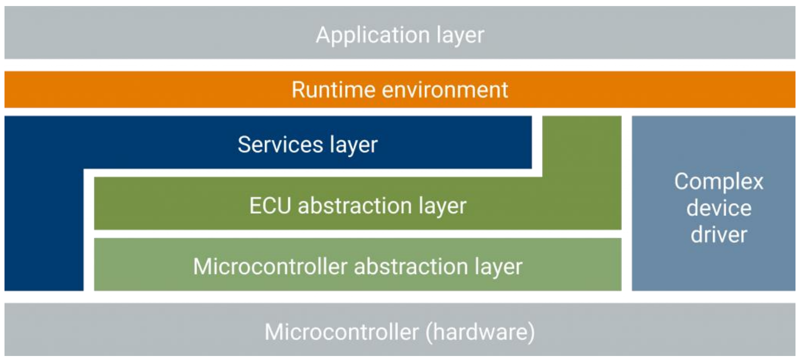


Figure 2. Main layers of the AUTOSAR Classic platform architecture [16].

3.3.3. Virtual Functional Bus

Application software is described by software components (SWCs) in AUTOSAR. SWCs are the building blocks of AUTOSAR and they can be combined to design the functions of a complete system. The communication between the different SWCs is done through ports. Objects are provided by P-Ports and required by R-Ports. Interfaces must be assigned to these ports. They control what can be transmitted and the used semantics.

AUTOSAR defines different communication types. The most frequently used communication mechanisms are Sender-Receiver and Client-Server communication. In Sender-Receiver communication, a SWC sends data to one or more SWCs by pushing the data elements. In Client-Server communication, a SWC implements a function (the server) that can be called by other SWCs (the clients).

The Virtual Functional Bus (VFB) manages the connections and interactions between the SWCs. The VFB isolates the applications from the system infrastructure.

The VFB provides enough information to allow software systems to be integrated and tested before deciding about the allocation of software components to ECUs, i.e., the whole functional behaviour of a system can be prototyped before the electrical architecture (network of the ECUs) is known [17].

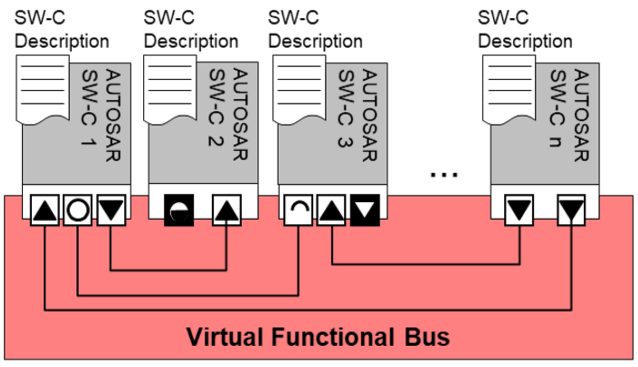


Figure 3. Communication between the SWCs in a VFB [7].

3.4. Safety standards for automated systems

This section describes the different safety standards that apply to automated systems. Currently, there are a variety of standards to be consider and fulfilled to guarantee safety in automated driving systems.

For the development of this project the following standards have been considered:

- V-Model
- SAE Automated driving levels

- ISO-26262
- ISO/TR-4804
- ASAM OpenODD Concept Paper
- ISO/PAS-21448

The subsequent sections briefly describe the topics each standard covers and what they intend to standardize.

3.4.1. V-Model

The V-model is a planning and execution methodology for system development projects established by the German commission. It is highly recommended to follow the phases indicated in the V-model standard to have a systematic development of a project. A graphical description of the V-Model is provided in Figure 4.

The V-Model considers the system entire lifecycle nicely fitting with the systems engineering theory [18]. The model is a graphical representation of a systems development lifecycle that is used to produce accurate development models and project management models. It is also known as Verification and Validation model [19].

The model splits the development process into two parts: project definition (on the left) and project testing and integration (on the right). Each side has multiple stages, each one must be finished before moving on to the next. The project integration and implementation process evaluate and verifies the project definition at each defined stage.

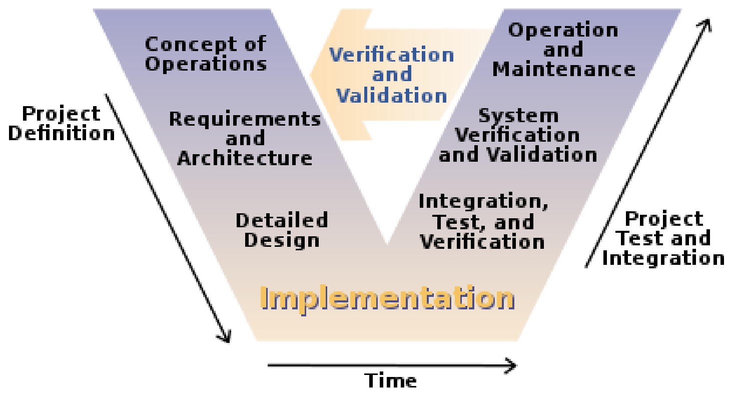

Concept of Operations
Verification and Validation
Operation and Maintenance
Project Definition
Requirements and Architecture
System Verification and Validation
Detailed Design
Integration, Test, and Verification
Project Test and Integration
Implementation
Time

Figure 4. V-model of the systems engineering process [19].

The advantages of implementing this model are as follow: (i) it is simple and easy to use; (ii) planning and designing activities occurs early on, developing a very good understanding of the project at the very beginning and saving time during implementation; (iii) avoids the downward flow of the defects, easy localization of the errors. The disadvantages are: (i) rigidity and lack of flexibility; (ii) since no prototypes are created before implementation, updating all previous levels to each modification is time-consuming and expensive [20].

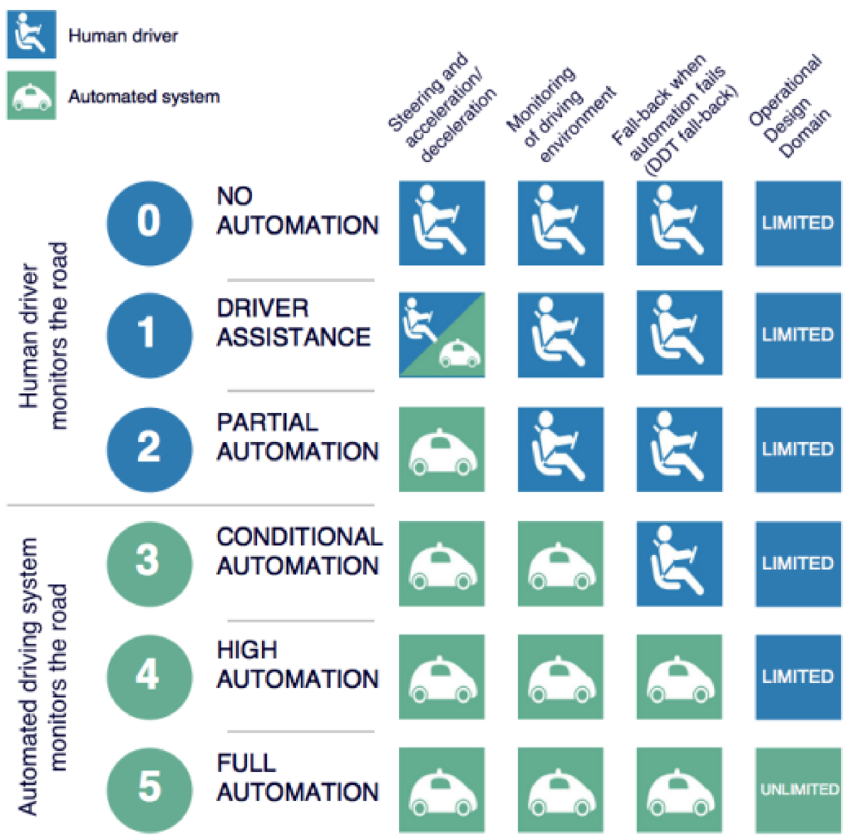
The V-model serves as the foundation for the development of this project. The purpose is to develop the architecture of the minimal autonomous system in a systematic way.

3.4.2. SAE Automated driving levels

The transfer of total control from humans to machines is classified by the Society of Automotive Engineers (SAE) as a stepwise process on a scale from 0 to 5. Level 0 involves no automation and level 5 means full-time performance by an automated driving system of all driving aspects, under all roadway and environmental conditions.

The multiple levels of driving automation are defined in the standard SAE J3016 [21]. The purpose of the standard is to be descriptive and broad about this evolution, but it does not provide strict requirements.

The SAE classification of driving automation for on-road vehicles is meant to clarify the role of a human driver, if any, during vehicle operation. The first discriminant condition is the environmental monitoring agent. In the case of no automation up to partial automation (levels 0-2), the environment is monitored by a human driver, while for higher degrees of automation levels 3-5), the vehicle becomes responsible for environmental monitoring. Figure 5 shows the remaining classification factors used to define each level [21], [22].



🟦 Human driver

🟩 Automated system

			Steering and acceleration/ deceleration	Monitoring of driving environment	Fall-back when automation fails (DDT fall-back)	Operational Design Domain
Human driver monitors the road	0	NO AUTOMATION	🟦	🟦	🟦	LIMITED
Human driver monitors the road	1	DRIVER ASSISTANCE	🟦/🟩	🟦	🟦	LIMITED
Human driver monitors the road	2	PARTIAL AUTOMATION	🟩	🟦	🟦	LIMITED
Automated driving system monitors the road	3	CONDITIONAL AUTOMATION	🟩	🟩	🟦	LIMITED
Automated driving system monitors the road	4	HIGH AUTOMATION	🟩	🟩	🟩	LIMITED
Automated driving system monitors the road	5	FULL AUTOMATION	🟩	🟩	🟩	UNLIMITED

Figure 5. Identification of SAE J3016 levels of driving automation [22]

Another discriminant criterion is the responsibility for dynamic driving task (DDT) fallback mechanisms. 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 5 shows the remaining classification factors used to define each level [21], [22].

3.4.3. ISO-26262

The ISO-26262 “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 [23], [24].

The first edition of the ISO-26262:2011 series was developed using current knowledge of state-of-the-art systems in automotive industry (such as steering, braking and airbag systems, etc.) and it does not completely address very complex, distributed systems and how to deal with availability requirements. The second edition resolves some of these issues, but further interpretations are needed.

Functional safety seeks to prevent any accidents or component failures as a result of inputs, hardware, or environmental changes. Unfortunately, it is unclear how autonomous vehicles shall act if an accident cannot be prevented and which risks to minimize [22].

The scope for the functional safety defined in the standard is only applied for static contexts. The standard ensures safety when the possible environment circumstances are known previously. However, this assumption is completely invalid in autonomous driving systems. The environment for the autonomous systems is dynamic, which makes it impossible to study and define all the possible scenarios to apply functional safety [24].

This means that the ISO-26262 cannot guarantee the complete safety for autonomous driving systems, since it is unable to define in detail all the environment scenarios in which it is going to operate. This is address in the ISO PAS 8800 “Road vehicles – Safety and artificial intelligence” [25], currently under development that will be publicly available specification in late 2023.

3.4.4. ISO/TR-4804

The ISO/TR-4804 “Road vehicles - Safety and cybersecurity for automated driving systems - Design, verification and validation” describes a framework and recommendations for the development, verification, validation, production, and operation of automated driving systems that are focused on safety and cybersecurity derived from worldwide applicable publications. All stakeholders in the automotive and transportation industries can benefit from it. It considers verification and validation methods for automated driving systems focused with level 3 and level 4, according to SAE J3016:2018 [26].

The purpose of this technical report is to provide a generic strategy for dealing with the risks presented by automated vehicles. While this basic approach can be taken as a starting point for safe automated driving, it does not describe a full and safe product.

The standard is designed to supplement current safety-related standards and publications. The document presents a more technical overview of recommendations, guidance and strategies for achieving a positive risk balance and avoiding unreasonable risk and cybersecurity related threats, with a focus on the importance of safety by design [26].

It presents a framework and recommendations for the development, verification, validation, production, and operation of automated driving systems that are focused on safety and cybersecurity. All stakeholders in the automotive and transportation industries can benefit from it.

3.4.5. ASAM OpenODD Concept Paper

ASAM stands for Standardization of Automation and Measuring Systems and it is a non-profit standardization organization founded by the German automotive industry back in 1998. ASAM has a high level of standardization of interfaces, protocols, APIs, data models, data exchange formats and other important aspects of E/E development processes. The standards defined by ASAM are only recommendations, and they do not have any impact on regulatory framework. It is currently active in 7 different domains [27], [28].

The relevant standard for this project is the OpenODD standard, which is defined in the ASAM Simulation domain. This standard is not yet complete. It is currently a concept that will be used to develop a future standard. The main objective is to provide a format that can be used to express Operational Design Domains (ODD) defined for Connected Automated Vehicles (CAVs) for simulation testing [27].

An ODD must be valid for the entire life cycle of the vehicle, as it is part of its safety and operating concept. The ODD chosen for the system will greatly impact 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. All traffic participants, weather conditions, infrastructure, location, time of day, and everything else that has an impact on automated driving are all part of the environment.

The ASAM OpenODD concept project was developed to manage all the ODDs that have been specified for automated driving systems. One of the main goals of the concept project is to define a machine-readable format for defining ODD specifications. This abstract format will allow stakeholders to share, compare and re-use the ODD specifications of systems [27], [29].

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" [21]. Figure 6 depicts the ODD taxonomy defined in the ASAM standard.

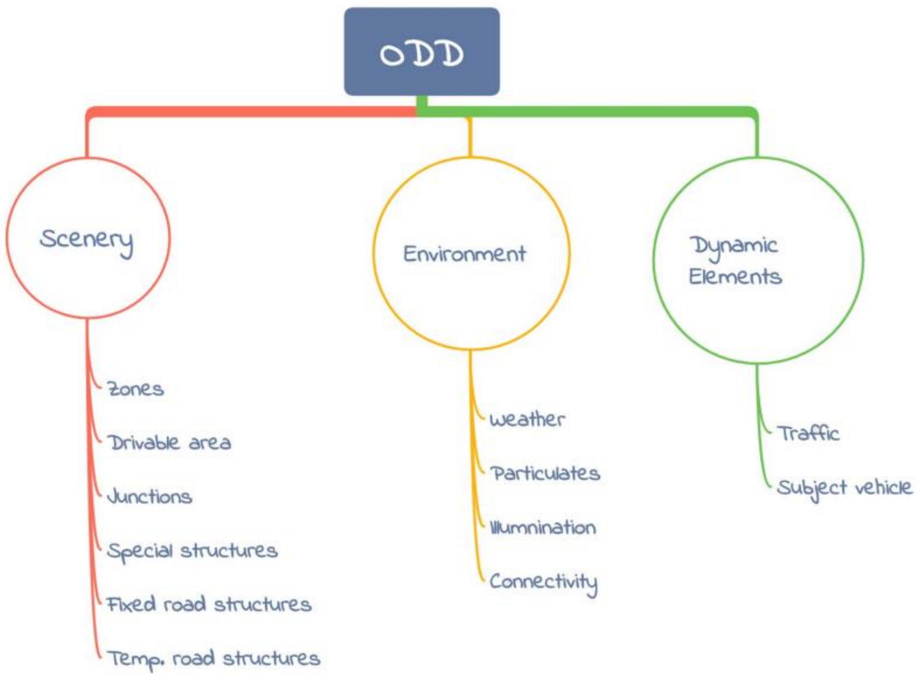


Figure 6. ODD Taxonomy according to BSI PAS 1883 [29].

Figure 7 shows how a system Operation Design Domains is developed based on the scenario in which the vehicle is located.

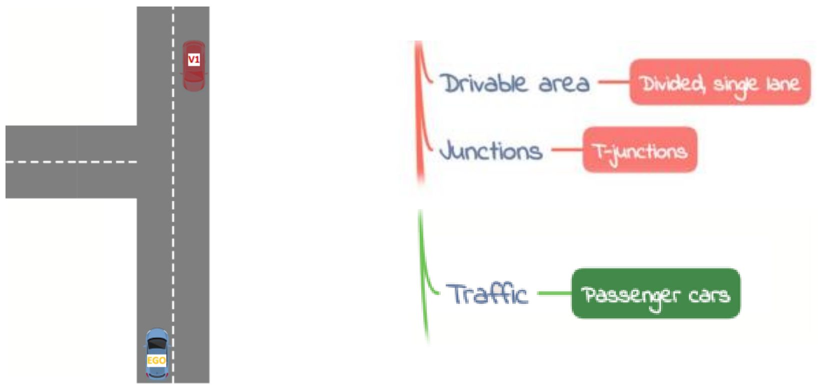


Figure 7. Example definition for an ODD taxonomy [30].

ASAM's strategy is to fill the gaps that now exist, not to do something that another standards organization is doing or to set new standards that contradict the current ones. In this case, the ASAM OpenODD standardization aims to complement the activities of BSI (BSI PAS 1883, which offers the ODD taxonomy) and ISO (ISO 34503 which uses the taxonomy to provide a high-level definition format for ODD).

3.4.6. ISO/PAS-21448

The standard ISO/PAS 21448 “Safety Of The Intended Functionality”, or “SOTIF”, was built specifically to address new safety challenges that are being faced by the developers of the autonomous software in the field of the automotive industry. This standard is as important as the roles of artificial intelligence (IA) and machine learning (ML) in the development of autonomous vehicles [31].

The ISO/PAS 21448 is essential because it has a completely different approach from what is known as functional safety (ISO 26262). Initially, the SOTIF standard was intended to be incorporated as an extra part of the ISO 26262, but after realizing how challenging it is to guarantee safety in the absence of a fault, it was decided to create a separate standard.

Due to the following factors, SOTIF is a crucial standard for the large systems being developed today for automation and artificial intelligence. The first factor is that the correct verification of an automated system is extremely difficult because of the large number of possible scenarios. The second is the fact that automated systems have a large volume of data, which is used in complex algorithms, there is an important challenge for the development of systems with AI and ML. The third and last one is that it also serves to avoid possible risks when the system must make certain decisions.

The following is an example from this project to help understand what SOTIF means: The road is icy and the system is unable to identify the situation. Consequently, the vehicle response will not be adequate for that case, affecting the operational safety of the system, as the vehicle will be moving faster than it should be, even if there is not any system failure.

In other words, the main objective of SOTIF is to reduce the unknown and unsafe conditions that may arise, as in the example above. It is however difficult to prove that all possible scenarios have been considered.

Safety has always been an important topic in the automotive industry. Nowadays, with the introduction of autonomous vehicles, it is becoming an important and critical issue for software development. As in any safety-relevant domain, ensuring functional safety remains a challenging task. The SOTIF provides a guideline where it indicates the different steps to perform the design, verification and validation. With the help of the various defined measures, the system safety can be achieved in non-failure situations.

4. Development

The goal of this project is to model an architecture for a minimal autonomous driving system. The safety standards examined in section 3.4 have been used as a foundation for the model. The key aspect of this 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 architecture for the minimal autonomous driving system has been applied to a case-study for the automated parking system (APS) defined an programmed in [6]. This APS has been developed in parallel to this project so that it integrates smoothly with the architectural design of this master thesis.

The components involved in the case study are the autonomous vehicle, a dedicated mobile App and an External Cloud Service (ECS). The external cloud service component is out of the scope in this project, but it is needed to behave the system properly (provide the free parking spots). The autonomous vehicle and the dedicated mobile app are the system under design for the APS, as shown in Figure 8. The APS definition is presented in Annex E.

