Christoph Pilz, BSc

Development of a Scenario Simulation Platform to Support Autonomous Driving Verification

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Software Engineering and Management

submitted to

Graz University of Technology

Supervisor

Assoc.Prof. Dipl-Ing. Dr.techn. Gerald Steinbauer

Institute for Softwaretechnology

Graz, April 2019

This document is set in Palatino, compiled with pdfLaTeX2e and `Biber`.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date	Signature

Abstract

Automotive industry is currently shifting from autonomous driving assistance systems to conditionally automated vehicles. Traditional automotive component testing methodologies are not enough to verify these increasingly complex systems. While previous research deals primarily with elementary components of complex verification systems for autonomous driving, only commercial software companies combine them without scientifically publishing results. The focus of this master thesis is to analyze the components necessary to design and build an autonomous driving verification system. The results of this analysis are then integrated into a proof-of-concept system whose performance is compared with expected results. The outcome of this master thesis provides a scientific basis for future developments of autonomous driving verification systems for automotive appliances.

Acknowledgements

The research work presented in this thesis was carried out at the VIRTUAL VEHICLE Research Center in Graz in cooperation with the institute for software technologies at technical university of Graz.

The project was funded by the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No 737469 and partially by the COMET K2 – Competence Centers for Excellent Technologies – Program.

I would like to thank my colleagues at the virtual vehicle competence center for the warm work environment. Ahead of everyone I would like to thank Markus Schratter for the close support with technical discussions.

I also extend my gratitude to Prof. Gerald Steinbauer for providing a scientific guideline and for clearing up misconceptions during the planning phases.

Finally I would like to thank my family: not only for the support during my studies, but for my whole life so far.

Funding

The grant agreement No 737469 of the Electronic Component Systems for European Leadership Joint Undertaking receives support from the European Union's Horizon 2020 research and innovation programme as well as Germany, Austria, Spain, Italy, Latvia, Belgium, Netherlands, Sweden, Finland, Lithuania, Czech Republic, Romania, Norway. In Austria the project was also funded by the program "IKT der Zukunft" and the Austrian Federal Ministry for Transport, Innovation and Technology (bmvit).

The VIRTUAL VEHICLE Research Center in Graz and therefore this thesis are partially funded by the COMET K2 — Competence Centers for Excellent Technologies — Program of the Federal Ministry for Transport, Innovation and Technology (bmvit), the Federal Ministry for Digital, Business and Enterprise (bmdw), the Austrian Research Promotion Agency (FFG), the Province of Styria and the Styrian Business Promotion Agency (SFG).

Contents

List of Figures

1 Introduction

Automotive industry is currently in the middle of going from advanced driver assistance systems (ADAS) to automated driving systems (ADS). More specific, the Society of Automotive Engineers (SAE) published a manual [1] in 2014 where they defined six levels of driving automation as shown in Figure 1.2. Many Original Equipment Manufacturers (OEMs) already equip their cars with level 1 systems like Adaptive Cruise Control (ACC) and Lane Keeping Assistant (LKA). Some of these OEMs even provide level 2 systems like the Traffic Jam Assistant (TJA) which guarantee an automated vehicle in traffic jam situations.

To guarantee a specific functionality of a system, the system has to be tested. The testing of an ADAS is generally easier than the testing of an ADS, as the tolerable rate of failures is far smaller. The problem lies with the nature of ADSs. As [1] states, conditional automation (level 3) supports a discrete set of driving modes, like traffic jams, but still has human interaction as a fallback. Especially when upgrading to highly automated systems in level 4, the system does not have the option of a human driver as fallback for specific situations as driving on the freeway. Because of that, the system has to provide higher safety levels, which in further consequence needs more testing.

For example one might compare a basic electronic control unit (ECU) like the direction-indicator control and an ADAS like ACC with an ADS for parking maneuvers. The direction-indicator control has a discrete set of states which lead to an easy to grasp set of input-output relations. Beginning with ADAS like ACC, it is already more complex. The simulated input can not be mapped to discrete states, like a single image or a single distance measure. Most of the time, a consecutive set of measurements is needed to determine an action. Even more, the output influences the input.

Coming now to ADS, the set of states is far from discrete as visualized in Figure 1.1. First of all, the output of acceleration and steering massively influence continuous test scenarios. Second, sensor and vehicle physics have to be taken into account. And third, ADS are a collection of several individual components. In other words, a collision avoidance system can only *suggest* evasive routes. Meaning, the decision has to be based on other semantic information like object classification and traffic rules.

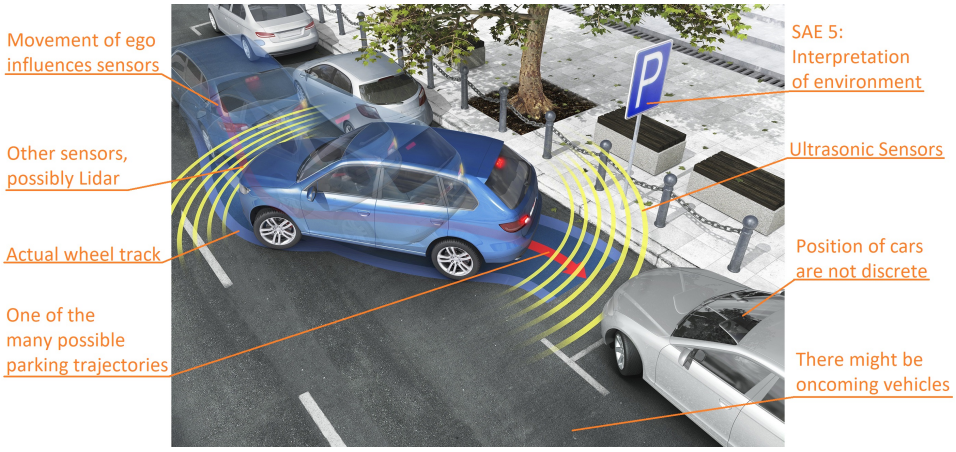

Figure 1.1: Automatic Parking [2]