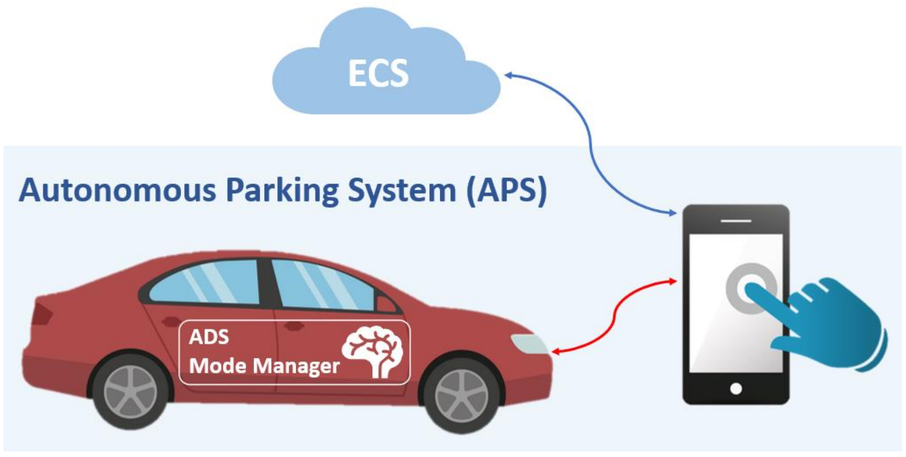


Figure 8. Relation of the ADS mode manager with the APS.

The architecture is however generic and supports incremental development of autonomous vehicle functions. Additional autonomous functions can be introduced by adapting the system requirements and deriving the appropriate modifications in the ODDs (section 3.4.5) and mode declarations groups.

It would have been impossible to develop this minimal autonomous driving system without making use of the systems engineering methodology described in section 3.1, especially because of the inherent complexity of the vehicle as a system. Additionally, the autonomous functionality increases the complexity of the system by interacting with the environment, so that the driver behaviour (another complex, safety relevant system) can be replaced. It has been crucial to apply not only systems engineering but also the design phases of the V-Model described in section 3.4.1. The requirements from the ISO/TR-4804 (section 3.4.4) and the system requirements of the APS (annex E) are the basis for the detailed design.

The logical architecture proposed by the ISO/TR-4804 has been modelled using AUTOSAR, the well-established reference software architecture in the automotive domain (see sections 3.2 and 3.3).

The Virtual Functional Bus (VFB) provides the necessary abstraction for the description of the software architecture [32], [33]. The realization on a technical architecture is not in the scope of this master thesis. However, the complete system has been configured to run in a virtual vehicle, so that it is easy to be integrated in a simulator, i.e., CARLA [34], [35].

The implementation of the VFB is the AUTOSAR Run-Time Environment (RTE), which manages all aspects of communication between the SWCs of the system, including mode management. The AUTOSAR Classic Platform provides a standardized way to describe mode declaration groups, how to model and manage vehicle modes and how to implement the mode-switching [36]. This capability has been ideal for the modelling of the ADS mode manager.

The ADS mode manager is the key element in charge of safety. To ensure the complete safety of the vehicle, the component shall analyse the context that it is in and choose -switching from one to another- the most convenient mode for the analysed context.

The definition and the implementation of the APS in a simulation environment is out of the scope of this master thesis. The relevant information needed for the results of this master thesis is provided in the annexes D and E.

4.1. System Requirements for ADS Mode Manager

The Automated Parking System (APS) shall be able to park and unpark the vehicle autonomously and safely. The ADS mode manager is the component responsible for maintaining the safety of the APS in all possible contexts and it shall fulfil all the system requirements defined in Table 1. All the requirements are derived from the recommendation of the technical report 4804 to ensure the safety [26] and adapted to the APS function.

Table 1. System requirements applied to the ADS mode manager for APS.

System Requirement	Description
SYS-1	The system under design is a component in the high-level architecture defined as Autonomous Driving System (ADS) Mode Manager.
SYS-2	The system shall support different mode groups.

SYS-3	The system shall distinguish when the vehicle drives, parks, unparks and stops.
SYS-4	The system shall support autonomous parking for indoor parking, outdoor parking and land road parking.
SYS-5	The system shall support deactivation of the autonomous parking system.
SYS-6	The system shall activate the safe mode only when the vehicle is parking or unparking. In all other situations, the APS shall be deactivated.
SYS-7	The system shall switch to driver operation when the parking or unparking is deactivated by the user in the APS App.
SYS- 8	The system shall be extendable. New modes should be easily added to the system when needed.
SYS-9	The system shall switch to safe mode once an ODD is being exited and may affect the safety of the system.
SYS-10	The following ODDs are to be recognized by the system. - Different weather conditions - Different environment objects - Different inclinations of the vehicle
SYS-11	The ODD Handling component shall provide the ODD recognized to the system.
SYS-12	The system shall switch to a safe mode if the ODD is not supported.
SYS-12	The system shall react to ODD changes immediately.
SYS-13	The system is (the only) responsible to change the operation mode of the vehicle.
SYS-14	The system shall communicate with the other components in real time.
SYS-15	Each SWC shall report failure to the system.
SYS-16	If one of the vehicle sensors provides an invalid value, the system shall evaluate whether the system is still providing trustworthy objects, otherwise the system must be degraded to a safe mode.

For simplicity in the design of the system and to fulfil the safety demands, we assume the following constrains defined in Table 2.

Table 2. Constrains applied to the ADS mode manager for APS

Constrains	Description
CON-1	When a SWC has a failure, the SWC shall report the status to the system. The actuators and sensor components are able by their own to detect if the transmitted value is invalid.
CON-2	There are no monitoring components.
CON-3	Connection loss with the simulation environment is not considered.
CON-4	The sensor data available is correctly analysed.

4.2. Logical architecture for safety

The software architecture for the automated driving system has been derived from the system requirements defined above in Table 1. Similar to the system requirements, the architecture is based on the information provided by the ISO/TR-4804 to guarantee the safety of the automated system [26].

A classical software architecture for an autonomous system is the sense-plan-act defined in the ISO/TR-4804 and represented in Figure 9 in a very high level. Firstly, the perception is sensed by radars, lidars, cameras, etc. With this information, the system is capable of interpreting its surroundings and predicting the future positions of obstacles. Secondly, it must plan its driving strategy after processing all the information it has received. Finally, it shall move the vehicle using its powertrain, steering, and brakes [37].

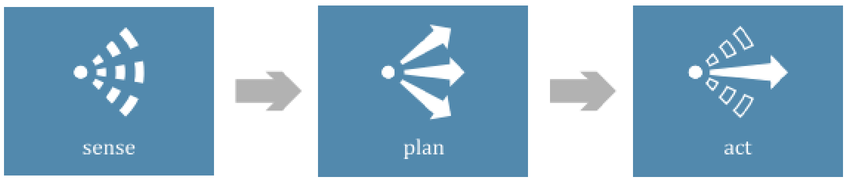


Figure 9. Classical software architecture "sense - plan – act" [26].

The classical architecture discussed above is outdated in the presence of autonomous systems that are currently being developed. This architecture is only useful in a highly controlled environment. A more complicated structure than the one mentioned above will be required if the context is constantly changing. The environment is dynamic, and it is difficult to foresee every situation in which an autonomous vehicle would operate, therefore ISO26262 is not applicable [24].

The system should check and monitor if everything around the system is behaving as it is expected, look at the state of the system and manage the whole system to assure that it is in a safe mode.

In any context (ODD) where it is defined, the system must work safely. As a result, the system is more complex than it is already, but it is also safer and reliable. To start engineering this kinds of systems and exploring new architectures, it is important to see the recommendations and the standards defined by some organizations, such as [26].

The ISO/TR-4804 proposes a generic architecture for an autonomous vehicle showed in Figure 10. The safety-based architecture for the minimal driving system will be based on this presented approach.

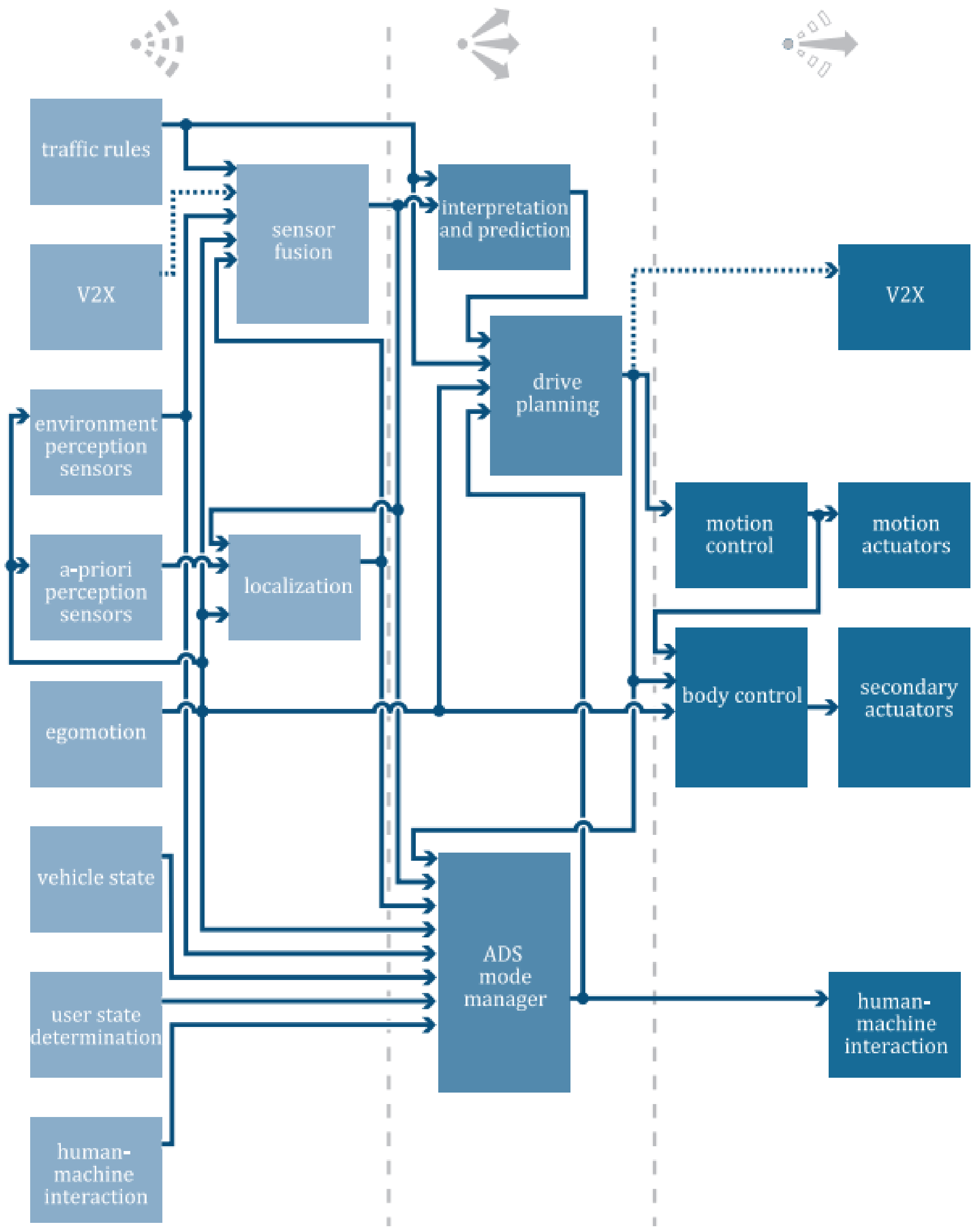


Figure 10. Example of safety-based architecture of the intended functionality [26].

The minimal autonomous driving system for the automated parking is aligned with the Sense - Plan - Act paradigm and the capabilities defined in the standardization. However, we consider only the set of components shown in Figure 11: Perception, Localization, Drive Planning, Vehicle Motion, ODD Handling and the ADS Mode Manager.

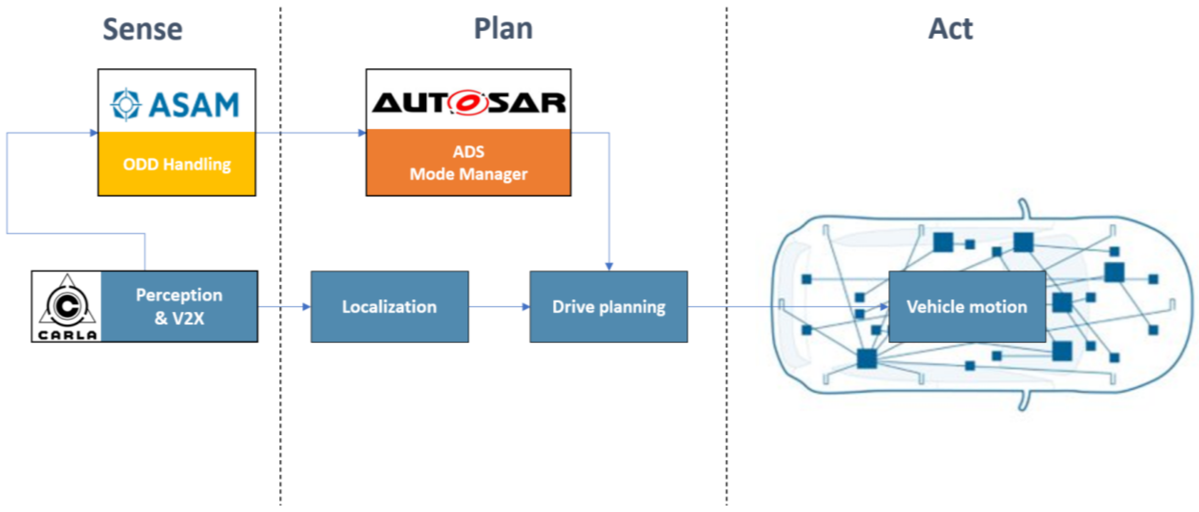


Figure 11. Safety-based architecture for minimal autonomous driving systems.

Some important components, such as prediction, monitoring, etc., have not been considered in order to simplify the architecture, as it is for a minimal autonomous system. The components in the upper level (ODD Handling and the ADS Mode Manager) are the ones that guarantee safety in a dynamic context.

Specifically, this master thesis focuses on modeling the components of the architecture according to the AUTOSAR standard and defining the different components presented in Figure 11. In addition, it illustrates the interaction between the components. Each component is described in the following section, including the data exchanged and the integrated runnable.

4.3. Detailed design

In this section, the definition of each component of the designed safety-based architecture is provided. Moreover, it will be discussed how each component contributes to the APS. To meet the safety requirements, this detailed designed is derived from the ISO/TR-4804 [26], [4].

4.3.1. Perception

Perception is responsible of capturing the data received from the different sensors of the vehicle and identifying all relevant external information to create a world object. This is where the various inputs from the on-board sensors and the optional V2X information are found to generate the actual information of the perception.

The components in perception include lidars, cameras, GNSS for localization, radars and

IMUs sensors. A combination of all these sensors is necessary to satisfy a minimal safe autonomous driving system, since a single sensor is not capable of simultaneously providing reliable and precise detections, classifications, measurements, and robustness to adverse conditions. A V2I infrastructure for the connection with the infrastructure is also included in the Perception component.

4.3.2. Localization

The Localization component is responsible of the identification of the vehicle environment. It is very important that the automated vehicle localizes itself and the environment appropriately and precisely only using the information provided by the Perception component. Each sensor on the vehicle provides information, then the Localization component is responsible for analyzing and fusing the data. Based on this information, it should be able to model the vehicle environment.

The Localization component has to be able to provide to the Drive Planning component the necessary information of the surrounding objects that have to be considered before planning any motion of the vehicle.

4.3.3. Drive Planning

The Drive Planning component establishes the function manoeuvre that must be carried out to complete the upcoming driving step without causing a collision. The component must handle location information, adhere to traffic rules, consider ego-motion, and adapt the functionality according to the switches provided by the Mode Manager.

The Automated Driving System obeys traffic rules so that the driving planning element produces a legal driving plan. Only collision avoidance manoeuvres can override traffic laws to avoid a crash.

As for the APS system to be developed, the model has two different parts for the movement, which are autopilot and parking manoeuvres. The autopilot mode is responsible for driving the vehicle to the parking space, while the parking manoeuvres mode is responsible for parking the vehicle in the parking space with the appropriate manoeuvre. To simplify the implementation, the autopilot will be included in each parking defined function.

There will be four different functions for the manoeuvres, two for parking the car and two for unparking it: (i) *PM_Forwards*, (ii) *PM_ForwardBackwards*, (iii) *UM_Backwards*, (iv) *UM_BackwardForwards*. The arguments of these runnable are: the localization of the slot (x,y,z), the parking side (right or left), the parking type (parallel or angular parking) and the parking angle (only for the angular parking). Figure 12 shows the defined parking types.

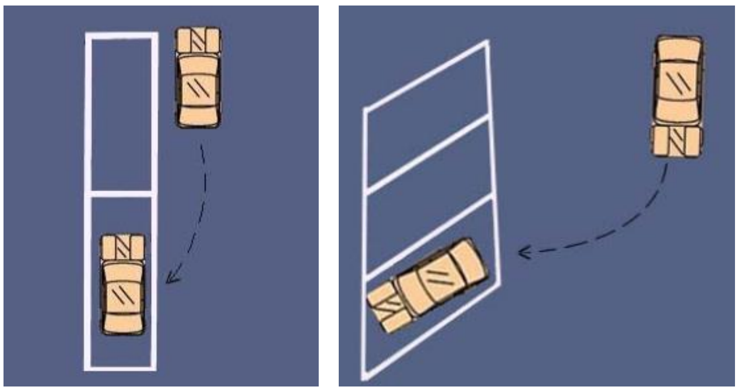


Figure 12. Defined parking types (parallel and angular types) [38] .

The *PM_Forwards* and *UM_Backwards* functions are used for parking and unparking in angular spots respectively, meanwhile the *PM_ForwardBackwards* and *UM_Backward-Forwards* are used for parallel slots.

In case something unexpected happens while the minimal autonomous system is running, the vehicle must enter in a safe mode and activate the necessary functions to ensure the safety of the vehicle. The function defined for this safe mode is called *M_Safe*.

The specific definition and the implementation of the defined functions above in the simulation environment is out of scope of this master thesis, but it has been done and implemented in my co-worker thesis [6].

4.3.4. AS Motion

The Automated System Motion component refers to its translation and rotation around the three axes (i.e. longitudinal, lateral and vertical and roll, yaw, and pitch [39]). To implement the desired motion in the vehicle, the actuator commands must be derived from the Drive Planning component.

The vehicle motion controller must be stable and able to balance the dynamic changes that may occur in the vehicle during the manoeuvres. The generated commands control the steering, brakes and powertrain in order to move the vehicle as planned. Like the Drive Planning implementation, this component is out of the scope of this master thesis.

4.3.5. ODD Handling

The ODD Handling component manages to identify in which context (ODD) the system is placed analysing the different information provided by the Perception component. The operation design domains for the project are defined according to the standard PAS 1883:2020 [40]. The standard provides all possible attributes of the ODDs that may be required by the ADS and gives different examples of what the taxonomy should look like.

A first selection is imposed by the use cases and the different constraints defined for the APS (see annex E). Consequently, the table is reduced as shown in Table 3. The capability column shows which context is relevant or influences to the automated parking system.

Table 3. Supported ODD's for the APS definition and based on the PAS 1883:2020 [40].

ODD taxonomy		Attribute	Sub-attribute	Capability
Scenary	Zones	Geo-fenced areas		Yes
		Traffic management zones		No
		School zones		No
		Regions or states		No
		Interference zones		Yes
	Drivable area	Type	Indoor parkings	Yes*
			Outdor 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, earth, sand)	No
			Segmented	No
			Uniform (Asphalt)	Yes
	special structures	Pedestrians crossings		Yes
		Bridges		No
		Rail crossings		No
		Tunnels		No
	fixed road structures	Buildings		Yes
		Street lights		Yes
		Street furniture		No
		Vegetation		No
Environmental conditions	weather	Water retentions on the slot		Yes
		Wind		Yes
		Rainfall		yes
		fog		Yes
		Sunny		Yes
	Ilumination	Day		Yes

		Night / low ambient lighting		Yes
		Cloudiness	Clear	Yes
			Partly cloudy	Yes
			Overcast	Yes
		Artifical ilumination		Yes
Dynamic elements	traffic	Parked vehicle		Yes
		Pedestrians		Yes
		Presence of special vehicles		Yes
		On road vehicles		Yes

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

A second selection is imposed by the simulation environment chosen for the integration, since it is not able to simulate all the contexts defined in the standard. As this is an educational project, just some of the ODDs are selected to simplify the implementation of the project. Table 4 shows the supported ODDs for the APS that have been modelled in the AUTOSAR tooling. Each of the chosen ODDs is relevant to the APS system.

Table 4. Selected ODDs for the implementation of the APS in CARLA simulator.

ODD taxonomy		Attribute	Sub-attribute	Capability
Scenery	Derivable Area	Type	Urban roads	Yes
		Geometry - Longitudinal plane	Up-slope	Yes
			Down-slope	Yes
			Level plane	Yes
		Surface type	Loose (gravel, earth)	Yes
			Uniform (Asphalt)	Yes
	fixed road structures	Buildings		Yes
		Street lights		Yes
	Special structures	Pedestrians crossing		Yes
Environmental conditions	weather	Water retentions on the slot		Yes
		Wind		Yes
		Rainfall		Yes
		fog		Yes
		Sunny		Yes
	Ilumination	Day		Yes
		Night / low ambient lighting		Yes
		Cloudiness	Clear	Yes
			Partly cloudy	Yes
			Overcast	Yes
		Artifical ilumination		Yes
Dynamic elements	traffic	Parked vehicle		Yes
		On road vehicles		Yes

The ODDs are then mapped to the modes specified in section 4.3.6.3. But first, it is important to explain how the CARLA simulator recognizes the specified contexts, i.e., what APIs the CARLA simulator uses to determine whether the vehicle is in the specified ODD. This is crucial for defining the interfaces to extract the data of the context from the simulator.

The CARLA simulator allows the definition of the world, i.e., the objects created by the client to appear in the simulation, this instance is called CARLA.World. The world contains the active map that is visible to us, i.e., the world assets, not the navigation map. The map also manages the weather and the present actors. Each simulation can only have one world, but it is always subject to change. There are various functions that can be used to know about the defined environment while the simulation is running. These ones will be used to know about contexts [41].

The following functions are called in the scenario to identify the contexts: (i) *Get_environment_objects* (self, object_type=Any), (ii) *Get_weather* (self) and (iii) *get_turned_on_lights* (self, light_group).

Cloudiness, rain, wind, and sun position are among the weather parameters currently active in the simulation and they are provided by the *Get_weather* (self) function. The weather information consists of a total of 14 float parameters. Each variable indicates a weather condition with a set value. The variables will be considered as binary, so if their value is greater than half, it indicates that the weather condition is occurring. Since for the project, it is only important if the weather condition exists, not its reference value.

The Get_environment_objects (self, object_type=Any) function returns a list of environment objects with the requested semantic tag [42]. By default, the method returns any object in the environment, but the object type argument allows the request to be filtered by semantic tags.

If a filter has been defined, the response is an array containing all filtered elements that have been found in the real world. The array is composed of a variable containing the ID, a name (the semantic tag defined by the simulator) and the location of the found object [43]. Each ODD that requires this function will have its semantic tag specified and filtered to determine whether it appears in the real world.

A list of the lights turned on in the scene that has been filtered by group is returned by the *get_turned_on_lights* (self, light group) method [44]. The return of this function is an array containing the information following for each recognized light: the identifier of the light, a Boolean indicating if it is on and the group of light that belongs [45]. Each ODD that requires this function will have its light group specified and filtered to determine whether it appears in the real world.

Additionally, the IMU radar is required to determine the inclination. The compass variable, which gives the car inclination with regard to north, is the only variable used in this situation.

The specific CARLA simulator variables and functions that provide the necessary data to calculate a vehicle ODDs are listed in Table 5.

Table 5. CARLA functions to define specific ODDs.

Sub-attribute	Function	Variable (Semantic Tag)
Urban roads	*Get_environment_objects*	Road Lines (7)
Up-slope	*IMU radar*	Compass
Down-slope	*IMU radar*	Compass
Level plane	*IMU radar*	Compass
Loose (gravel, earth)	*Get_environment_objects*	Terrain (22)
Uniform (Asphalt)	*Get_environment_objects*	Terrain (22)
Buildings	*Get_environment_objects*	Building (1)
Street lights	*Get_environment_objects*	Traffic Light (18)
Pedestrians crossing	*Get_environment_objects*	Pedestrians (4)
Water retentions on the slot	*Get_weather*	Precipitation deposits
Wind	*Get_weather*	Wind intensity
Rainfall	*Get_weather*	Precipitation
fog	*Get_weather*	Fog density
Sunny	*Get_weather*	Sun altitude angle
Day	*Get_weather*	Sun altitude angle
Night / low ambient lighting	*Get_weather*	Sun altitude angle
Sky clear	*Get_weather*	Cloudiness
Partly cloudy	*Get_weather*	Cloudiness
Overcast	*Get_weather*	Cloudiness
Artifical ilumination	*get_turned_on_lights*	Streetlight
Parked vehicle	*Get_environment_objects*	Vehicles (10)
On road vehicles	*Get_environment_objects*	Vehicles (10)

4.3.6. ADS Mode Manager

The ADS Mode Manager plays a crucial part in the system safety, having a central role of this master thesis. This component fulfils the task of safely switching between manual driving modes and automated driving modes.

The Mode Manager must confirm that all conditions and prerequisites, such as ODD, have been met and they are the right ones before switching to another mode (e.g., if the vehicle is on the right road, if the weather conditions allow the transition of the mode, etc.).

4.3.6.1. Mode management groups

According to the APS definition, various groups of modes have been designed to address all potential situations of the automated parking system.

The Mode Manager alternates between the defined modes once the APS is turned on. The three groups of modes that need to be considered are described and specified below: (i) Vehicle Mode, (ii) APS Mode and (iii) Parking Location Mode. Each specified mode has a value to be identifiable inside its mode group.

The first mode group, Vehicle Mode, lists the four possible driving modes that a vehicle may have, as shown in Table 6.

Table 6. Definition of the Vehicle Modes.

Group mode 1		Definition	Value
Vehicle Mode	VEH_DRIVE	The vehicle is being driven autonomously.	0
	VEH_PARKING	The vehicle is parking autonomously.	1
	VEH_UNPARKING	The Vehicle is unparking autonomously.	2
	VEH_STOPPED	The vehicle is stopped, and the engine is off.	3

The second mode group, APS Mode, defines the five different modes that the automated parking system can manage during the operation of the vehicle, as shown in Table 7.

Table 7. Definition of the Automated Parking System Modes.

Group mode 2		Definition	value
APS Mode	APS_OFF	APS is deactivated.	0
	APS_INDOOR	APS ready for parking manoeuvres in indoor car parks.	1
	APS_OUTDOOR	APS ready for parking manoeuvres in outdoor car parks.	2
	APS_LANDROAD	APS ready for parking manoeuvres in land-road car parks.	3
	APS_SAFEMODE	APS ready for an emergency.	4

When the system recognizes that it has reached a condition that could threaten the vehicle safety, such as when a solid object is in the path of a potential crash or when a crash has occurred. To avoid worse results, the Mode Manager will take care of switching the vehicle into a safe mode.

The ADS Mode Manager computes the APS and Vehicle modes, while the ECS supplies a third mode group. This third mode group, Parking Location Mode, provides the type of the parking space selected for parking, as shown in Table 8.

Table 8. Definition of the Parking Location Modes.

Group mode 3		Definition	value
Parking Location Mode	STREET_PARALLEL	The spot is on the side of the street and is parallel type.	0
	STREET_ANGULAR	The spot is on the side of the street and is angular type.	1
	PARKING_OUTDOOR	The spot is inside an outdoor/open-air car park.	2
	PARKING_INDOOR	The spot is inside an indoor/closed-air car park.	3
	PARKING_ROAD	The spot is on the side of the road (only parallel).	4

It is crucial to understand that there is only one type of parking a car in a car park, whether it is indoors or outdoors: the angular type. A car park is a closed area reserved exclusively for parking with several parking spots. Single parking spots on the side of the street are excluded from this definition. The parking options in the CARLA simulator are shown in Figure 13.

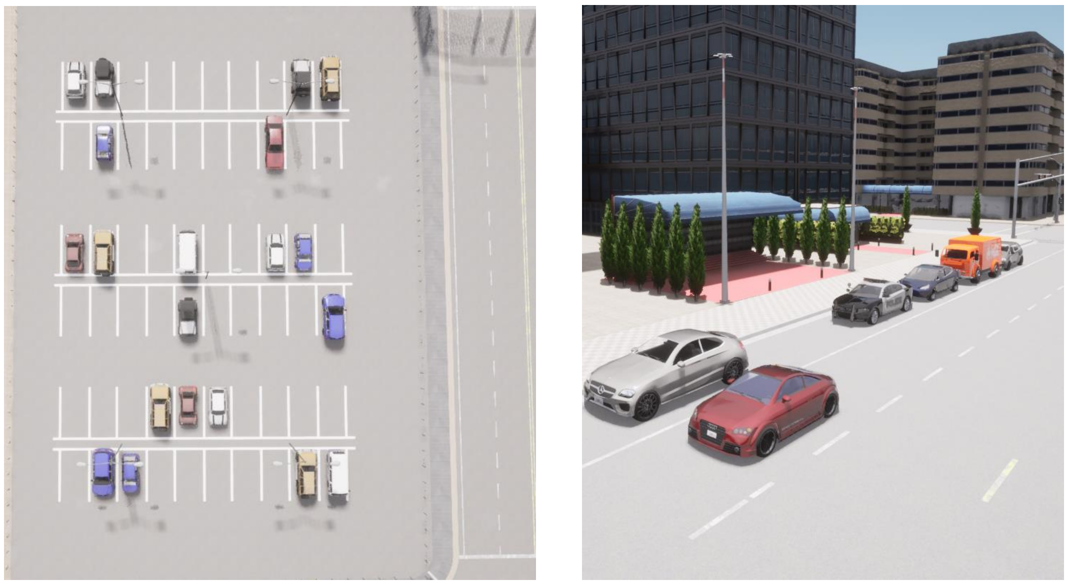


Figure 13. Types of parking spot in CARLA: Car park and street parking.

Additionally, it should be noted that in a car park, sensors can always be used correctly if they are available, regardless of the weather. Whereas in street parks, depending on the external context, a sensor may be available, but it cannot be used. This is the reason for having the different APS modes.

A clear illustration of this would be the fact that on a snowy day, the line detection system would not be able to detect the white lines on the floor of a street parking spot, but it would be able to detect them in a car park, as it has the staff to adapt the car park to any situation.

4.3.6.2. Mode switching based on function manoeuvres

It is important to specify which runnable defined in the Drive Planning component should be called according to each mode that the APS can manage to assure the system safety. Moreover, the parking spot type chosen for the parking manoeuvre shall be taken into account.

Figure 9 shows the mapping for the parking operations. This information must be modelled in the internal behaviour of the Drive Planning component so that the execution of the correct runnable is derived from the architectural design. In addition, this simplifies the implementation of the APS, since only the five functions defined above for parking and unparking need to be programmed. Everything else will be decided by the architecture itself.

Table 9. Mode switch for defining the manoeuvre functions for the Drive Planning.

Parking Manouver Runnables	STREET_PARALLEL	STREET_ANGULAR	PARKING_OUTDOOR	PARKING_INDOOR	PARKING_ROAD
APS_OFF	disabled	disabled	disabled	disabled	disabled
APS_INDOOR	disabled	disabled	disabled	PM_Forward	disabled
APS_OUTDOOR	PM_ForwardBackwards	PM_Forward	PM_Forward	disabled	disabled
APS_LANDROAD	PM_ForwardBackwards	disabled	disabled	disabled	PM_ForwardBackwards
APS_SAFEMODE	M_Safe	M_Safe	M_Safe	M_Safe	M_Safe

Table 10 shows the mapping carried out for the unparking operations.

Table 10. Mode switch for defining the manoeuvre functions for the Drive Planning.

Unparking Manouver Runnables	STREET_PARALLEL	STREET_ANGULAR	PARKING_OUTDOOR	PARKING_INDOOR	PARKING_ROAD
APS_OFF	disabled	disabled	disabled	disabled	disabled
APS_INDOOR	disabled	disabled	disabled	UM_Backward	disabled
APS_OUTDOOR	UM_BackwardFordwards	UM_Backward	UM_Backward	disabled	disabled
APS_LANDROAD	UM_BackwardFordwards	disabled	disabled	disabled	UM_BackwardFordwards
APS_SAFEMODE	M_Safe	M_Safe	M_Safe	M_Safe	M_Safe

As a result, the stated system is more scalable, as in case there is an interest in adding a new mode or a new parking location, the system simply needs to indicate which function to activate for each given mode and location. No new code needs to be programmed.

4.3.6.3. Mode switching based on context

The system shall specify which contexts selected in the ODD Handling are acceptable for each of the Automated Parking System modes defined above. The Mode Manager shall not permit the activation of a new mode if the context does not allow it. This ensures the system safety in any context, which is the main purpose of the architecture. Different weather scenarios of the above-described ODDs are shown in appendix D.II.

Table 11 maps the selected ODDs in section 4.3.5 to the vehicle mode groups. For the following tables, the *x* indicates that the mode is compatible with the related context. In this case, the vehicle can drive autonomously in almost all contexts, but the parking and unparking modes are a slightly more restricted by gradients and dark areas.

Table 11. ODD mapping with vehicle mode group.

ODD taxonomy		Attribute	Sub-attribute	VEH_DRIVE	VEH_PARKING	VEH_UNPARKING	VEH_STOPPED
Scenary	Drivable area	Line type	Road lane	x	x	x	x
		Geometry - Longitudinal plane	Up-slope	x			x
			Down-slope	x			x
			Level plane	x	x	x	x
		Surface type	Loose (gravel, earth)				x
			Uniform (Asphalt)	x	x	x	x
	Fixed road structures	Buildings		x	x	x	x
		Street lights		x	x	x	x
	Special structures	Pedestrians crossing		x	x	x	x
Environmental conditions	Weather	Water retentions on the slot		x			x
		Wind		x	x	x	x
		Rainfall		x	x	x	x
		fog		x	x	x	x
		Sunny		x	x	x	x
	Ilumination	Day		x	x	x	x
		Night / low ambient lighting		x			x
		Cloudiness	Clear	x	x	x	x
			Partly cloudy	x	x	x	x
			Overcast	x	x	x	x
		Artifical ilumination		x	x	x	x
Dynamic elements	Traffic	Parked vehicle		x	x	x	x
		On road vehicles		x	x	x	x

Focusing on the parking and unparking modes, Table 12 maps the ODDs to the modes of the APS mode group. Logically, the *APS_OFF* mode supports all contexts, as the system shall be disabled. Additionally, as already mentioned, indoor parking spots offer the benefit of being adaptable to all weather situations. However, the outdoors slots will be more limited to bad weather, as shown in Table 12.

Table 12. ODD mapping with the APS mode group.

ODD taxonomy		Attribute	Sub-attribute	APS_OFF	APS_INDOOR	APS_OUTDOOR	APS_LANDROAD	APS_SAFEMODE
Scenary	Drivable area	Line type	Road lane	x		x	x	x
		Geometry - Longitudinal plane	Up-slope	x				x
			Down-slope	x				x
			Level plane	x	x	x	x	x
		Surface type	Loose (gravel, earth)	x				x
			Uniform (Asphalt)	x	x	x	x	x
	Fixed road structures	Buildings		x		x		x
		Street lights		x	x	x		x
	Special structures	Pedestrians crossing		x	x	x		x
Environmental conditions	Weather	Water retentions on the slot		x	x			x
		Wind		x	x	x	x	x
		Rainfall		x	x			x
		fog		x	x			x
		Sunny		x	x	x	x	x
	Ilumination	Day		x	x	x	x	x
		Night / low ambient lighting		x				x
		Cloudiness	Clear	x	x	x	x	x
			Partly cloudy	x	x	x	x	x
			Overcast	x	x			x
		Artifical ilumination		x	x	x	x	x
Dynamic elements	Traffic	Parked vehicle		x	x	x		x
		On road vehicles		x	x	x	x	x

The influence of the surroundings, such as buildings, cars, pedestrians, traffic signs and lights, etc., is what distinguishes *APS_OUTDOOR* from *APS_LANDROAD*. Particularly, these surrounding objects are not considered in *APS_LANDROAD*. Therefore, if these objects are in the context, it means that the vehicle is not located on a land road, then the appropriate mode in this situation would be *APS_OUTDOOR*.

4.3.6.4. Implementation of the ADS Mode Manager

The ADS Mode Manager for the Automated Parking System function is implemented as the state machine shown in Figure 14.

The variables required to evaluate the transitions between the states of the state machine are the following ones: (i) *APS_activation*, (ii) *Parking/Unparking*, (iii) *Context_Data*, (iv) *Parking_Location_Mode* and (v) *APS_Done*. Except for *Context Data* and the *Parking_Location_Mode*, all the variables are Boolean type and defined in Table 13.

Table 13. Definition of the Boolean variables.

Variable	Description
APS_Activation	Indicates weather the user has activated the APS. Value 0 means APS is off.
Parking/Unparking	Indicates weather to park or unpark the vehicle. Value 0 means parking.
APS_Done	Indicates if the APS performance has finished. Value 1 means it is done.

The information provided by the ODD Handling described in subsection 4.3.5 is contained in the *Context_Data* variable. The *Parking_Location_Mode* variable is an integer, each number from 0 to 4 represents a different mode as defined in section 0.

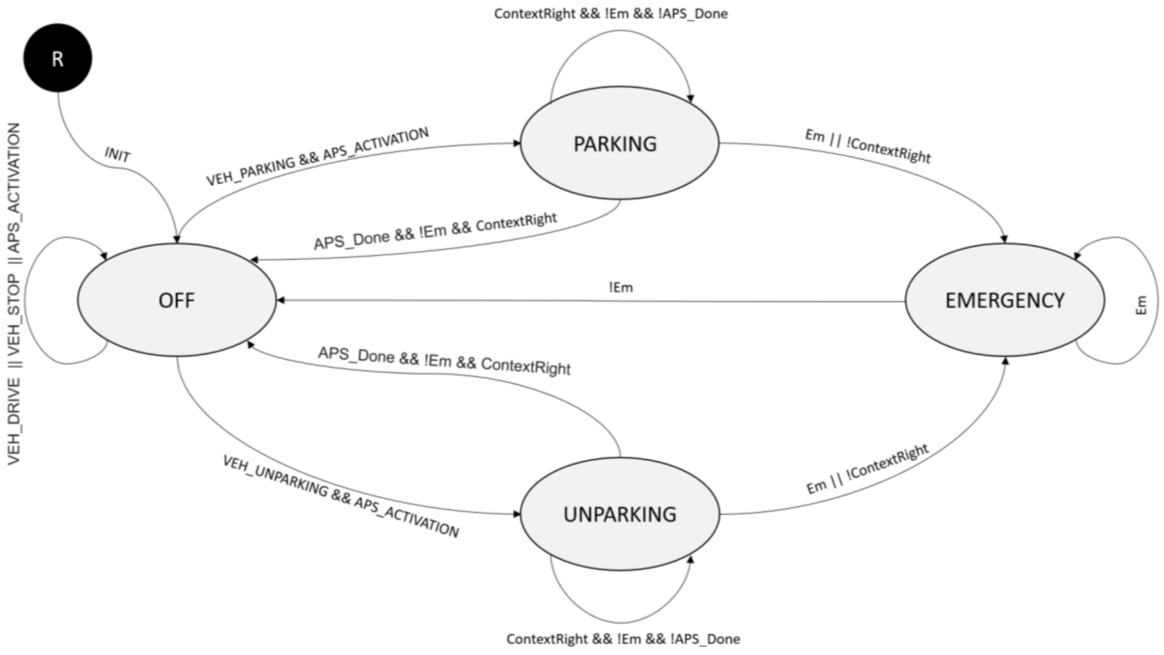

ContextRight && !Em && !APS_Done
R
INIT
VEH_PARKING && APS_ACTIVATION
PARKING
Em || !ContextRight
VEH_DRIVE || VEH_STOP || APS_ACTIVATION
APS_Done && !Em && ContextRight
OFF
!Em
EMERGENCY
Em
APS_Done && !Em && ContextRight
VEH_UNPARKING && APS_ACTIVATION
UNPARKING
Em || !ContextRight
ContextRight && !Em && !APS_Done

Figure 14. ADS Mode Manager state machine.