SAE level	Name	Narrative Definition	Execution of Steering and Acceleration/ Deceleration	*Monitoring* of Driving Environment	Fallback Performance of *Dynamic Driving Task*	System Capability *(Driving Modes)*
***Human driver* monitors the driving environment**						
0	**No Automation**	the full-time performance by the *human driver* of all aspects of the *dynamic driving task*, even when enhanced by warning or intervention systems	Human driver	Human driver	Human driver	n/a
1	**Driver Assistance**	the *driving mode*-specific execution by a driver assistance system of either steering or acceleration/deceleration using information about the driving environment and with the expectation that the *human driver* perform all remaining aspects of the *dynamic driving task*	Human driver and system	Human driver	Human driver	Some driving modes
2	**Partial Automation**	the *driving mode*-specific execution by one or more driver assistance systems of both steering and acceleration/ deceleration using information about the driving environment and with the expectation that the *human driver* perform all remaining aspects of the *dynamic driving task*	**System**	Human driver	Human driver	Some driving modes
***Automated driving system* ("system") monitors the driving environment**						
3	**Conditional Automation**	the *driving mode*-specific performance by an *automated driving system* of all aspects of the dynamic driving task with the expectation that the *human driver* will respond appropriately to a *request to intervene*	System	**System**	Human driver	Some driving modes
4	**High Automation**	the *driving mode*-specific performance by an automated driving system of all aspects of the *dynamic driving task*, even if a *human driver* does not respond appropriately to a *request to intervene*	System	System	**System**	Some driving modes
5	**Full Automation**	the full-time performance by an *automated driving system* of all aspects of the *dynamic driving task* under all roadway and environmental conditions that can be managed by a *human driver*	System	System	System	**All driving modes**

Figure 1.2: Six Levels of Driving Automation [3], [1]

1.1 Motivation

Many papers provide an overview on methods to test ECUs in automotive industry. This part is important, as the components themselves are tested. But for higher automation levels, the whole system is important. The whole system is thereby the physical car and all its mechanical, electrical components, as well as the controlling software. Traditionally a prototype car is tested by test drivers in closed environments and later on, when the basic stability of the car and its subsystems are confirmed, the cars can be tested on public roads. To legally test ADS on the ALP.Lab [4] test track — which is part of a freeway in Austria — 15000 driven kilometers are necessary. Whereby at least 1000 kilometers have to be driven on a test track and the rest can be done in simulations.

To cut down the time needed for testing, simulators are sped up to test different scenarios in shorter time frames. The market for simulators is highly competitive as Chapter 5.1.2 will show. However, most of the manufacturers keep the design of their system a secret. Furthermore, only a few papers directly target efficient, scalable testing with realistic graphics and physics.

In other words, scientific research concentrates primarily on individual fundamentals of simulated verification as presented in Chapter 4. Therefore this thesis aims to combine these existing elements. The result achieved by closing the gaps will then support existing software with a scientific foundation.

1.2 Goals and Challenges

In terms of testing, this thesis has the focus on software in the loop (SIL) testing. To be more specific, existing automated driving (AD) software is coupled with a simulator. The architecture of the interface should also support testing of multiple different traffic scenarios.

The expected primary difficulties can be summarized as follows: (i) the range of functions of existing simulators, (ii) the formatting of scenarios and (iii) the architecture of the controlling software.

As Chapter 5.1 will show, existing simulators vary in their field of features. The disparities do not only comprise graphics and available vehicle models. Especially the available communication interfaces, sensors and rendering engine implementations are designed for specific use cases.

Concerning scenario formatting, Chapter 5.2 summarizes the differences in modeling scenarios. While random assisted tests might be sufficient to verify functionality, its hard to analyze and reproduce or to even test specific situations.

On the software side scalability, performance and parallelizability are major factors. The better these factors are met, the more kilometer can be driven in specific time frames. Which leads to safer vehicles as the likelihood to discover bugs increases.

1.3 Contribution

This thesis provides a guideline for the development of automatic test environments for SIL testing of ADSs. The focus points are hereby (i) the analysis of existing simulators as rendering engines for virtual environments, (ii) the discussion of methods for modeling scenarios and (iii) the design process of architecture for scenario testing software.

To support the findings of this thesis, a proof of concept software will point out difficulties during the implementation and limits of the used hardware and software. The limitations found are then further analyzed to support future development.

1.4 Outline

This thesis will continue with Chapter 4, which contains: (i) an analysis of existing testing methods for ADAS, (ii) an overview of existing simulators, their strengths and weaknesses and (iii) a summary of methods to describe scenarios. After the preliminary research, a system will be designed based on scalability and modular expandability in Chapter 7. Chapter 8 will point out some details of the implementation. Finally Chapter 9 will analyze the proof of concept software and Chapter 10 will close with suggestions for future developments.

2 Important Terms

Science in the field of ADAS and ADS testing uses a few terms which might be unfamiliar. To avoid misunderstandings and provide a quick reference, this Chapter outlines the most important and most used terms in this thesis.

ADAS and ADS are distinguishable via their purpose in SAE levels [1]. ADASs are assistance systems to support the driver in specific situations, while ADSs posses the possibility to autonomously drive a car in specific situations. Therefore ADASs find purpose only in the lower SAE levels.