The OFF state selects the APS Mode based on the Parking Location Mode given by the ECS, as depicted in Table 14.

Table 14. Selection of the APS Mode based on the Parking Location Mode.

Parking Location Mode	Respective APSMode
STREET_PARALLEL	APS_Outdoor
STREET_ANGULAR	APS_Outdoor
PARKING_OUTDOOR	APS_Indoor
PARKING_INDOOR	APS_Indoor
PARKING_ROAD	APS_Landroad

The parking and unparking state are quite similar. In both states, the context data is checked to verify if it is appropriate for the active APS mode. The internal variable *ContextRight* is enabled if the context is suitable. Otherwise, it is deactivated. The context data provided from CARLA simulator is compared with those from the previous loop to identify changes in the context. Once changes are detected, *ContextRight* variable should be updated.

The difference between these states is the Vehicle Mode selected. In Parking state, the *VEH_Parking* mode is turned on. In Unparking state, the *VEH_Unparking* mode is set. Finally, the emergency state activates the *APS_SafeMode* to ensure safety during the emergency.

As a result, the ADS Mode Manager is able to provide to the Drive Planning which mode is active for each mode group defined in section 4.3.6.1. Emergency (*Em*) is activated when the Localization components detect that some obstacle is compromising the safety of the system.

4.4. Vehicle modelling

This section describes the implementation of the autonomous vehicle according to the architecture. For modelling the vehicle, it is crucial to know all the different sensor modules that the vehicle integrates, as well as the information being exchanged by these sensors.

The implementation of the APS function is carried out in a simulation environment, specifically in the CARLA simulator. CARLA is an open-source autonomous driving simulator described in annex D [35]. The CARLA vehicle is the hardware that configures the system in this simulation environment. For pragmatic reasons, reverse engineering has been employed to determine the configuration of the vehicle so that all sensor types are available at integration time.

The sensors that can be used in the CARLA simulator and relevant in the automated parking system are the following: (i) Cameras Sensor, (ii) GNSS Sensor, (iii) IMU Sensor, (iv) Lidar Sensor, (v) Radar Sensor and (vi) Semantic Lidar Sensor [46]. In addition, as it is a simulation environment, it is possible to use as many sensors as one wishes, it is only necessary that they are defined in the World instance of the CARLA simulator.

In order to model the vehicle as realistic as possible, the number of sensors and types used in current autonomous vehicles were analysed [47], [48]. The configuration chosen for the vehicle implementation is displayed in Figure 15.

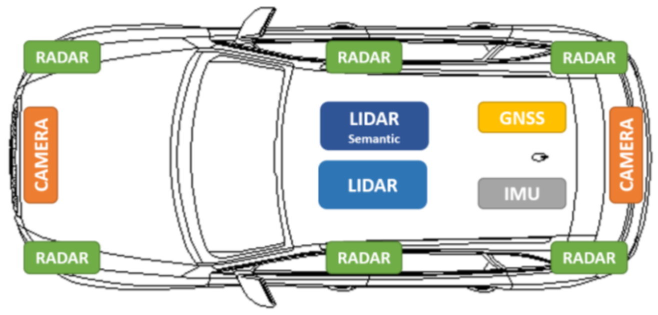


Figure 15. Positioning of the sensor modules for the modelling of the vehicle hardware.

AUTOSAR has standardised most commonly used sensor types so that sensors can be used by vehicle manufacturers and suppliers in a simple way without having to model each sensor. This is one of the goals of the standard, so that everyone ends up using the same modules [49]. After analysing the APIs of the sensor modules offered by the CARLA simulator, it was clear that they were not compatible with the AUTOSAR interfaces. Therefore, all the APIs of the CARLA sensor modules have been modelled as AUTOSAR interfaces, so the architecture is compatible with the simulation environment to be used for the integration.

The modelling of the sensor interfaces used in CARLA should ease the task for future developers who want to integrate an AUTOSAR architecture in the CARLA simulator. The following sections define each sensors modules to get a clear understanding about the type of data it exchanges. The definition is based on the information provided by the CARLA simulator [46].

4.4.1. Radar sensor

Radar sensors are conversion devices that transform microwave echo signals into electrical signals. Unlike other sensors, radar sensors are not affected by light and darkness and with the ability to detect obstructions like glass, it can “see” through walls and it is not affected by weather conditions. One of the biggest advantages radar sensors have over other sensors is its detection of motion and velocity of the surrounding objects [50], [51].

To have the capability of sensing all objects around the vehicle it is necessary to have more than one sensor radar installed, at least on at each edge of the vehicle. However, to guarantee the detection of objects on the laterals, we have decided to implement one sensor per side. Figure 15 above shows the exact location of each sensor module: two at the front, two in the middle and two at the end.

The sensor module used in the CARLA simulator has four instantiated variables, as shown in Table 15, where the variable, units and type are defined [52].

Table 15. Instance variables for the radar module defined in the CARLA simulator [52].

	Variable	Units	Type
1	Altitude angle	rad	Float
2	Azimuth angle	rad	Float
3	Depth	mts	Float
4	Velocity	m/s	Float

4.4.2. Camara sensor

The camera sensor module allows the vehicle to see in high resolution and recognise all the objects detected by the radar sensors that the vehicle is equipped with. It is important to consider that cameras do not work effectively in all weather conditions, and unlike radar and lidar, which provide numerical data that is easily analysed, camera technology requires more sophisticated computation to analyse the images to understand the environment and the objects it sees [53].

The cameras have a very wide viewing angle, reaching up to 120 degrees [54]. Unlike sensors, this allows us to capture and identify most of the components surrounding the vehicle with only one camera in front and other at the back. Unlike sensors, this means that with two cameras, placed as shown in section 4.4, it is possible to capture and identify the components surrounding the vehicle.

The camera sensor module has four instance variables and two defined methods, which act as a function with arguments [55]. Table 16 shows the instance variables provided by the camera module.

Table 16. Instance variables for the camera module defined in the CARLA simulator [55].

	Variable	Units	Type	Description
1	Fov	Degrees	Float	Horizontal field of view of the image
2	Height	Pixels	Int	Image heigh
3	Width	Pixels	Int	Image width
4	Raw_data	-	Bytes	Array of RGBA 32-bit pixels.

The defined methods are needed because the module needs information from the vehicle to provide data in the required type and format. This module requires the following methods:

- The first method is called Convert and converts the image following the colour converter pattern (*colour_converter*). The argument is the *color_converter* pattern defined in the CARLA simulator [56].

- The second method is named *save_to_disk* and saves the image obtained in a path using a converter pattern stated as *colour_conveter*. The arguments are the path (string) that will contain the image and the *colour_converter* pattern [56].

4.4.3. Lidar sensor

LIDAR stands for “Light Imaging Detection and Ranging” and it can identify objects around it, since it measures shape, size and distance. This module uses laser light pulses to scan the environment, unlike radar radio waves. Specifically, the LIDAR module shoots millions of laser signals, which are reflected on the surfaces of objects around and returned to the receiver incorporated on the module. With the information obtained, the LIDAR is able to create a 3D model of the vehicle surroundings [53].

Lidar can identify objects in a higher resolution than radar. Nevertheless, it is the most expensive option for the OEMs. In addition, it is limited by weather conditions. The module tends to degrade in adverse weather conditions such as fog, rain and snowfall [57].

An autonomous vehicle relying on LiDAR should be able to assess in a real-time manner its limitations and raise an alarm in such scenarios to ensure safety. Currently, vehicles developed to implement autonomous systems are equipped with a single large 360-degree LIDAR sensor on the roof that provides a complete view of the surroundings [48]. Therefore, it was decided to model the vehicle with a single LIDAR sensor.

The LIDAR sensor module has six instance variables and one defined method [58]. The interface needed for the camera sensor is shown in Table 17.

Table 17. Instance variables for the LIDAR module of CARLA simulator [58], [59].

	Variable		Units	Type	Description
1	Channels		-	Int	Number of lasers shot
2	Horizontal_angle		Radians	Float	Horizontal angle of rotation at the measure.
3	Raw_data	x	meters	Float	Distance from origin to spot on X axis
4		y	meters	Float	Distance from origin to spot on Y axis
5		z	meters	Float	Distance from origin to spot on Z axis
6		intensity	-	Float	Computed intensity for this point (x,y,z) as a scalar value between [0.0 , 1.0].

The Lidar module requires only one method to work properly. Same as the *save_to_disk* from the camera sensor [58].

4.4.4. Semantic lidar sensor

This sensor module simulates a rotating LIDAR implemented using ray-casting that exposes all the information that the rays hit. Its behaviour is similar to the LIDAR sensor, but there are some differences between them. The semantic lidar module include more data per each point of the raw data, and it does not include neither intensity, drop-off nor noise model attributes [60]. Table 18 shows the instantiated variables for the semantic LIDAR module.

Table 18. Instance variables for the Semantic LIDAR module of CARLA simulator [61].

	Variable		Units	Type	Description
1	Channels		-	Int	Number of lasers shot
2	Horizontal_angle		Radians	Float	Horizontal angle of rotation at the measure.
3	Raw_data	x	meters	Float	Distance from origin to spot on X axis
4		y	meters	Float	Distance from origin to spot on Y axis
5		z	meters	Float	Distance from origin to spot on Z axis
6		Cos_inc_angle	-	Float	Cosine of the incident angle between the ray, and the normal of the hit object.
7		Object_idx	-	Unit	ID of the actor hit by the ray.
8		Object_tag	-	Unit	Semantic tag of the component hit by the ray.

The method required for the semantic LIDAR is the same required for the LIDAR sensor, which is called *save_to_disk*. The definition of this method can be found in the previous section. As with the LIDAR, it is decided to use only one unit for the modelling of the vehicle.

4.4.5. GNSS sensor

GNSS stands for Global Navigation Satellite System and provides geospatial positioning with global coverage in an autonomous manner. This module use triangulation to determine the position of a receiver in three dimensional space by calculating the distance between the vehicle and several satellites [48].

A single sensor implemented in the vehicle is sufficient to obtain valid and accurate information for both, simulation and real life. Weather conditions have minimal impact on these GNSS modules, this is because they operate at frequencies around 1.575 GHz. These operating frequencies are relatively insensitive to weather conditions. However, the windshield wipers can interfere with reception, making it impossible for a GNSS device to identify a complete string of navigation data from satellites, resulting in inaccurate data [62].

The GNSS sensor module used in the CARLA simulator has three instantiated variables, as shown in Table 19 [63].

Table 19. Instance variables for the GNSS module defined in the CARLA simulator

	Variable	Units	Type	Description
1	Altitude	Meters	Float	Height regarding ground level.
2	Latitude	Degrees	Float	North/South value of a point on the map
3	Longitude	Degrees	Float	West/East value of a point on the map

4.4.6. IMU sensor

IMU stands for Inertial Measurement Unit and consist of two sensors: accelerometer and gyroscope. The accelerometer measures a vehicle's three linear acceleration components whereas the gyroscope measures a vehicle's three rotational rate components. With the information from the GNSS sensor, which provides the initial location of the vehicle, the IMU can provide current information on current vehicle location and orientation [64].

Like the GNS sensor, a single sensor is sufficient to obtain the correct orientation of the vehicle. Although real vehicles may carry more than one to ensure data redundancy. This sensor is completely insensitive to external conditions, as it simply depends on the relative motion of the vehicle.

The IMU sensor module used in the CARLA simulator has 3 instantiated variables as shown in Table 20 [65]. A vector3D is a helper class defined in the CARLA simulator to perform 3D operations in a simple way, the instance variables defined are X, Y and Z, which represent the value of each axis of the vector [66].

Table 20. Instance variables for the IMU module defined in the CARLA simulator

	Variable	Units	Type	Description
1	Accelerometer	m/s^2	Vector3D	Linear acceleration
2	Compass	radians	Float	Orientation to North [0,0, -1,0, 0,0]
3	Gyroscope	rad/s	Vector3D	Angular velocity

4.5. Architecture model in AUTOSAR

The architecture designed in this thesis has been modelled in AUTOSAR, configured for a virtual integration and implemented in C code. Commercial products are available for this aim by specialized companies. The project has been developed using the tools provided by ETAS GmbH. The RTA-CAR solution has been used to generate an AUTOSAR Run-Time Environment (RTE) for the designed architecture.

RTA-CAR provides the integrated environment to configure and generate the classic AUTOSAR stack for ECUs. The program focuses on 4 basic concepts: (i) the configuration of the Application Layer, (ii) the configuration and generation of the Run-Time Environment (RTE), (iii) the configuration and generation of the Basic Software Layer (BSW) and (iv) the configuration and generation of the Operating System (OS). RTA-CAR has a specific tool for each task that RTA-CAR focusses on, as described in Table 21 [67].

Table 21. Description of the ETAS tools.

	ETAS tool	Description
1	ISOLAR-A	Assists user in designing application software to AUTOSAR standards.
2	ISOLAR-B	Provides the Basic Software Layer configuration tool
3	RTA-BSW	Provides BSW automatic configuration and code generation
4	RTA-RTE	Is the Run-Time Environment generator
5	RTA-OS	Is the Operating System configurator and generator

The designed architecture has been modelled using the ISOLAR-A tool. The methodology used to model the architecture step by step in the ISOLAR-A software and on the AUTOSAR standard are detailed below.

4.5.1. Vehicle configuration

First, one needs to model the elements of the architecture as software component types. This includes the different architecture components discussed in section 4.3 and the sensor modules discussed in section 4.4.

Then, a composition that configures the vehicle is defined and named in our project as the Virtual Vehicle. All the sensors specified in the vehicle model defined in section 4.4 are inserted in this composition as a component prototype. The component prototypes are instances of the component types created previously. When a component is multiply instantiated, such as a radar or camera, its instance name will be given a different label yet still relate to the same component type, as shown in Table 22.

Table 22. Definition of the repeated components prototypes.

Component prototype	Description	Component type
CPT_SnsCameraFront	Front camera	SnsCamera
CPT_SnsCameraRear	Rear camera	
CPT_RadarFL	Front left radar	snsRadar
CPT_RadarFR	Front right radar	
CPT_RadarML	Middle left radar	
CPT_RadarMR	Middle right radar	
CPT_RadarRL	Middle left radar	
CPT_RadarRR	Middle right radar	

Figure 16 displays a preliminary version of the composition.

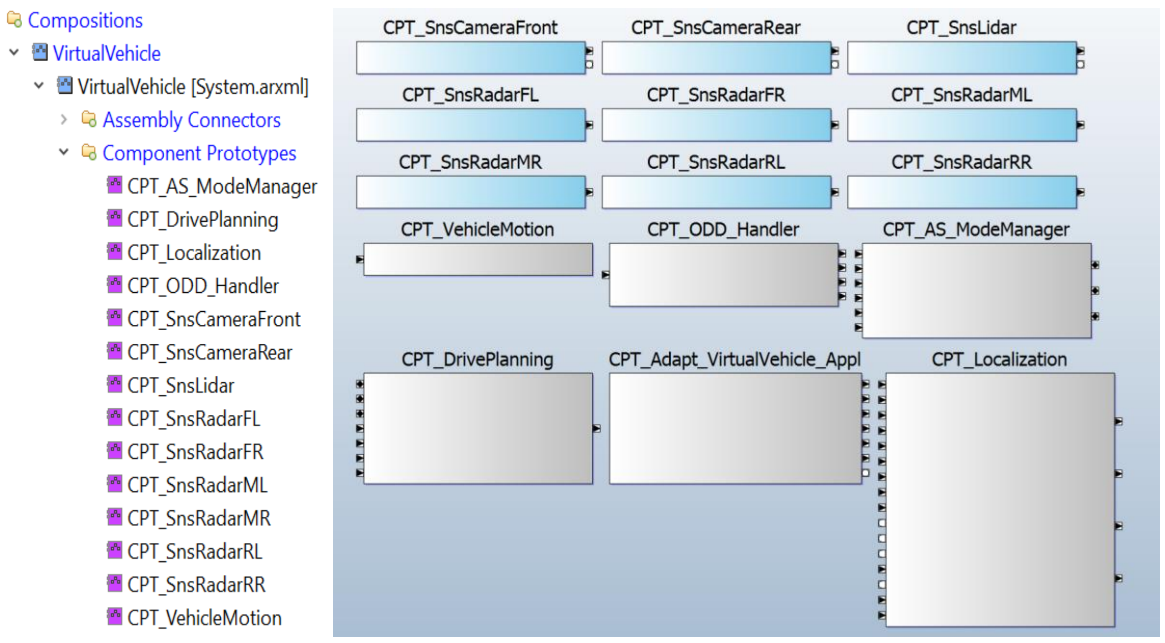


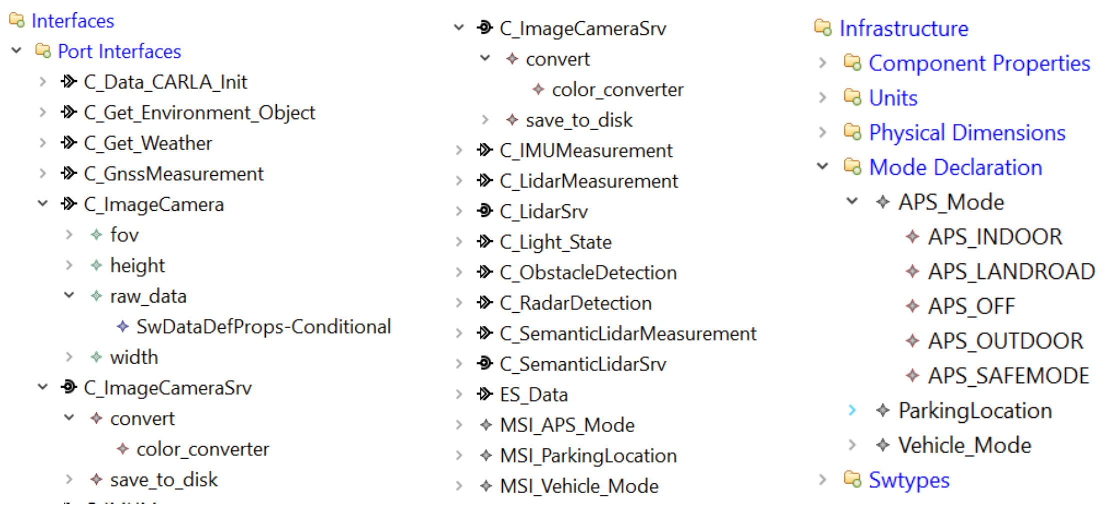
Figure 16. Preview of the virtual vehicle composition with the components defined.

Components communicate via ports, that are typed by the so-called interfaces. The sensor interfaces to be used have already been defined in section 4.4, which have been derived from the CARLA simulator APIs.

Radar, IMU, and other sensors that only transmit data through variables will be modelled with sender-receiver interfaces, whereas sensors like cameras and lidar that need to pass arguments to methods will need two interfaces: a client-server interface and a sender-receiver interface.

Figure 17 depicts an example of modelling camera interfaces. The variables are defined in a sender-receiver interface; the method calls are defined in a client-server interface with all the necessary details presented before. The units and a few other features of the variables are defined using the *SwDataDefProps*, which stands for Software Data Definition Properties.

It is necessary to specify the various modes for each group in the project infrastructure before defining the mode interfaces for each of the mode groups. The mode-switching interface type is required, which is different from the others, as shown in Figure 17.



```
Interfaces
  Port Interfaces
    C_Data_CARLA_Init
    C_Get_Environment_Object
    C_Get_Weather
    C_GnssMeasurement
    C_ImageCamera
      fov
      height
      raw_data
        SwDataDefProps-Conditional
      width
    C_ImageCameraSrv
      convert
        color_converter
      save_to_disk
    C_ImageCameraSrv
      convert
        color_converter
      save_to_disk
    C_IMUMeasurement
    C_LidarMeasurement
    C_LidarSrv
    C_Light_State
    C_ObstacleDetection
    C_RadarDetection
    C_SemanticLidarMeasurement
    C_SemanticLidarSrv
    ES_Data
    MSI_APS_Mode
    MSI_ParkingLocation
    MSI_Vehicle_Mode

Infrastructure
  Component Properties
  Units
  Physical Dimensions
  Mode Declaration
    APS_Mode
      APS_INDOOR
      APS_LANDROAD
      APS_OFF
      APS_OUTDOOR
      APS_SAFEMODE
    ParkingLocation
    Vehicle_Mode
  Swtypes
```

Figure 17. Components and mode interfaces for the architecture.

Component types have ports to communicate with components. Ports can be of two types: required and provided ports. Each provided port connects to a required port of the same interface type. Once the connections are made, the composition of the virtual vehicle is shown in Figure 18.

In plan to implement the communication of the Virtual Vehicle using the AUTOSAR Run-Time Environment, which can be automatically generated by RTA-RTE. The RTE generator requires a totally configured ECU instance and the mapping of component runnables to operating system tasks. ISOLAR-A provides a generation feature of the adapters needed to configure the system for RTE generation.

The generated adapters create a new component named *CPT_Adapt_VirtualVehicle_Appl*. This component includes all the non-used instantiated components in the model and generates the remaining connections required, as shown in Figure 18.

Now that the architecture has been modelled, the internal behaviour of the software component types can be defined, i.e. the runnables. The following sections provide an overview of the runnables that have been defined for each component.

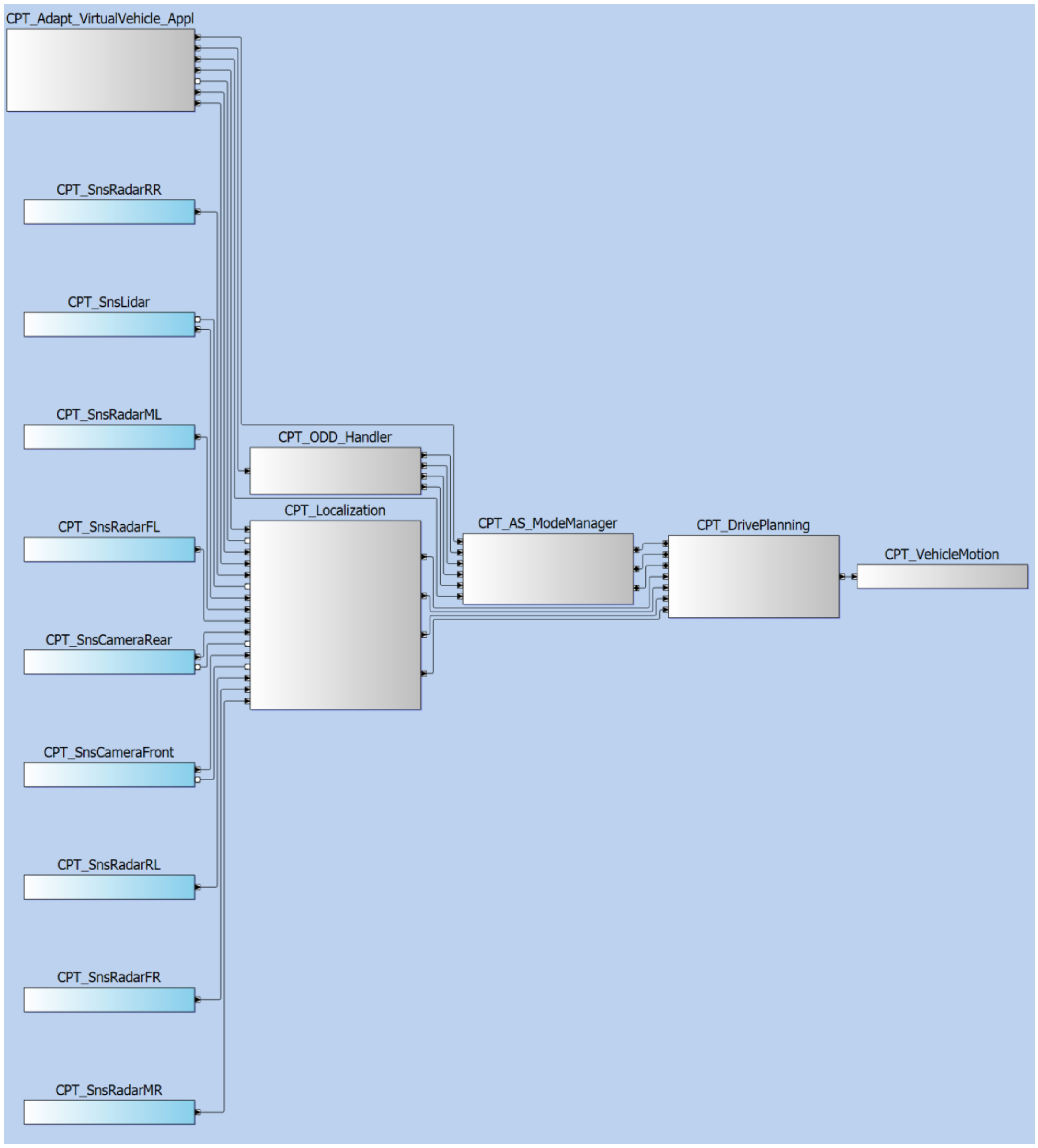
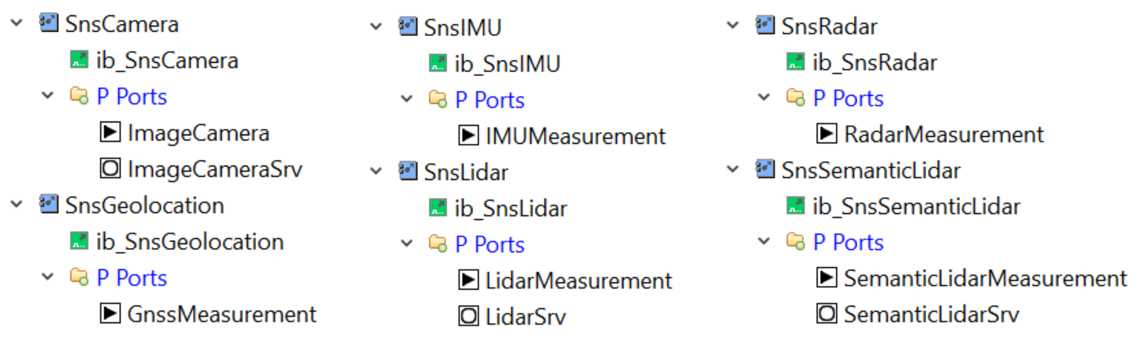


Figure 18. Architecture composition of the virtual vehicle with all the connections.

4.5.2. Perception

The Perception component itself does not exist as a modelled component. This component denotes all the specified sensor modules. Each sensor offers a port with a specific interface for the module. The sensor modules that also require a method must provide two ports, one operating as a sender receiver interface and the other as a server interface, as shown in Figure 19.



- SnsCamera
  - ib_SnsCamera
  - P Ports
    - ImageCamera
    - ImageCameraSrv
- SnsGeolocation
  - ib_SnsGeolocation
  - P Ports
    - GnssMeasurement
- SnsIMU
  - ib_SnsIMU
  - P Ports
    - IMUMeasurement
- SnsLidar
  - ib_SnsLidar
  - P Ports
    - LidarMeasurement
    - LidarSrv
- SnsRadar
  - ib_SnsRadar
  - P Ports
    - RadarMeasurement
- SnsSemanticLidar
  - ib_SnsSemanticLidar
  - P Ports
    - SemanticLidarMeasurement
    - SemanticLidarSrv

Figure 19. Provided ports for the sensor modules (Perception component).

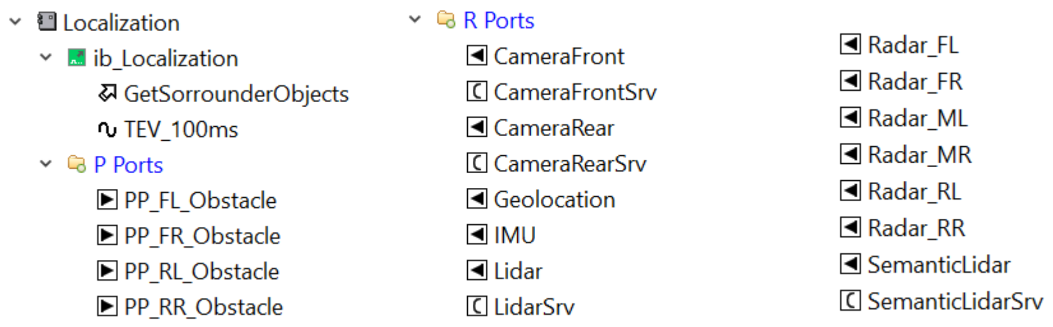
The data from the sensor modules generated in the CARLA simulator must flow through the provided ports.

4.5.3. Localization

All provided ports by the Perception components shall be connected to the Localization component. This means that each of the provided ports mentioned above will have their respective required ports defined. Figure 20 shows the final configuration of the Localization component.

A single runnable is what the Localization component needs to recognize the surrounding elements. This is labelled as a *GetSorrounderObjects* and specified as a 100-millisecond timing event (TEV 100 ms). The runnable shall be able to identify if there are any obstacles around the vehicle with the information provided by the sensor modules.

At each of the four edges of the vehicle, the runnable indicates whether there is an obstacle. As a result, four provided ports (two at the front and two at the back) shall be defined, one at each edge. Through the defined port interface, the value of the obstacle distance must be provided.



- Localization
  - ib_Localization
    - GetSorrounderObjects
    - TEV_100ms
  - P Ports
    - PP_FL_Obstacle
    - PP_FR_Obstacle
    - PP_RL_Obstacle
    - PP_RR_Obstacle
  - R Ports
    - CameraFront
    - CameraFrontSrv
    - CameraRear
    - CameraRearSrv
    - Geolocation
    - IMU
    - Lidar
    - LidarSrv
    - Radar_FL
    - Radar_FR
    - Radar_ML
    - Radar_MR
    - Radar_RL
    - Radar_RR
    - SemanticLidar
    - SemanticLidarSrv

Figure 20. Ports and runnable configuration for the Localization component.

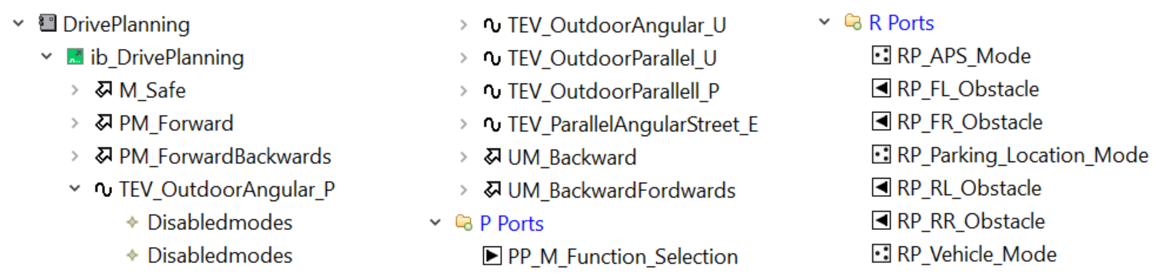
4.5.4. Drive Planning

The required ports for the component are the ones provided from the Localization component and the selected modes coming from the ADS Mode Manager. With this information, the Drive Planning component is responsible of choosing which of the five functions shall be performed.

For each manoeuvrer a runnable is required. For simplicity, we assume that our system will call the code in a 100 ms raster. The runnables will be triggered by a timing event of 100 ms.

Mode management is specified at runnable level, better said, for each timing event that starts the runnable. The modes that should be disabled must be selected according to the mapping defined in section 4.3.6.2. As a result, based on the modes selected, a single function will be activated.

A provided port is required to send the data of the selected manoeuvre function to the AS Motion component. The configured interface has five Boolean variables. Each one indicates when a manoeuvre function is activated. Figure 21 shows the final configuration of the Drive Planning component.



```
DrivePlanning
  ib_DrivePlanning
    M_Safe
    PM_Forward
    PM_ForwardBackwards
    TEV_OutdoorAngular_P
      Disabledmodes
      Disabledmodes
    TEV_OutdoorAngular_U
    TEV_OutdoorParallel_U
    TEV_OutdoorParallell_P
    TEV_ParallelAngularStreet_E
    UM_Backward
    UM_BackwardFordwards
  P Ports
    PP_M_Function_Selection
  R Ports
    RP_APS_Mode
    RP_FL_Obstacle
    RP_FR_Obstacle
    RP_Parking_Location_Mode
    RP_RL_Obstacle
    RP_RR_Obstacle
    RP_Vehicle_Mode
```

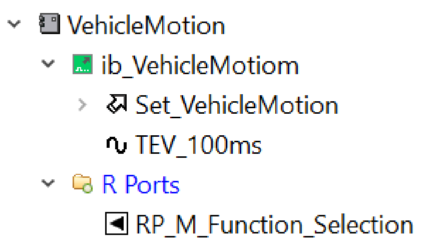
Figure 21. Ports and runnable configuration for the Drive Planning component.

4.5.5. AS MOTION

The AS Motion component implements the desired motion in the simulation environment. The Drive Planning provides the manoeuvre function that has to be applied.

To set the planned motion, one runnable (specified as a timed event of 100 ms) must be defined. This is the responsible to send the information to the CARLA simulator.

For this simplified model, no provided ports are needed because the CARLA simulator implements the vehicle motion. Figure 22 shows the configuration of the Drive Planning component.



```
VehicleMotion
    ib_VehicleMotiom
        Set_VehicleMotion
        TEV_100ms
    R Ports
        RP_M_Function_Selection
```

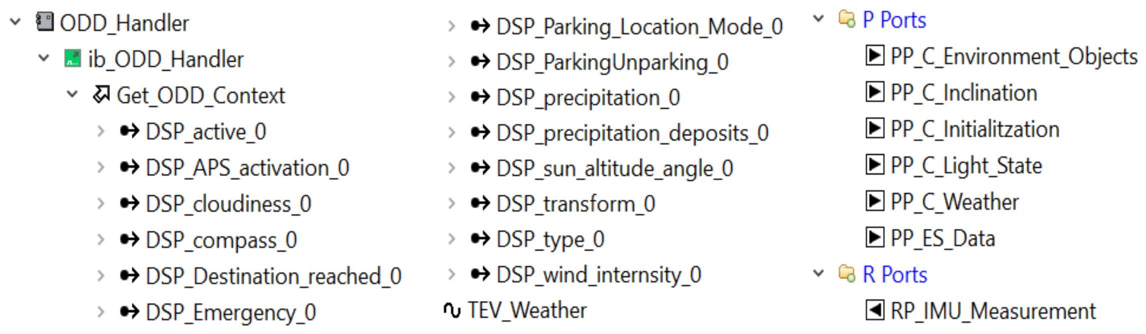
Figure 22. Ports and runnable configuration for the AS Motion component.

4.5.6. ODD Handling

The ODD Handling shall communicate with CARLA simulator to get all the information about the contexts. The IMU sensor module, which is used to determine the inclination of the vehicle, is the only provided port necessary for the component.

One runnable (called by a timing event of 100 ms) is specified to acquire all the ODDs defined in section 4.3.5. To keep the data type in the way that is generated in CARLA, the following four provided ports are defined: (i) Environment Objects, (ii) Light State, (iii) Weather, (iv) ES (External Cloud Service), (v) Initialization.

The ECS and the initialization include necessary variables needed for the correct simulation. Since the ECS is out of the scope and not programmed, a simulation of it is incorporated into the APS developed in the CARLA simulator, i.e., when the APS is to be activated, the values that would flow from the External Cloud Service are generated in the simulated environment [6]. The ODD Handling component configuration is depicted in Figure 23.



```
ODD_Handler
    ib_ODD_Handler
        Get_ODD_Context
            DSP_active_0
            DSP_APS_activation_0
            DSP_cloudiness_0
            DSP_compass_0
            DSP_Destination_reached_0
            DSP_Emergency_0
            DSP_Parking_Location_Mode_0
            DSP_ParkingUnparking_0
            DSP_precipitation_0
            DSP_precipitation_deposits_0
            DSP_sun_altitude_angle_0
            DSP_transform_0
            DSP_type_0
            DSP_wind_internsity_0
        TEV_Weather
    P Ports
        PP_C_Environment_Objects
        PP_C_Inclination
        PP_C_Initialitzation
        PP_C_Light_State
        PP_C_Weather
        PP_ES_Data
    R Ports
        RP_IMU_Measurement
```

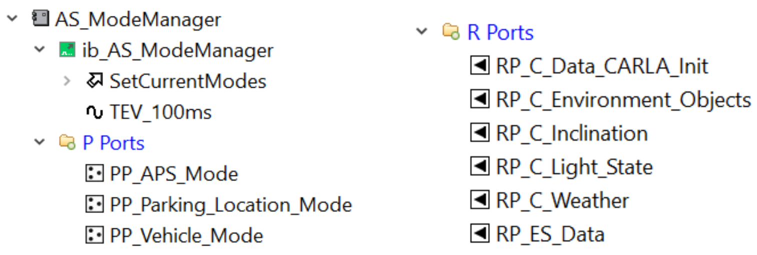
Figure 23. Ports and runnable configuration for the ODD Handling component.

4.5.7. ADS Mode Manager

All the ports provided by the ODD Handling, which identify the system context, are necessary for the Mode Manager. In addition, the data provided by the ECS is also required. This data has been simulated in the simulation environment.

One runnable (called by a timing event of 100 ms) is needed to implement the state machine described in section 4.3.6. The Mode Manager is responsible of selecting the right modes of each group based on the information obtained.

Three provided ports are defined to send the selected modes to the other components, each port for each mode group. Figure 24 shows the final configuration of the Drive Planning component. The runnable *SetCurrentModes* is hand-coded and implements the state machine.



```
AS_ModeManager                          R Ports
  ib_AS_ModeManager                       RP_C_Data_CARLA_Init
    SetCurrentModes                       RP_C_Environment_Objects
    TEV_100ms                             RP_C_Inclination
  P Ports                                 RP_C_Light_State
    PP_APS_Mode                           RP_C_Weather
    PP_Parking_Location_Mode              RP_ES_Data
    PP_Vehicle_Mode
```

Figure 24. Ports and runnable configuration for the ADS Mode Manager.