HIL, MIL and SIL are often used when talking about verification and validation of automotive systems. The "IL" in these terms stands for "in the loop". It means that a specific part is taken from its original surroundings and the real surroundings are emulated. In other words, the virtual environment provides valid and invalid data. The system under test then has to produce output which is checked for valid behavior.

Chronologically the three main test methods can be sorted by model in the loop (MIL), SIL and hardware in the loop (HIL), whereby parallel development may be possible. MIL deals with models of systems in the loop. When a component — like an ECU — is designed it gets specific requirements. These requirements are put into a model (i.e. Matlab). The model can then be verified before any hardware or software costs emerge.

The principle of MIL can be applied to SIL and HIL. Virtual input is provided first to the software itself, then to the software running on actual hardware. The target of all methods is to validate the output of the components, given a specific input..

VeHIL is a concept introduced by TNO Automotive [5]. The term vehicle hardware in the loop (VeHIL) is a form of HIL where a prototype vehicle is placed on a dynamometer. Real life objects are then moved on platforms towards and past the vehicle. The whole system is a very expensive way of verifying ADASs, as a lot of space and equipment is necessary.

VIL is similar to VeHIL. Both use a prototype car. But while VeHIL tests the car itself, vehicle in the loop (VIL) has a human driver in the car. Also different to VeHIL, the car is driving on real roads or parking lots. So physics and sensor inputs are real at first. VIL additionally overlays the sensor input, as discussed by [6]. For example other cars can be overlayed on the real life camera input. The driver is also provided with the virtual environment via Augmented Reality (AR). With the support of a head mounted display (HMD), the driver can monitor the situation at any given time.

At this point it is important to point out the difference between VIL testing and verification of AD software. While the full software of an AD car may be tested and validated in a virtual environment, its still SIL testing. Even if physics and graphics are close to the real world.

Scene, Situation and Scenario are three common terms in AD scenario research. Definition of these terms is very technophilosophical. The authors of [7] provide a detailed analysis of each concept. For a quick summary, the following lines provide a reference, while Figure 4.4 in Chapter 4.4 shows the terms in a simple simulator implementation

The scene is the modeled environment, with dynamic aspects like weather, all actors, the road layout and all surrounding objects.

The situation is the current state of the environment. The current weather of the simulator, the current settings of the traffic lights and the current state of all actors.

The Scenario is a screenplay. Its a sequence of actions and events within a scenario. When a scenario is played, each single step forms a situation.

Ego Vehicle is the vehicle under test. For SIL tests in context of this thesis it is also the only vehicle which is controlled by external software. Consequently "non-ego vehicles" are all other — internally controlled — vehicles.

Robot Operating System (ROS) is a software environment administered by the Open Source Robotics Foundation [8]. robot operating system (ROS) has the advantage of a big community and high compatibility with different hardware. Due to easy and fast setup procedures it is very widespread in prototyping robotics. For a quick and good structured overview of ROS, especially in context of testing ADASs and ADSs, [9] is a good start.

3 Problem Formulation

Research and development in the automotive industry is on the verge of implementing AD systems in their cars with high SAE levels as introduced in Chapter 1. As systems get more complex, also testing is more complex. This in further consequence creates the need for more sophisticated simulation environments.

The problem can be exemplified with the following test scenario: ego vehicle is driving straight ahead in the city along parking cars. A person jumps out between the parked cars. The ego vehicle (i) must not hit the person, (ii) must not hit other things and (iii) must comply with traffic rules.

To evaluate this scenario, three main necessary components can be identified: (i) the input has to be defined. The scenario defined as text has to be put in a computer parsable and reusable format. Furthermore, as mentioned in Chapter 1, (ii) the complexity of the input-output relations require the assistance of a 3D simulation to provide a physics model for evasive maneuvers. Finally, (iii) the outcome has to be evaluated. Did the test case fail or succeed? How can it be determined?

The goal of this thesis is to find out how to design a test platform to do SIL testing of complex ADS. With the above requirements, one can say that the system has to rest on three pillars: (i) a 3D simulation engine, (ii) a scenario format and (iii) a controlling software.

The Simulation Engine has to be capable of simulating graphics and physics close to realism. Additionally it has to have interfaces, able to connect to a vehicle platform. Chapter 5.1.1 will take these vague requirements and pinpoint quantifiable parameters to discuss the selection of a simulation engine.

The Scenario Format has to be light-weight, but still capable to support more complex scenarios in the future. Chapter 5.2.1 will further analyze the requirements necessary for a format of this kind.

Controlling Software has to read and prepare the scenarios. It then has to communicate with the simulation engine, to load and play the scenario. It also has to synchronize with the AD-system to guarantee the SIL testing. In the end it has to report the success or fail of the tested scenario.

As there are already simulators with such capabilities on the market, it can be anticipated that there are publications for each individual pillar. But in the end, the overall concept should be analyzed.

4 Related Research

The first approach towards a solution for the problem stated in Chapter 3 was to find related research. Therefore, the paper headlines of five different symposia were skimmed, limited to the last five years. While the International Conference on Intelligent Transportation Systems (ITSC) and the Intelligent Vehicles Symposium (IV) provided the majority of relevant papers, the International Conference on Intelligent Robots and Systems (IROS), the International Conference on Autonomous Robot Systems and Competitions (ICARSC) and the International Conference on Robotics and Automation (ICRA) also provided interesting insight into simulation aided testing. During this search method, around 250-300 papers were collected, with a title suggesting some influence to simulated testing of ADAS and ADS.

From this accumulated pool of papers, around eighty were at least partly in the context of the simulation part of this thesis. Around fifteen additional papers were at least close to the scenario generation component. These left over papers have all been skimmed to find additional suitable papers. During this process the papers also got rated to get the most relevant.

None of the read papers directly covered the topic of this thesis. But they helped to get necessary background information and guidelines for different parts of this project. Hence, Chapters 4.1-4.6 will point out related research done in the field of testing ADAS and ADS in descending applicability.

4.1 High Level Components Validation

To save costs and time during development, every component in a car is traditionally tested in stand alone environments. Simple components

like ECUs controlling the headlights or the basic form of anti-lock braking system (ABS) are simple to test. Only a small set of inputs leads to a small set of different outputs.

If, however, one considers more complex components like ADAS, the input and output sets get less trivial. For testing its necessary to provide virtual environments. Else it is very hard to reproduce a sequence of events, leading to a correct functioning system. A general guideline for the steps necessary to test and evaluate new ADAS, is provided by [10]. This paper deals with the first step of using virtual testing to validate automatic car parking systems. The authors point out the importance of virtual environments: the input for the ADAS depends on physical reactions of the car. Specifically, the trajectory of the parking maneuver itself has a major influence on the sensors.

Overall [10] presents a good comparison between real world testing and simulated testing. It also confirmed the basic idea from robotics to simply deal with the pose of involved vehicles, which will be used in the final implementation. However, the method applied in the paper lacks the ability to automatically run and evaluate scenarios.

Moving a step further, [11] deals with a more complex simulation environment to validate high level ADAS components like an automated emergency braking assistance (AEB) system. The authors built a co-simulation system which combines a *vehicle mechanics and sensor simulation* with the traffic flow simulation engine SUMO. An important point made by this paper is that the co-simulation environment decouples the testing logic from the physical simulation part. In other words, the test cases are less dependent on the physical behavior of a car and can be generalized. Which furthermore reduces the complexity and increases the test coverage.

Compared to the previous paper ([10]), the task in [11] is therefore closer to the problem formulated in Chapter 3. Most importantly, it provides a basic architecture on how to stimulate high level AD components with a simulator. Figure 4.1 shows the proposed concept, which will also be taken into account by this thesis. Finally it can be said that the system designed within [11] is able to generate different traffic scenarios for coverage. But overall it lacks the possibility to easily define specific scenarios as well as a suitable 3D simulation engine.

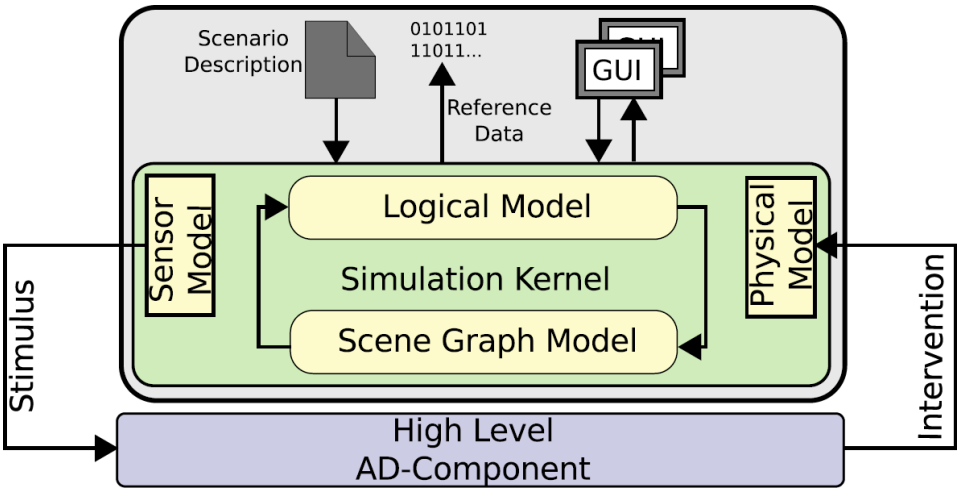

Figure 4.1: Testing Taxonomy [12]

4.2 Test Criteria And Test Metrics

Important for validation is the usage of test criteria and metrics. Several papers like [12] and [13] provide more general information on taxonomy and criteria for testing ADAS. Thereby, [12] focuses on the introduction of test criteria and metrics. A noteworthy point is that test criteria have a strong relation to test scenarios and test metrics have a strong relation to reference knowledge of the results as shown in Figure 4.2. Additionally, test methodologies in general have a big influence on the kind of virtual test environment. Therefore [13] discusses different approaches like VeHIL and VIL.

These two papers have a more general view on the principles of ADAS testing. Although using a very high level approach, they provide fundamental building blocks for this thesis. On the one hand, these are the mentioned criteria and metrics to decide if a test case succeeded or failed. On the other hand, these are the principle testing approaches of VeHIL, VIL and ADAS-SIL. This information has to be kept in mind when designing and evaluating scenarios as test cases.