5. Results

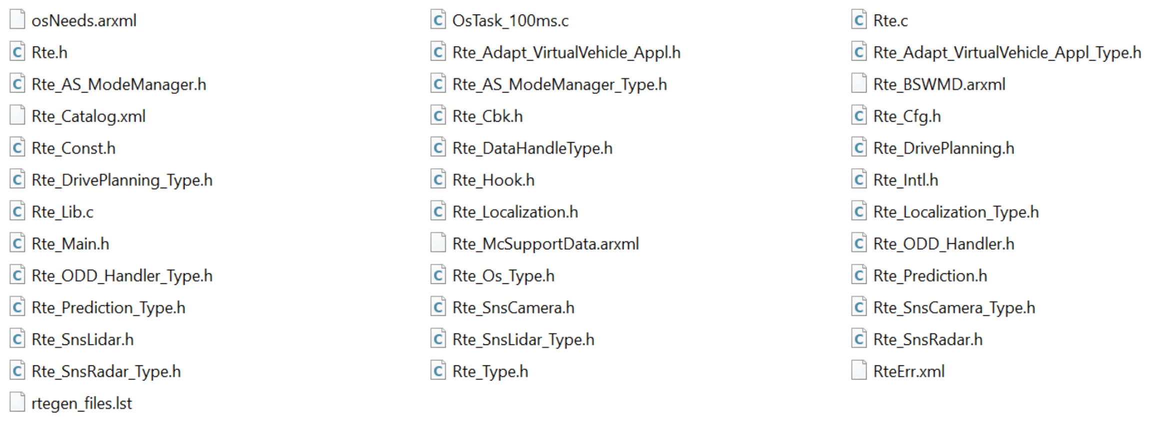
This section aims to present the results obtained of the modelled architecture and its smooth integration with the Automated Parking System. The RTE, which implements the communication between the application layer and the basic software services, must be generated to run the modelled architecture on hardware.

The concept is validated in a virtual environment, i.e. the code runs on a personal computer. The generated AUTOSAR Run-Time Environment is integrated with the simulation environment and the autonomous vehicle implemented by the CARLA simulator.

5.1. Run-Time Environment

ISOLAR-A is the AUTOSAR authoring tool used to model the software architecture and integrates several tools like RTA-RTE for the generation of the AUTOSAR Run-Time Environment (RTE). The specifications and requirements defined for the Run-Time Environment followed in the development for the project are defined in [33] and [32].

The RTE generator implements the standardized APIs for setting/getting data and allocates the memory for the data layer. The RTE generated files are shown in *Figure 25*.



osNeeds.arxml
OsTask_100ms.c
Rte.c
Rte.h
Rte_Adapt_VirtualVehicle_Appl.h
Rte_Adapt_VirtualVehicle_Appl_Type.h
Rte_AS_ModeManager.h
Rte_AS_ModeManager_Type.h
Rte_BSWMD.arxml
Rte_Catalog.xml
Rte_Cbk.h
Rte_Cfg.h
Rte_Const.h
Rte_DataHandleType.h
Rte_DrivePlanning.h
Rte_DrivePlanning_Type.h
Rte_Hook.h
Rte_Intl.h
Rte_Lib.c
Rte_Localization.h
Rte_Localization_Type.h
Rte_Main.h
Rte_McSupportData.arxml
Rte_ODD_Handler.h
Rte_ODD_Handler_Type.h
Rte_Os_Type.h
Rte_Prediction.h
Rte_Prediction_Type.h
Rte_SnsCamera.h
Rte_SnsCamera_Type.h
Rte_SnsLidar.h
Rte_SnsLidar_Type.h
Rte_SnsRadar.h
Rte_SnsRadar_Type.h
Rte_Type.h
RteErr.xml
rtegen_files.lst

Figure 25. RTE files generated for the modelled architecture.

These files come in the following formats: (i) ARXML files, (ii) C Header Source files, (iii) LST files, (iv) XML files and (v) C Source files. The C source files are the ones that are most relevant because they implement the communication and the sequence how the runnables are to be executed.

The *OsTask_100ms.c* file is highly interesting since it specifies the order of execution of the runnables as specified in the RTE event to task mapping. The file defines a 100-millisecond repetition task. The runnables must be ordered logically such that all necessary information is available before the next execution step. The sequence is defined during the deployment and RTE configuration in ISOLAR-A.

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 functions in accordance with the mapping described in section 4.3.6.2. Finally, it performs the motion specified by the selected function. *OsTask_100ms.c* code is shown below.

```
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 can be used to write or read the internal variables defined in the architecture. The RTE generates the APIs only for the variables defined in each runnable as a *DataReceivePoints* and *DataSendPoints*. Below is shown the APIs of the precipitation variable: one API writes the data in the ODD Handling component and the other API reads the value for the Mode Manager.

```
Rte_Write_ODD_Handler_PP_C_Weather_precipitation(VAR(float32, AUTOMATIC) data);
Rte_DRead_AS_ModeManager_RP_C_Weather_precipitation(void);
```

The mode declaration groups are different from the conventional variables because they have been specified as a mode switch interface. These can be defined for reading the actual mode (*ModeAccesPoints*) and for switching the mode groups (*ModeSwitchPoints*). The mode declaration groups could be defined one way or both, depending on what is required. The following code shows the APIs of the APS mode group.