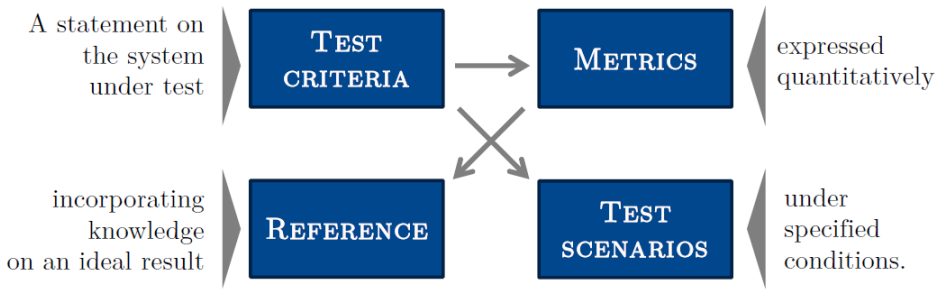

Figure 4.2: Testing Taxonomy [12]

4.3 Simulator Architectures

With the definition of metrics and criteria from the previous Chapter 4.2, the high level testing of ADAS approaches multi-domain simulation. The authors of [14] discuss the basic structure of a simulator for high level and multi-domain ADAS testing, as shown in Figure 4.3. The integration part of this thesis will use a simulator, already consisting of these elements. Nevertheless, it is important to understand the underlying architecture to combine it with a scenario loading engine.

Concerning other implementations of ADAS simulators, three papers have to be pointed out. In the first one [15] the authors introduce a photorealistic simulator and compare it to USARSim and Gazebo. In 2014 this concept was a great idea, as photorealistic graphics engines for computer games were still under development. But as Chapter 5.1.2 will show, the development of engines for decent graphics and physics renders this approach a graphically detailed yet too complicated solution.

The second conceptional different architecture was also published in 2014 within [16] and deals with semivirtual simulations. The architecture utilizes real world driving data and renders virtual objects and obstacles into the sensor data. This approach again provides good sources for basic sensor data. However, the problem with this approach is again the need for real life data. While state of the art simulators are not yet capable to provide real world physics and graphics, they are close enough to render approaches like semivirtual simulations infeasable.

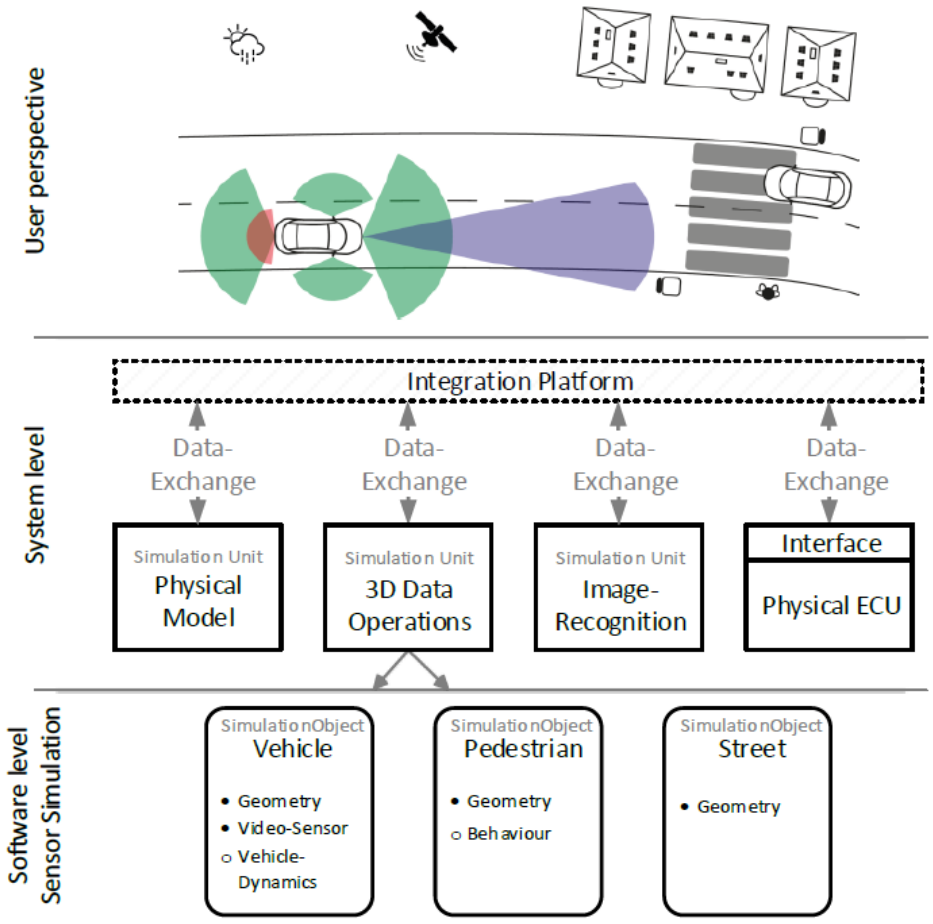

Figure 4.3: Architectural levels of a multi-domain simulator [14]

Finally, the third interesting architectural approach published within [17] already utilizes a 3D simulator for testing. The relation to this thesis is the extra functionality of vehicular ad hoc network (VANET) communication. This extra functionality has influence on the behavior of the vehicles in the 3D simulator, similar to the planned ego control of this thesis. However, like the other simulation engines presented in this Chapter, the system in [17] was not designed to automatically load independently designed scenarios as test cases.

Important to point out is also the fact that previous research dealt a lot with the co-simulation of cooperative maneuver scenarios in contrast to the verification of ADS with simulators. Therefore, most of the papers used graphically simpler traffic simulations like SUMO. Only more recent papers like [17] started to upgrade to 3D simulators. But yet most of them lack the idea of using a specific standard to load scenarios.

4.4 Scenario Terminology

Each field of research has its own Terminology. Although the terminology of AD scenario research is on the edge of related research for this thesis, [7] has to be stated as prime source for the basics. In this paper, the authors discuss the fundamentals of scenario description. Especially the concepts scene, situation and scenario can get mixed up easily. For a quick reference, Figure 4.4 illustrates these terms.

Within this thesis, [7] was also very helpful to identify the core concepts of scenario modeling. It provides a more objective view when looking for a fitting scenario format. In further consequence it even provides better insight into the goals and values when defining and parsing a scene.

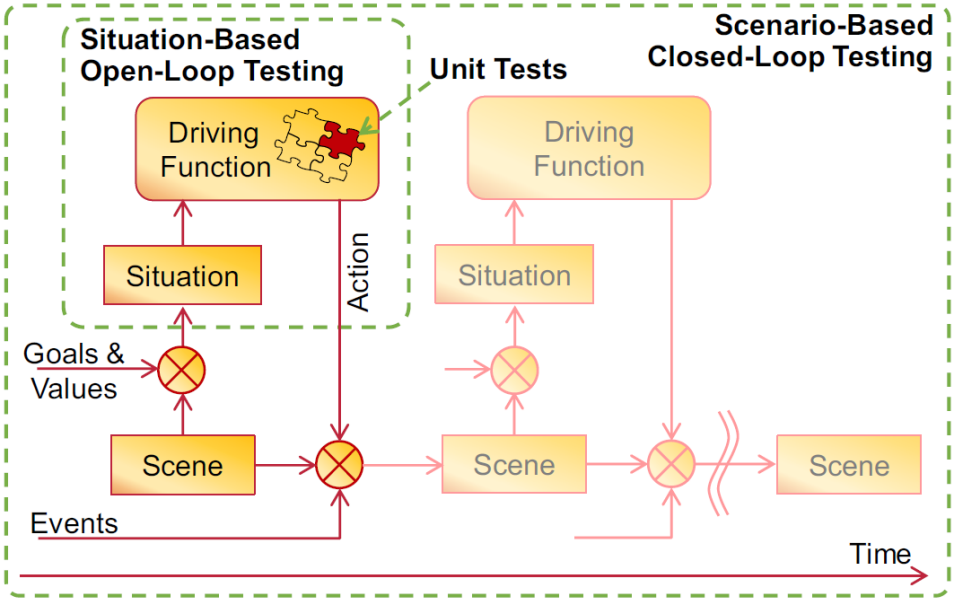

Figure 4.4: Scenario-Based Closed-Loop Testing [7]

4.5 Scenario Modeling

Compared to combustion engine research, ADAS and AD research has just started. Therefore the field of scenario research offers diverse papers with basics. For example in 2017 [18] discussed the automated generation of diverse and challenging scenarios for the test and evaluation of autonomous vehicles. In fact, the simulation engine Cognata [19] also reviewed in Chapter 5.1.2 uses deep learning algorithms to provide diverse scenarios.

However, the creators of these simulation tools have not published their architectural approach of the scenario description. In consequence it is interesting to know how diverse scenarios are generated, but the results are not applicable to this thesis, as the created software should be able to use scenarios, but not generate them.

Other research teams, like the authors of [20] deal with the generation of test case libraries. Most of them concentrate on test cases generated from national traffic accident databases. However, the authors of the mentioned

paper go a step further and use the naturalistic driving data from test drives. These test drives are automatically analyzed and compressed into a special database scenario format. Looking at the practical application, the team evaluated the database format and did not try to couple it with a simulation engine. Even more, the scenario format is too specific and not open enough to be used for more general applications as within this thesis

Scenario formats in database form lead to the third important paper in this Chapter. The authors of [21] use software engineering methods to describe a database format for scenarios. Their scenario database domain model, shown in Figure 4.5 has the advantage that everything is decoupled from each other. Different to the previous paper ([20]), the authors of [21] implemented a small simulation engine to play their scenarios. However, they did not try to couple their database with a complete 3D physics and graphics simulation engine. As Chapter 5.2 will elaborate, the scenario format is very fitting for this thesis, which helped to outline the design of the scenario loading mechanism of the resulting software.

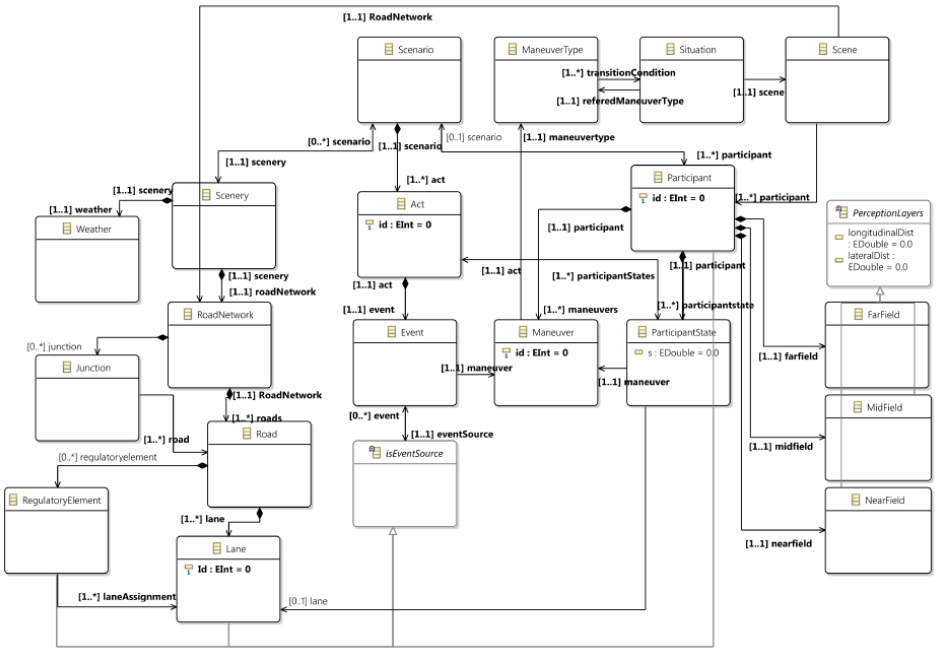

Figure 4.5: Scenario-Based Closed-Loop Testing [7]

4.6 Overlaps With Traditional Robotics

AD cars are basically autonomous robots with enough space for humans to sit in. Traditional robotics is already dealing a long time with the autonomy of robots. For instance the RoboCup [22], first time held in 1997, is famous for the autonomous robotic football matches. Since its foundation, the RoboCup added different Categories. For example the logistics league, which was won by the GRIPS team of the TU-Graz in June 2018 ([23]). However, each robot has to deal with a different kind of uncertainty, which can be simulated. In context of autonomous car robots, uncertainties are the environment around the road, other vehicles and pedestrians.