```
Rte_Switch_AS_ModeManager_PP_APS_Mode_Mode(VAR(uint8, AUTOMATIC) data);
Rte_Mode_AS_ModeManager_PP_APS_Mode_Mode(void);
```

All the files generated automatically by the RTE generator shall not be edited. In addition to these files, the ISOLAR-A provides the option to create the component Code-Frame of each component that has runnables defined, as shown in Figure 26. These files contain the runnables that shall be executed according to the *Os_Task_100ms.c*. These files are to be edited to adapt the designed architecture to the implementation. *AS_ModeManager*, *DrivePlanning*, *ODD Handling*, and *VehicleMotion* are the files that have been edited in this master thesis.

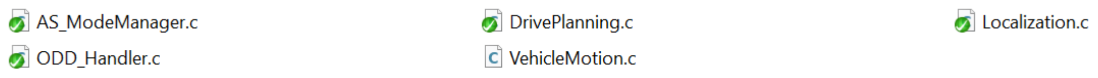

AS_ModeManager.c
ODD_Handler.c
DrivePlanning.c
VehicleMotion.c
Localization.c

Figure 26. Generated Code-Frame components.

The implementation of the runnables has been kept very simple. The interaction with the CARLA provided functionality is described below.

The code fragment added to the *Get_ODD_Context* runnable (from ODD Handling Code-Frame) separates all the data coming from the CARLA simulator and writes it to each related variable. The write APIs generated by *Rte.c* were used. The information would have to be extracted from the sensors, but to simplify the implementation, it is obtained directly from the CARLA simulator. This information is separated by commas.

The extra piece of code added to the *SetCurrentModes* runnable (from AS Mode Manager Code-Frame) is the reading of all necessary variables and the implementation of the state machine defined in section 4.3.6.4.

For the Drive Planning, the inserted code fragment in each of the five runnables consist of setting the Boolean variables related to the corresponding manoeuvre function to true. To make sure that only one function is activated, the variables related to the other manoeuvres are set to false.

Finally, the piece of code added to the *Set_VehicleMotion* runnable (from Vehicle Motion Code-Frame) reads the data from the Drive Planning to know the manoeuvre to be active and sends this information to the CARLA simulator.

To carry out the specified tasks in the RTE, a new file with an infinite loop must be created in addition to all the files the ISOLAR program has already generated. Only the *Os_Task_100ms* has been defined. This file aim is to call in each loop the specified task to be executed. This file is named *Server.c* (see section 5.2).

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 programming language while the simulation environment uses Python language, the communication protocol must be compatible with 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. The IP utilized is the Localhost, which is 127.0.0.1, because the RTE and the CARLA simulator operate on the same computer [68]. The ports 4455 has been used, since this port is one of the used for the Transmission Control Protocol [69].

TCP sockets are typically used by two types of applications: servers and clients. A TCP server accepts connections from TCP clients by listening on a well-known port (or IP address and port pairs). To establish a connection with a TCP server, a TCP client must send a connection request to the server [70], [71]. For the APS implementation, the RTE is referred as a server and the simulation environment as a client, as shown in Figure 27.

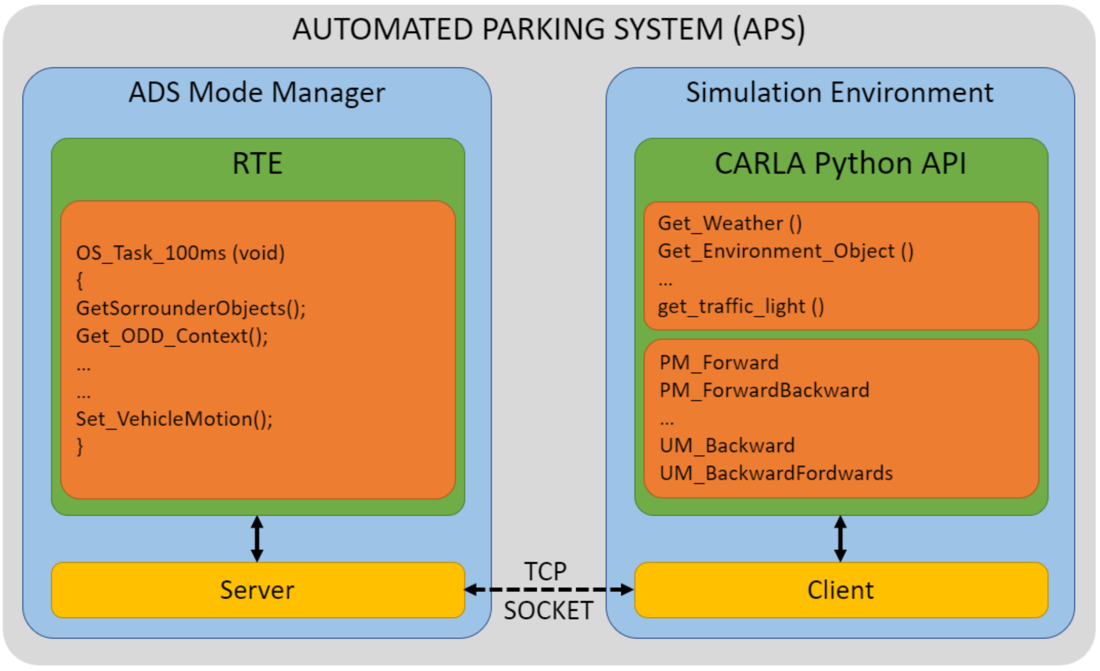


Figure 27. TCP socket communication protocol.

The extra file created to execute the RTE tasks is considered the server of the APS, hence the name of the file (*Server.c*). Before the infinite loop, the file starts listening on the specified port to see if any clients are requesting to establish a connection. Once the client has sent the request to establish the connection, the server accepts it and then, the infinite loop is executed to call the *Os_Task_100ms*.

The RTE task specifies the data flow direction. The server obtains the context information before providing the manoeuvre function to the simulator. As a result, the client provides the data first, and the server replies.

As stated above, client-side coding (5 manoeuvring functions) is out of the scope of this master thesis. The programming was done in another thesis in parallel with this one [6].

The reception of the data sent by the client has been coded in the ODD Handling Code-Frame, inside the runnable *Get_ODD_Context*. The *Set_VehicleMotion* runnable (Vehicle Motion Code-Frame) is coded to send the information about the selected manoeuvre to the client. The server sends a number from 0 to 4. Each number is related to a function manoeuvre as indicated in Table 23.

Table 23. Function manoeuvre codification for the TCP.

Value	Function manoeuvre
0	PM_FORDWARD
1	PM_FORWARDSBACKWARD
2	UM_BACKWARD
3	UM_BACKWARDFORDWARD
4	M_SAFE

5.3. Developed prototype (MVP)

A prototype is derived once the architecture is fully implemented with the APS in the simulation environment. By including enough features to attract early clients and allow the product idea to be evaluated, the entire system is a Minimum Viable Product (MVP).

The overall concept of the MVP has been represented in a sequential diagram to simplify the understanding of the data flow. The sequential diagram depicted in Figure 28 represents not only the entire data flow of the architecture itself, but also how it relates to the CARLA simulator through the TCP socket. Everything discussed in section 4 can be understood simply and graphically with the presented sequence diagram in Figure 28.

The components highlighted at the top of Figure 28 refer to the simulator (CARLA vehicle) and the architecture components defined in section 4.3 (Localization, ODD Handling, ADS Mode Manager, Drive Planning & Vehicle Motion).

The diagram shows in Figure 28 the interface names of the data transmitted. Subsequently, different scenarios are indicated with the detailed information that is actually transmitted. The dashed lines in the diagrams represent data being transmitted over the TCP socket connection.

A technical problem faced is the tick rate of both sides, as the communication between the RTE and the simulation environment has to be synchronous. The RTE operates with a tick rate of 100 milliseconds, whilst the simulation environment requires a tick rate of 5 milliseconds. It is not possible to violate the frequency anywhere, because undesired states start to appear. To resolve the inconsistency, TCP communication with the architecture is only called every 20 times in the infinite loop of the simulation environment.

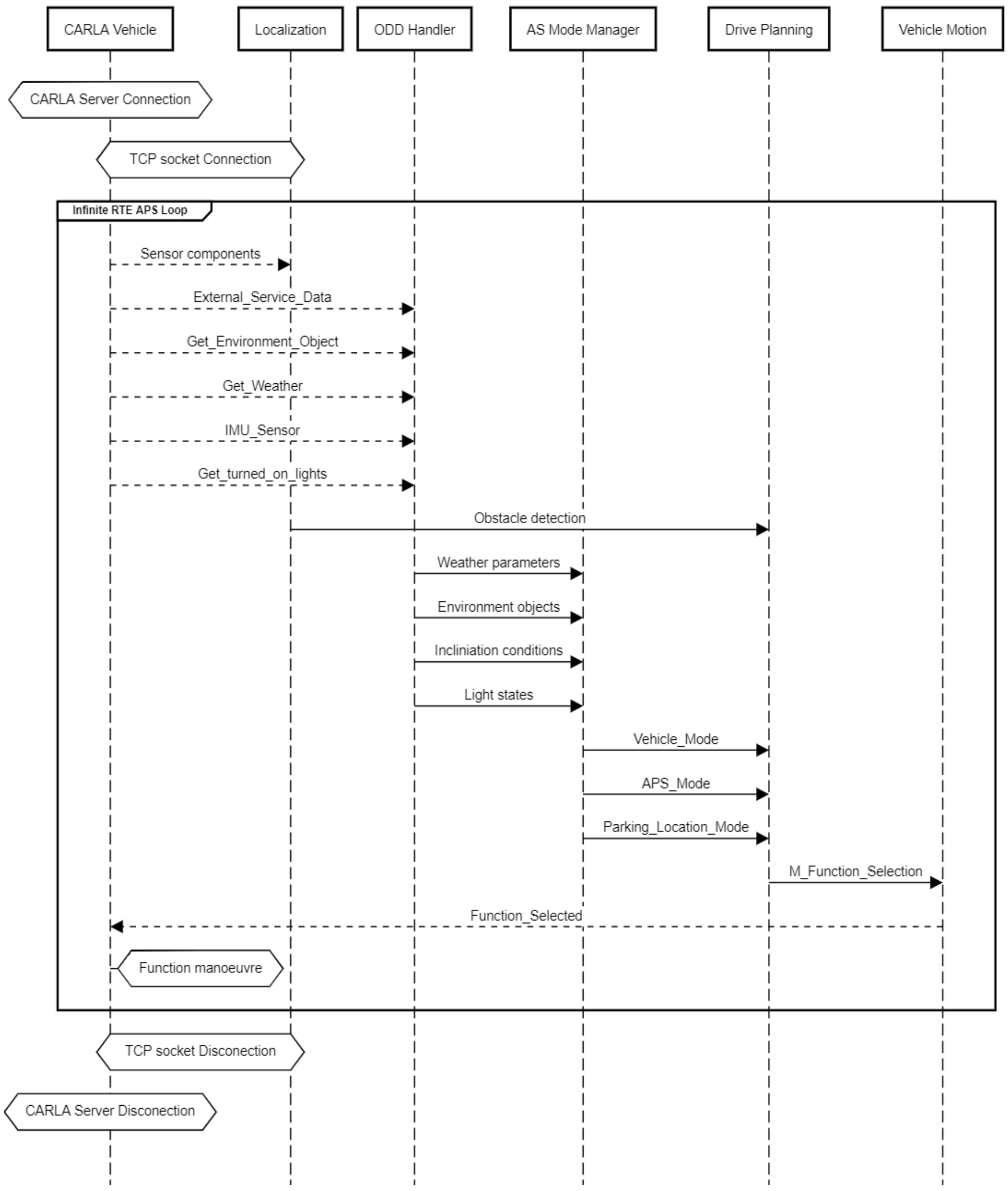


Figure 28. Generalist sequence diagram of the APS integrated to the architecture.

Below, some cases are presented in a sequence diagram to demonstrate the real data flow that the product should have and how it should behave in various scenarios, with the aim of validating the MVP.

Figure 29 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.

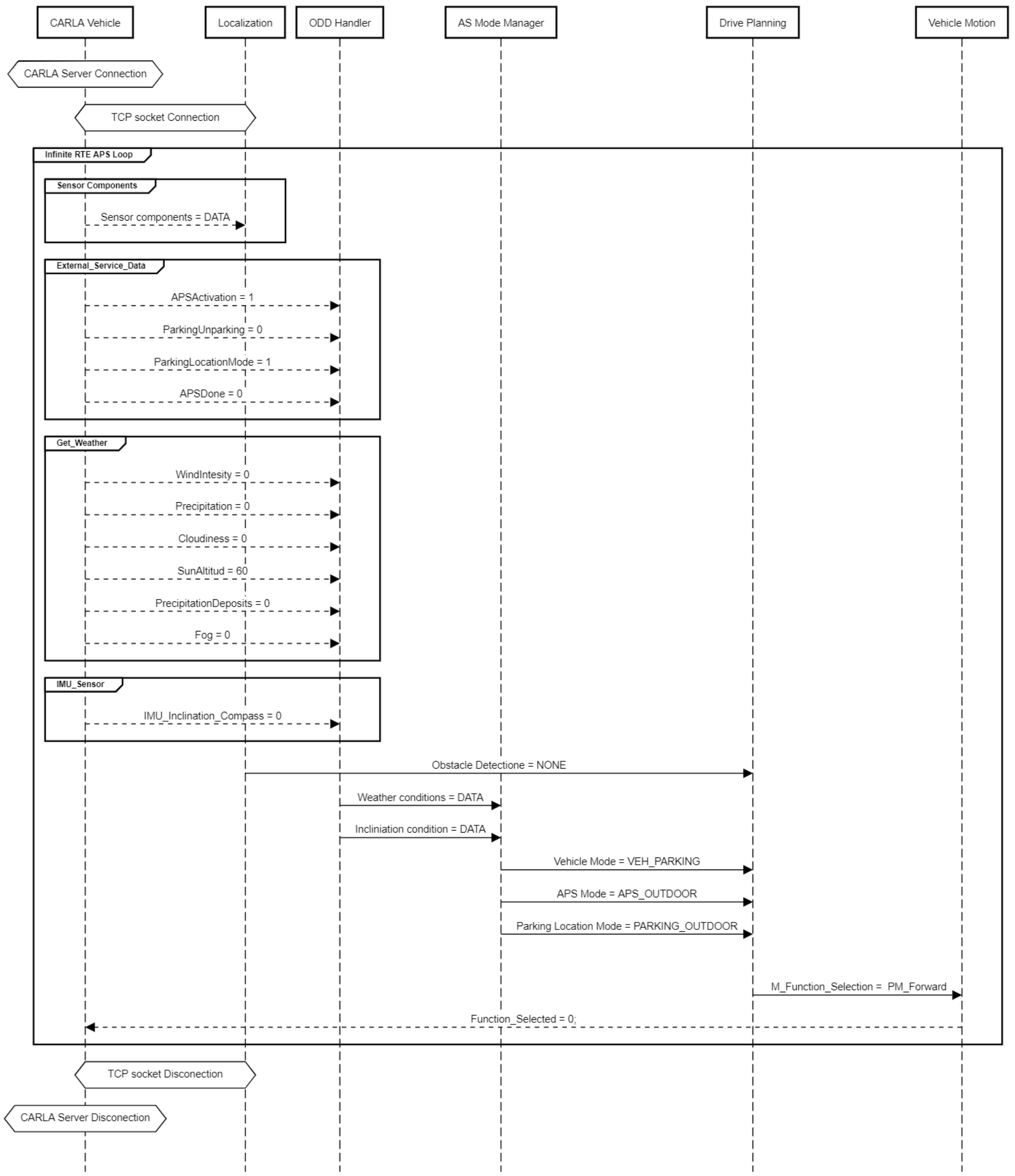


Figure 29. Detailed sequence diagram for the activation of the PM_Forward manoeuvre.

The MVP must react to a contex change and verify that the specified modes are still operable under the new ODDs, with the aim of ensuring safety. Considering the above case, during the chosen parking manoeuvre, a storm with wind, rain and water on the ground breaks out. Consequently, the system switches to *APS_SafeMode* and executes the safe mode function, as the previous APS mode selected was not operable under the new conditions.

Furthermore, an obstacle has also been sensed in the front position of the vehicle (F_L and F_R), which is another reason to degrade the system to the safe mode. The Obstacle Detection of the Localization component informs the location of the obstacles detected to the Drive Planning through the following variables: F_L (Front Left), F_R, M_R (Middle Right), M_L, R_R (Rear Right) and R_L. Figure 30 illustrates the data flow transmitted in this specified case.

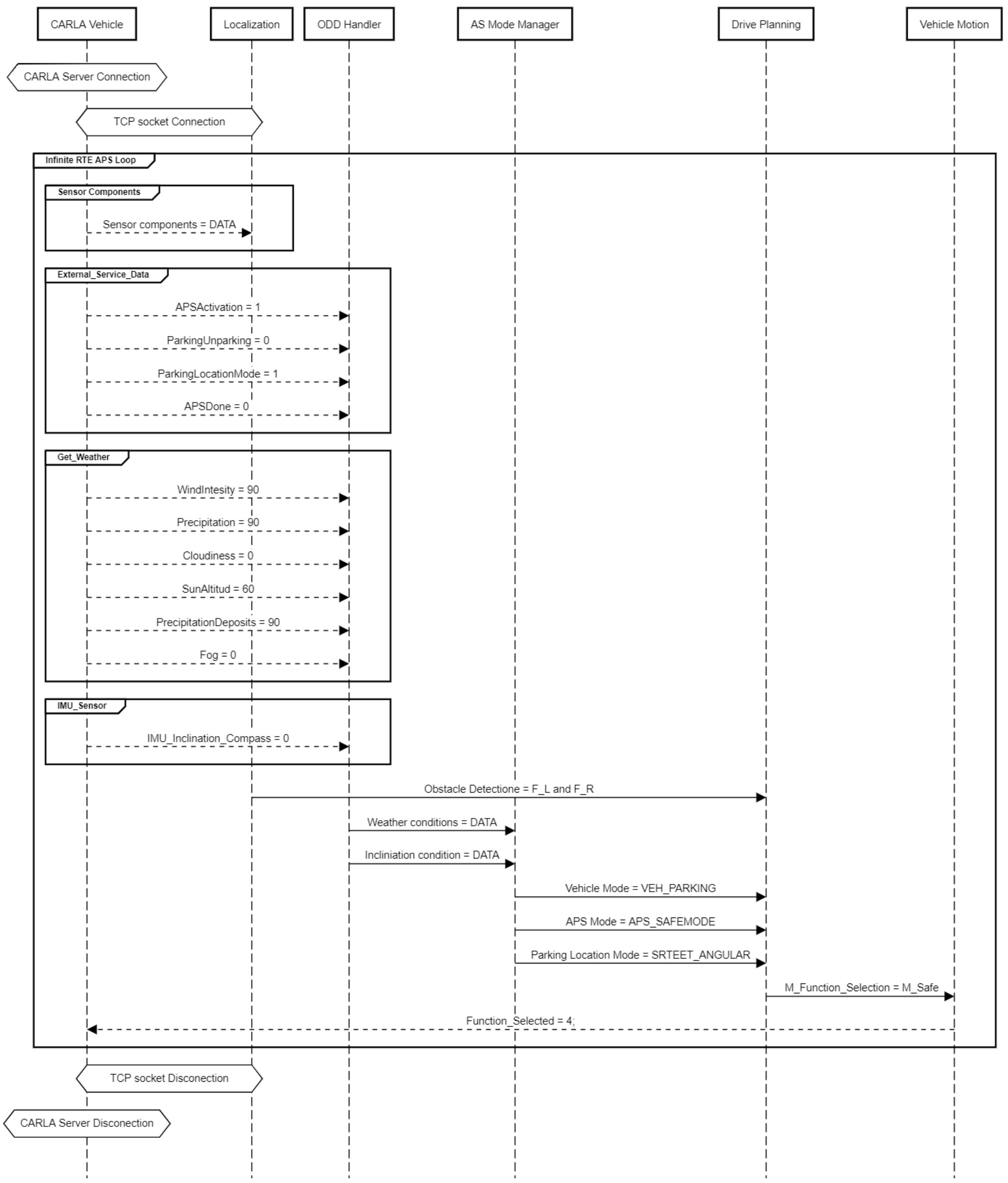


Figure 30. Detailed sequence diagram for the activation of the safe mode.

The system has been simplified to ease the understanding of the data flow shown in the sequential diagrams above. Some details, such as surrounding objects, have not been considered.

The MVP shall be tested with such presented examples, and it must ensure safety in any context. Whenever this is not possible, as defined in section 4.3.6.3 , the system should switch to a safe mode as shown in Figure 30.

6. Conclusions

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 have been evaluated. The AUTOSAR standard, widely established reference software architecture for the automotive industry domain, has been extensively analysed and applied for the development of this work.

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 is a software architecture instantiated in the Automated Parking System (APS) which 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. However, the architecture is generic and enables the gradual development of autonomous vehicle functions. By adapting the system requirements, additional autonomous functions can be added.

The main contribution of this project has been to explore the applicability of the AUTOSAR Mode Management methodology for the design of the Autonomous System (AS) Mode Manager component. For this purpose, we have narrowed the system to the minimal number of components required to describe how the parking manoeuvre adapts to the context by switching between configured modes in order to ensure the safety of the system.

Nevertheless, the developed system is scalable due to having a well-defined logical architecture. Thus, adding new functionalities to the system is easier and requires minimum code development.

The results presented in this thesis showed that implementing a clearly defined architecture reduces considerably the complexity of programming autonomous systems while meeting strict safety requirements coupled with state-of-art simulation environment.

6.1. Future works

The objective of developing an AUTOSAR safety-based architecture for a basic autonomous driving systems has been accomplished. The current work is a first step in providing the basis for a minimal autonomous system architecture. However, there is room for improvement and possible extensions.

In the automotive industry the AUTOSAR Classic Platform and the Adaptive Platforms will coexist. It is not clear whether other service-oriented architectures like the Robot Operating System (ROS2) will be adopted. Robotic architecture seems to be often ignored by automotive software engineers, even though they have a huge potential. A future work would be to evaluate robotics standardized architectures with respect our AUTOSAR architecture in order to identify the features that would improve the results of our work.

The AUTOSAR methodology relies strongly on configuration and code generation. In this work, the state machine implementing the ADS Mode Manager is hand coded. We believe that the AUTOSAR Basic Software Mode Manager offers the capability to configure the ADS Mode Manager. If this was the case, the implementation of the state machine could be automatically generated and integrated as part of the ECU configuration.

In this work, the architecture of the Virtual Vehicle was reversed engineered from the CARLA simulated vehicle consisting of a number of sensor modules. A future work would be to generate the Python code of the CARLA vehicle from the architecture of the Virtual Vehicle modelled as an AUTOSAR top level composition. It should be explored which meta-data is required, for instance, to describe the location of the sensors in the physical simulated vehicle.

Acknowledgement

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. I would like to thank her for her patience, her words of encouragement and her enthusiasm for every achievement along the way. I am glad to know that there is still a long way to explore together to make autonomous driving 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. Juan Manuel Moreno Eguílaz for his advice, for motivating me to take advantage of this opportunity to do my master thesis and for showing me the great advantages of doing it.

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".

I don’t want to forget ETAS GmbH for the "Gratuitous Loan for Use" of the AUTOSAR Tools ISOLAR-AB for modelling the architecture of the autonomous system and RTA-RTE for implementing the communication for the virtual vehicle.

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 to Roger Tremosa, who has worked closely with me in the development of this project and its implementation.

A. Environmental and Social Impact

Nearly everyone depends on the automotive industry, one of the largest in the world, in their daily life. There are numerous public and private transportation options, although many people routinely rely on their personal vehicles.

The objective of the APS function is to benefit users and, at the same time, promote environmental sustainability. The principal advantages are [72]:

I. Driving safely. I. Driving safely. Nearly 94 percent of fatal crashes, according to the US Department of Transportation, are indeed the result of human error. Therefore, big manufacturers are pushing the use of self-driving vehicles [73].

II. Boost customer satisfaction. It is interesting to note that parking will likely continue to be difficult for a very long time due to the predicted 293.6 million motor cars that will be on American roads in 2021. Drivers will not have to waste time searching for parking thanks to automation. Adopting simple user interfaces, such as mobile apps, can also make the user experience comfortable and memorable.

III. Reduce environmental impact. Automated parking systems minimize emissions because vehicles do not waste time searching for a parking space and instead drive directly to their designated location.

IV. Save space and money. Users can park at narrower parking spaces with an Automated Parking System, as they do not have to stay in the car. This advantage allows smaller parking spaces.

B. Budget

This section shows the cost related to the design of the architecture and its implementation.

B.I Equipment