Robotic engineers have the option to use the widespread ROS software environment. The recommended 3D simulator for ROS software is Gazebo. The authors of [24] analyze the capabilities of ROS and Gazebo in context of simulated testing in robotics. While the authors can show a variety of features applicable to this thesis, the overall package of Gazebo is not applicable. The problem is that Gazebo is designed for robots and not for traffic simulation. Gazebo is good, if one has to create the physical model of a robot. But it lacks scalability for larger traffic scenarios. Furthermore, the paper focuses on the functional scope of ROS and Gazebo, but not on automotive test scenarios in general.

The similarities between traditional robotics and automotive AD vehicles can be shown with [25]. The authors of this paper design a hardware in the loop system for robotic applications. Especially the architecture of the proposed architecture for the robot hardware in the loop (RHIL) has many similarities to the architecture of automotive HIL systems. The difference mainly lies within the interfaces. Although the paper is not applicable to simulation testing of automotive AD software, the approach of synchronization seemed interesting: as the authors execute their RHIL, they simply send a synchronization trigger in the beginning. The rest is done by the simulation. As the RHIL operates in real-time, the simulation model simply has to provide real-time data. However, this approach gets tricky if the simulation is too slow, due to high complexity, as with the case of 3D traffic simulators.

The most applicable paper for scenario loading principles from robotic context is [26]. The authors used a 3D simulator to improve the performance of a grasping robot with deep learning. To train the used network, the authors had to provide different scenarios. The important thing to learn from this paper was environment reusability. To be more specific: the first scenario is loaded with an item in the box. After a successful grasp of the arm, the environment has not to be reloaded. One can instead remove the object, reset the arm and spawn a new object in the box, before resetting the SIL. The same approach is planned to be implemented within this thesis. The server should provide the world map. The vehicles necessary for the test scenario are then spawned and deleted according to the scenario specification.

5 Selection Process For Simulator And Scenario Format

Scenario loaders, as proposed in Chapter 1, need two key components: (i) the simulator, which provides the virtual world with physics and graphics support and (ii) the scenario files, which provide information on the traffic situations to load. The next few Chapters concentrate on finding suitable components. Chapter 5.1 concentrates on finding a suitable simulator conforming to requirements derived from the problem formulation in Chapter 3. Chapter 5.2 will then deal with different scenario formats to find a suitable representation.

5.1 Simulation

The field of simulation includes several applications like HIL, MIL and SIL with steps to VIL and full traffic simulations without external stimuli. For this thesis the next few Chapters concentrate on finding a suitable simulator for SIL testing of ADAS and ADS.

The road ahead will thereby be paved by: (i) Chapter 5.1.1 which starts with a brief overview on the requirements, followed by (ii) Chapter 5.1.2 which grants a broad overview on the strengths and weaknesses of existing simulators, until closing with (iii) Chapter 5.1.3 which points out the final decision.

5.1.1 Requirements

Basic requirements are simple. For a start the simulator has to have compatible interfaces. They either have to be compatible with ROS already, or to be flexible, open and specified enough to support translation between the simulator interface language and ROS message interfaces.
Additionally the simulator has to provide specific information. This comprises basic pose information of the ego vehicle, the full set of non-ego vehicles poses and the pose of pedestrians. Additionally more advanced information, like data streams from camera or light detection and ranging (LIDAR) systems have to be directly accessible.

Due to future plans of the AD group at the VIRTUAL VEHICLE Research Center (VIF), the simulator also has to have graphics and physics close to realism and still be cost efficient. In other words, costs for software and hardware should be reasonable and integrate into other projects.

The requirements were structured by strategic importance and are listed as follows:

- Suitable ROS compatible interface
- Minimal accessible live output:
  - Ego vehicle pose
  - Non-ego vehicle pose
  - Pedestrian pose
  - Camera and LIDAR feeds
- Graphics close to realism
- Physics close to realism
- Cost efficient

5.1.2 Existing Simulators

To find a suitable simulator — for the task stated in Chapter 3 and the requirements further specified in Chapter 5.1.1 — some information was already available and reusable from my preceding master project [27]. The majority of additional information was then gathered by literature research

and extended by desk research and personal interviews with experts at the VIF.

For the simulators found in literature research, the pool of papers from Chapter 4 was taken to extract around ten simulators for testing of ADASs and ADSs. The rest of the simulators was then collected via desk research and informative interviews at the VIF. As one can see in Table 5.1, the conducted desk research was an important part, as many commercial simulators for testing of ADAS and ADS are not published in symposia of the Institute of Electrical and Electronics Engineers (IEEE).

Additionally, papers like [28] and [29] were helpful to get insight into simulator architectures and what to expect from their interfaces. Also, how to evaluate the strength of engines and the advantages/disadvantages of open-source implementations when dealing with configuration files, data interfaces and special sensors.