The costs of machinery and digital tools for in the project development appear in Table 24.

Table 24. Equipment costs.

Concept	Unit cost (€)	Quantity	Total (€)
Personal computer	800.00	0.20	160.00
ISOLAR-A License for 5 months	10500.00	0.20	2100.00
Working station (GPU)	3000.00	0.20	600.00
Microsoft Office	30.00	0.20	6.00
Total			2866.00

B.II Human resources

Table 25 lists the number of working hours dedicated to the thesis and related tasks. The expense associated with the supervisory procedure is also included.

Table 25. Human resources costs.

Concept	Unit cost (€/h)	Quantity	Total (€)
Research	20.00	350	7000.00
Architecture modelling	20.00	175	3500.00
Code developing	20.00	120	2400.00
Implementation	20.00	160	3200.00
Writing	20.00	150	3000.00
Supervision	120.00	100	12000.00
Total			31100.00

B.III Total budget

Table 26 presents the total budget that results equipment and human resources.

Table 26. Total budget of the thesis.

Concept	Total (€)
Equipment	2866.00
Human resources	31100.00
Total	31100.00

C. Time planning

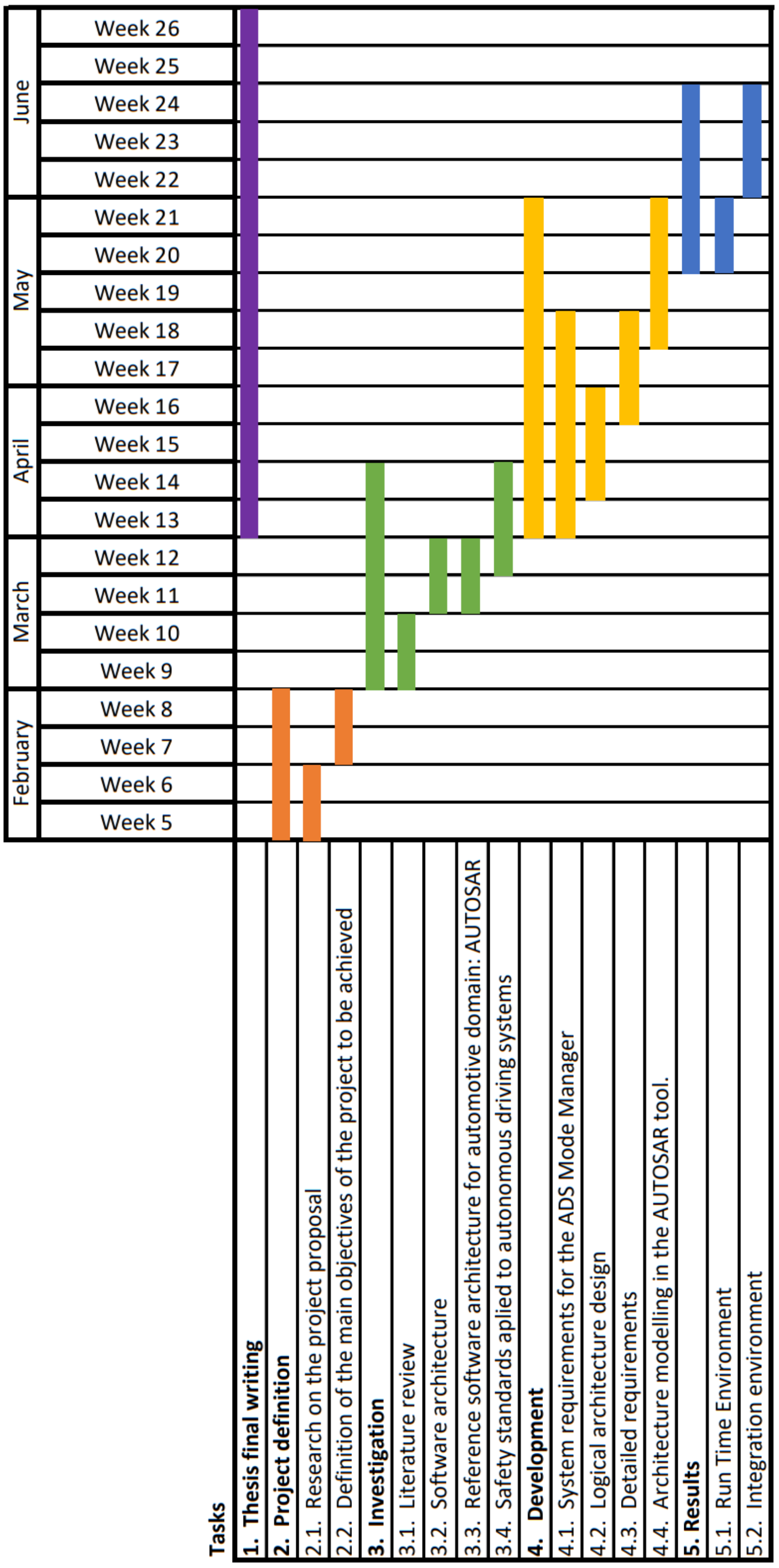


Figure 31. Gantt diagram presenting the project evolution.

D. CARLA simulator

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 32. 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.

D.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.

The CARLA simulator consists of a scalable client-server architecture (illustrated in Figure 33) 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[74].

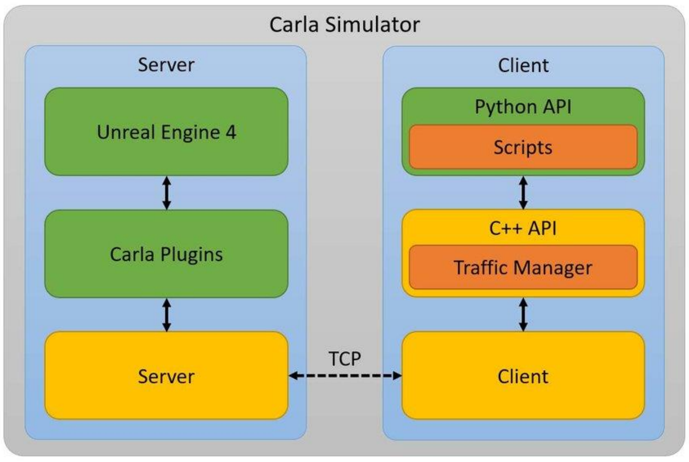


Figure 33. CARLA Simulator System Architecture Pipeline [35].

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. Further information can be found in section 3.44.1.5.

- 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 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 34. Cloudy road junction on map 5 of CARLA simulator.

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 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 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: (i) 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 and (ii) Detectors: Retrieve data when the object they are attached to registers a specific event. The types of detectors: collision, lane invasion and obstacle.

Other: Different functionalities. Other types: GNSS, IMU, LIDAR, radar RSS and semantic LIDAR.

D.II Weather Operation Design Domain

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 35 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.

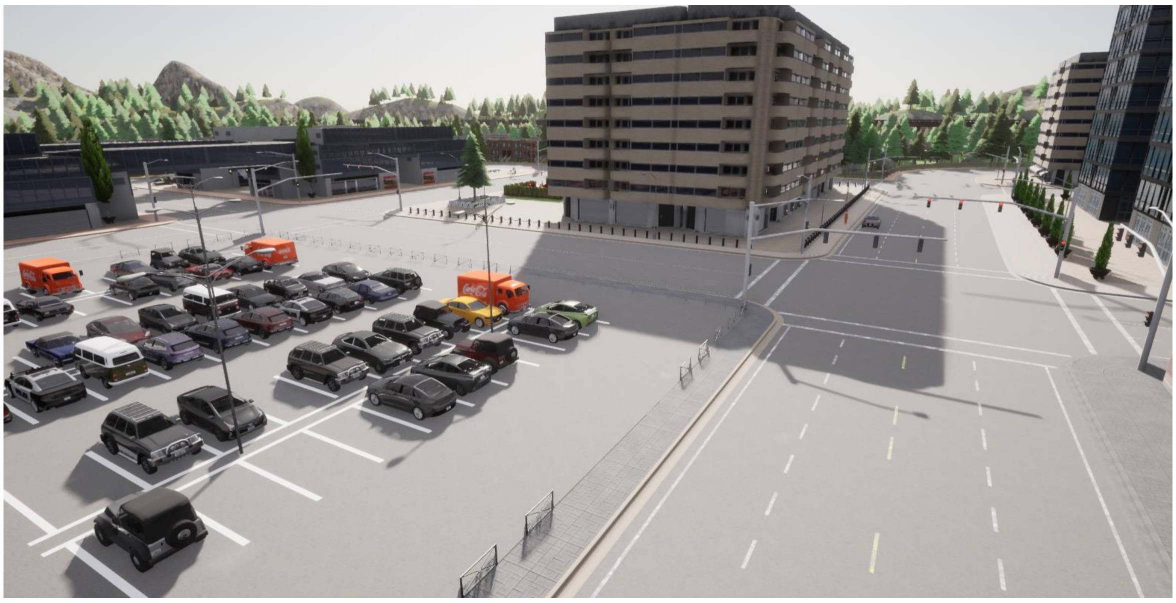


Figure 35. Sunny day.

Figure 36 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 capacity to determine ambient light level is essential.

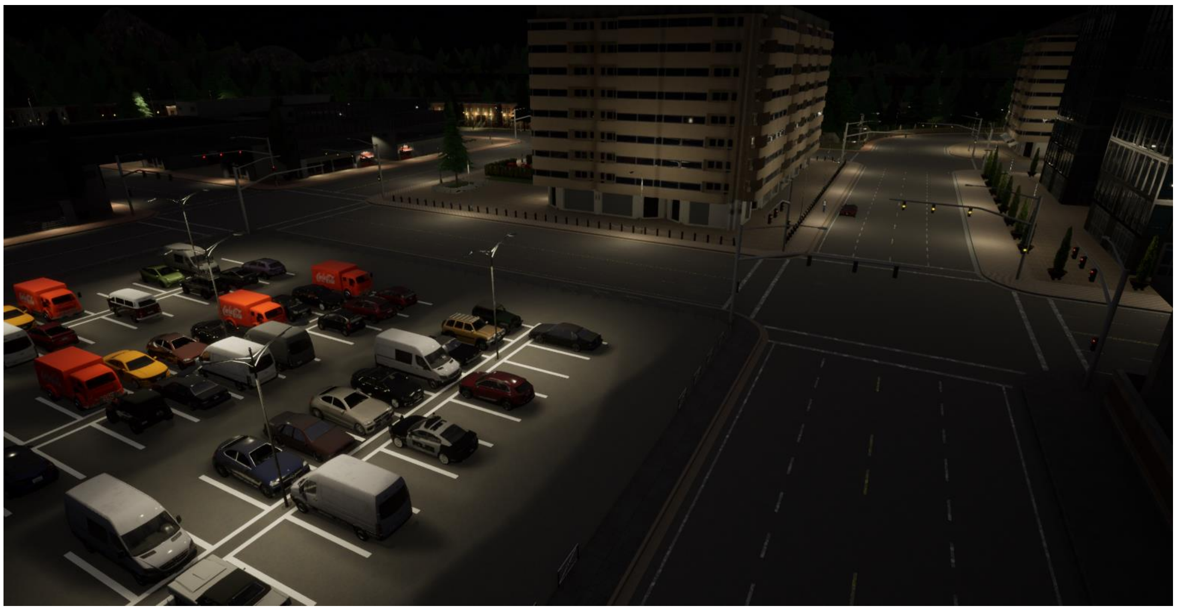


Figure 36. low ambient lighting.

Figure 37 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.

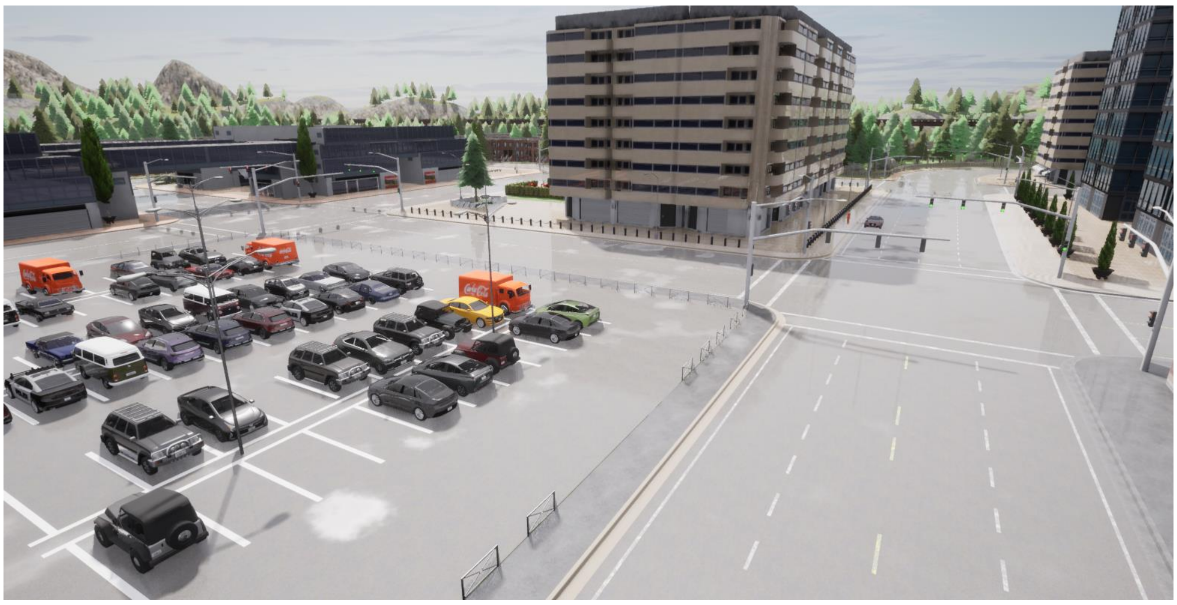


Figure 37. Sunny day with water deposits.

Figure 38 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.

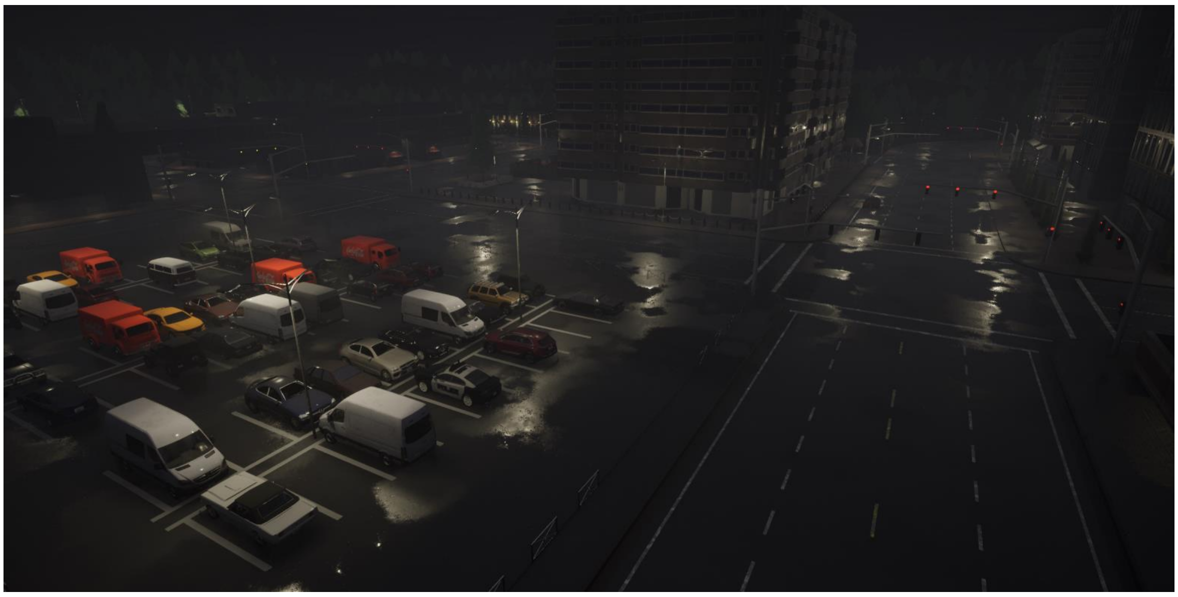


Figure 38. Night ambient with water deposits.

Finally, Figure 39 shows a foggy day. For this ODD, the APS should only work for indoor parking lots, just like the previous two contexts.

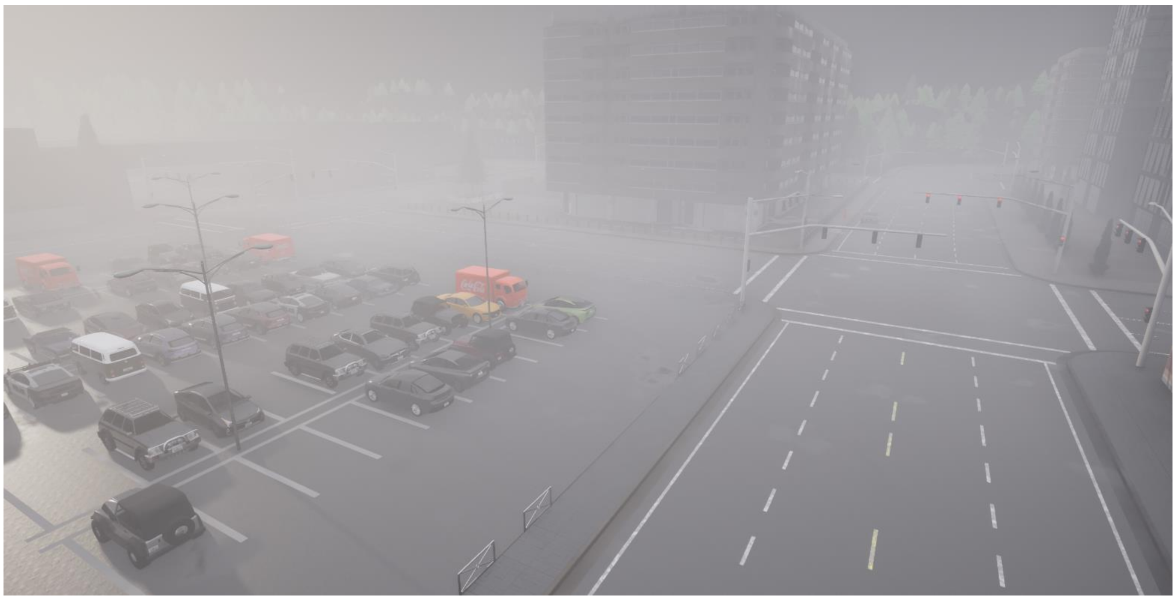


Figure 39. Foggy day.

E. Automated Parking System Definition

The definition of the Automated Parking System has been done together with other master thesis [6]. A systems engineering approach is used to define the APS. Table 27 presents the uses cases of the APS.

Table 27. Uses cases for the APS.

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: o Parallel parking o Angular parking o Perpendicular parking	**SYS-2, SYS-4, SYS-15**
UC-3	APS supports the following drivable area types: o Indoor parking o Outdoor parking o Urban road o 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: o The status of the function: active or inactive. o The menus to select the parking spot. o 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**

For simplicity and satisfaction on safety demands, this works assumes the constrains presented in Table 28.

Table 28. Constrains for the APS.

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 (i.e., a land car park).	**UC-3**
CON-13	The APS must respect the traffic rules (i.e., speed limit).	

The requirements showed in Table 29 shall be fulfilled by the system under design, which, as is known, is composed of the autonomous vehicle and the mobile application.

Table 29. System requirements for the APS

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)

Bibliography

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

[2] SAS, ‘MSCA Safer Autonomous Systems (SAS) - Ensuring trustworthiness of autonomous systems’. https://be.linkedin.com/in/msca-safer-autonomous-systems-sas/de (accessed Jun. 07, 2022).

[3] Nationallizenz Ebooks Medicine, ‘AUTOSAR — The Worldwide Automotive Standard for E/E Systems’, 2013, [Online]. Available: https://link.springer.com/article/10.1007/s40111-013-0003-5

[4] M. Wood, C. Knobel, D. Wittman, and Y. Wang, ‘SAFETY FIRST FOR AUTOMATED DRIVING’. Accessed: May 26, 2022. [Online]. Available: https://www.mercedes-benz.com/content/dam/brandhub/innovation/safety-first-for-automated-driving/safety-first-for-automated-driving-withepaper_en.pdf

[5] NHTSA, ‘Automated vehicles for safety’. https://www.nhtsa.gov/technology-innovation/automated-vehicles-safety (accessed Jun. 07, 2022).

[6] R. Tremosa, ‘Software architectural design for safety in automated parking system’, Universitat Politècnica de Catalunya (ETSETB), 2022.

[7] AUTOSAR, ‘General information about AUTOSAR’. Accessed: Mar. 15, 2022. [Online]. Available: https://www.autosar.org/about/

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

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

[10] SYNOPSYS, ‘Software Architecture & Software Security Design’. https://www.synopsys.com/glossary/what-is-software-architecture.html (accessed Jun. 07, 2022).

[11] D. Garlan, ‘Software architecture: a roadmap’, Pittsburgh, Pennsylvania, May 2000, pp. 91–101. Accessed: Jun. 14, 2022. [Online]. Available: https://dl.acm.org/doi/10.1145/336512.336537

[12] B. L, C. P, and K. R, ‘Software Architecture in Practice’, *2012*.

[13] Wikipedia, ‘Reference architecture’, Accessed: May 20, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Reference_architecture

[14] AUTOSAR, ‘12 th AUTOSAR Open Conference’, 2021. Accessed: Mar. 12, 2022. [Online]. Available: https://www.autosar.org/news-events/details/12th-autosar-open-conference-2021-feb-05/

[15] AUTOSAR, ‘CLASSIC PLATFORM’, Accessed: May 23, 2022. [Online]. Available: https://www.autosar.org/standards/classic-platform/

[16] J. Dorfner, ‘AUTOSAR – The Classic Platform’, Accessed: May 23, 2022. [Online]. Available: https://www.methodpark.de/blog/autosar-part-1-the-classic-platform/

[17] ETAS, ‘RTA-RTE User Guide’.

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

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

[20] G. Regulwar and P. Jawandhiya, ‘Variations in V Model for Software Development’, Jan. 2021. [Online]. Available: https://www.researchgate.net/publication/348488626_Variations_in_V_Model_for_Software_Development

[21] 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/

[22] 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

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

[24] International Organization for Standardization (ISO), ‘ISO standard 26262:2011 Road vehicles - Functional safety’, 2011. Accessed: May 14, 2020. [Online]. Available: https://www.iso.org/standard/68383.html

[25] 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

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

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

[28] Wikipedia, ‘ASAM - Association for Standardisation of Automation and Measuring Systems’. Accessed: May 24, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Association_for_Standardisation_of_Automation_and_Measuring_Systems

[29] ASAM e.V.all, *Webinar - Introduction to ASAM OpenODD Concept*. Accessed: May 24,

ETSEIB

2022. [Online]. Available: https://www.asam.net/conferences-events/detail/webinar-asam-openodd-concept/

[30] BSI, ‘PAS 1883:2020 - Operational Design Domain (ODD) : taxonomy for an automated driving system (ADS) – Specification’. Aug. 2020. Accessed: Jun. 02, 2022. [Online]. Available: https://www.bsigroup.com/globalassets/localfiles/en-gb/cav/pas1883.pdf

[31] ‘SOTIF_documentation’. [Online]. Available: https://www.perforce.com/blog/qac/sotif-iso-pas-21448-autonomous-driving

[32] AUTOSAR Release Management, ‘Requirements on Run-Time Environment’, 84, Nov. 2021. [Online]. Available: 04/05/2022

[33] AUTOSAR Release Managemen, ‘Specification of RTE Software’, AUTOSAR, 84. [Online]. Available: 01/04/2022

[34] CARLA team, ‘CARLA Simulator’, 2021. https://carla.readthedocs.io/en/latest/ (accessed May 01, 2022).

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

[36] AUTOSAR Release Management, ‘Guide to Mode Management - CP’, 440, Nov. 2021. [Online]. Available: 04/05/2022

[37] Bosch Mobility Solutions, ‘Sense, Think, Act: What automated vehicles need to be capable of’. Sense, Think, Act: What automated vehicles need to be capable of (accessed May 25, 2022).

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

[39] 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.

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

[41] CARLA team, ‘Carla World definition. Getters and methods defined.’ https://carla.readthedocs.io/en/latest/python_api/ (accessed Jun. 06, 2022).

[42] CARLA team, ‘Carla’s environment objects - Semantic tag definition objects’. https://carla.readthedocs.io/en/latest/ref_sensors/#semantic-segmentation-camera (accessed Jun. 06, 2022).

[43] ‘Carla environmnet objects - Defintion of the all environment objects in the simulation world. Return of the get_environment_objects function.’ https://carla.readthedocs.io/en/latest/python_api/#carla.EnvironmentObject

[44] CARLA team, ‘Carla Light Manager - Definition of the light Groups’. https://carla.readthedocs.io/en/latest/python_api/#carla.LightGroup (accessed Jun. 06,

2022).

[45] CARLA team, ‘Carla light group. Definition of the return of the function get_turned_on_lights(self, light_group).’ https://carla.readthedocs.io/en/latest/python_api/#carla.Light (accessed Jun. 06, 2022).

[46] CARLA team, ‘CARLA Sensor types’. https://carla.readthedocs.io/en/latest/python_api/#carla.SensorData (accessed Jun. 30, 2022).

[47] P. G. Bejerano, ‘Sensors required for autonomous driving.’ https://blogthinkbig.com/sensores-esencia-coche-autonomo/

[48] T. Dawkins, ‘Sensors used in autonomous vehicles.’, Aug. 03, 2021. https://levelfivesupplies.com/es/automoviles-autonomos-101-que-sensores-se-utilizan-en-los-vehiculos-autonomos/ (accessed May 30, 2022).

[49] AUTOSAR Release Managemen, ‘Specification of Sensor Interfaces’.

[50] C. Atwell, ‘Radar sensor’. https://www.fierceelectronics.com/sensors/what-a-radar-sensor (accessed May 30, 2022).

[51] CVEL, ‘Distance Sensors -RADAR’. https://cecas.clemson.edu/cvel/auto/sensors/distance-radar.html (accessed May 31, 2022).

[52] CARLA team, ‘CARLA Radar sensor’. https://carla.readthedocs.io/en/latest/python_api/#carla.RadarDetection (accessed May 30, 2022).

[53] The Zebra, ‘How do Self-Driving cars work? Deep description.’, Apr. 28, 2022. https://www.thezebra.com/resources/driving/how-do-self-driving-cars-work/ (accessed May 30, 2022).

[54] P. Bejerano, ‘5 sensors: here’s what a car needs to drive like a human’. https://blogthinkbig.com/sensores-esencia-coche-autonomo#:~:text=Los%20principales%20sensores%20con%20los,otros%20adicionales%2C%20como%20el%20sonar. (accessed May 30, 2022).

[55] CARLA team, ‘Carla Image (Camera)’. https://carla.readthedocs.io/en/latest/python_api/#carla.Image (accessed May 30, 2022).

[56] CARLA team, ‘CARLA Color converter’. https://carla.readthedocs.io/en/latest/python_api/#carla.ColorConverter (accessed May 30, 2022).

[57] G. Sebastian, T. Cattem, L. Lukic, C. Bürgy, and T. Schumann, ‘RangeWeatherNet for LiDAR-only weather and road condition classification’, Nagoya, Japan, Jul. 2021. Accessed: May 31, 2022. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9575320

[58] CARLA team, ‘CARLA LIDAR measurement sensor data’.

https://carla.readthedocs.io/en/latest/python_api/#carla.LidarMeasurement (accessed May 31, 2022).

[59] CARLA team, 'CARLA Location data'. https://carla.readthedocs.io/en/latest/python_api/#carla.Location (accessed May 31, 2022).

[60] CARLA team, 'Semantic Lidar sensor - Definition in CARLA simulator'. https://carla.readthedocs.io/en/latest/ref_sensors/#semantic-lidar-sensor (accessed May 31, 2022).

[61] CARLA team, 'CARLA semantic lidar definition of the exchanged data.' https://carla.readthedocs.io/en/latest/python_api/#carla.SemanticLidarMeasurement (accessed May 31, 2022).

[62] RoboticsBiz, 'Impact of weather conditions on sensors in autonomous vehicles', Mar. 09, 2022. https://roboticsbiz.com/impact-of-weather-conditions-on-sensors-in-autonomous-vehicles/ (accessed May 31, 2022).

[63] CARLA team, 'CARLA GNSS Sensor module - Definition'. https://carla.readthedocs.io/en/latest/python_api/#carla.GnssMeasurement (accessed May 31, 2022).

[64] Sensible4, 'Inertial Measurement Units for Localisation and Navigation of Autonomous Vehicles', Jul. 2021, Accessed: May 31, 2022. [Online]. Available: https://sensible4.fi/technology/articles/imu-and-autonomous-vehicles/#:~:text=It%20consists%20of%20two%20sensors,current%20vehicle%20location%20and%20orientation.

[65] CARLA team, 'CARLA IMU sensor-Definition'. https://carla.readthedocs.io/en/latest/python_api/#carla.IMUMeasurement (accessed May 31, 2022).

[66] CARLA team, 'CARLA Vector 3D', Apr. 21, 2020. https://carla.readthedocs.io/en/latest/python_api/#carla.Vector3D (accessed May 31, 2022).

[67] ETAS, 'RTA-CAR User Guide v9.2.1'.

[68] A. Harma, 'Local Host', p. 3, 2021.

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

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

[71] N. Jennings, 'Socket Programming in Python (Guide)', Accessed: Jun. 04, 2022. [Online]. Available: https://realpython.com/python-sockets/

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

ETSEIB

[73] 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

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