The evaluated simulators are summarized for an overview in Table 5.1, which also provides ratings with respect to the requirements listed in Chapter 5.1.1. The used symbols should be interpreted as (--) very poor, (-) poor, (o) not rated or irrelevant, (+) good, (++) very good, as well as (i) which indicates that its not yet implemented, but can be done with some effort. Additionally each simulator has either a preceding “I” if it was found during desk research or within informative interviews. Or a preceding “II” if it was found mainly withing literature.

Simulator Name	ROS	Ego	Non	Ped	Sens	Viz	Phy	CE
I-Carla	++	++	++	+	+	+	+	++
I-Gazebo	++	++	++	+	+	o	+	++
I-SynCity	+	++	+	+	++	++	+	--
I-PreScan	i-	++	+	o	++	-	o	--
I-VTD	i-	++	++	o	+	++	++	+
I-AutonoVi-Sim	i	++	++	--	-	--	+	+
I-Righthook	i	++	+	o	+	++	+	--
I-Carmaker	i-	++	+	o	+	+	++	--
I-Racer	i	++	o	--	-	+	++	-
I-SCANeR	i	++	+	o	+	+	+	--
I-AirSim	+	++	+	-	+	+	+	++
I-rFpro	i	++	+	o	+	+	++	--
I-Udacity	++	++	+	--	+	-	o	++
I-Cognata	o	++	+	o	+	++	o	--
II-Aimsun+Vissim	i	+	+	--	-	+	-	+
II-SiVIC	i	++	+	+	++	+	++	--
II-Sumo	i	+	+	o	--	--	--	++
II-RTMaps	i	o	--	--	++	o	o	--
II-CVIS	i	+	+	-	o	--	--	--
II-RoadView	-	--	--	--	+	o	o	+
II-USARSim	i	++	o	-	+	-	o	+
II-PELOPS	i	+	+	--	-	--	--	o
II-MORSE	i	++	o	-	+	-	o	+

Table 5.1: Simulators with respect to suitability

I-Carla

Carla is a free and open-source driving simulator for AD research. The platform, visible in Figure 5.1 is based on the Unreal Engine which handles the graphical and physical modeling. Due to the Open-Source character and also because research appliances are targeted, the interfaces are very open. During run-time you get the current pose of every acting model like vehicles, pedestrians and traffic lights. For the sensor output like camera and LIDAR, the development community even provides a bridge compatible with ROS. [30]

However, the simulator is still in development and has not reached a full release at the time this thesis is written.

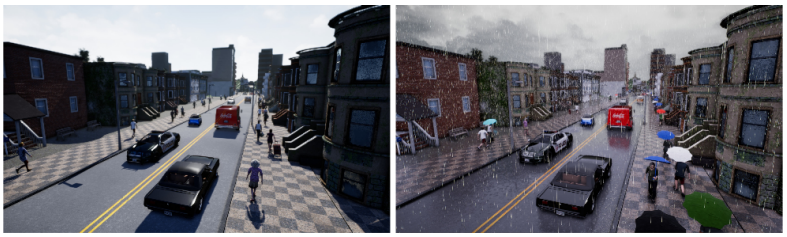

Figure 5.1: Carla Simulator Footage [30]

I-Gazebo

Gazebo is the simulator which can be installed with any newer ROS distribution. The major advantage of Gazebo is its full ROS connectivity. Real world parameters which are accessible in ROS are stimulated by Gazebo. Therefore every pose and sensor information needed for the task of this thesis is directly available. [31]

However, Gazebo has only mediocre graphics, as visible in Figure 5.2, which would be a problem for future appliances for the VIF. Additionally, pedestrians are not animated and a simulation world in general takes more effort to be built than with simulators designed for world simulation. Furthermore, the physics model is light weight meaning not as detailed as its competitors which are designed for detailed physical behavior.

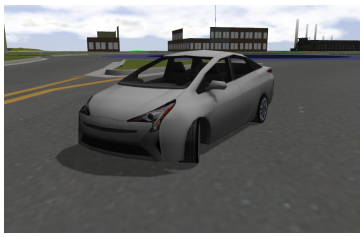

Figure 5.2: Gazebo Simulator Footage [32]

I-Syncity

Syncity is one of the best options for AD simulation. Although it was not possible to get detailed information without buying a license, the available marketing of CVEDIA and several YouTube videos suggest an all-round simulation software for AD.

On the Graphics and Physics side it can be estimated that the platform is at least equal to Carla in its current stage as visible in Figure 5.3. A big feature of Syncity is the possibility to simulate physical effects of rain on camera lenses. On the connectivity side, Syncity provides a ROS interface. Now, at the time of writing this thesis, this simulator also provides an editor for driving scenarios. [33]

However, for research facilities like the VIF also the yearly budget is a big factor. While the software costs would be bearable, the hardware requirements for detailed simulations ruled out the Syncity platform.

Figure 5.3: Syncity Simulator Footage [33]

I-PreScan

PreScan is a simulation platform emerging from sensor simulation. Concerning the sensor side it is a good option. However the problem starts with mediocre graphics, as visible in Figure 5.4, and unavailable ROS-bridge at the time of preliminary research. Even though PreScan can be found in a variety of papers on simulation of different parts of AD, it was not an option for this thesis. [34]

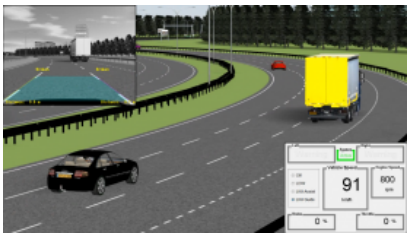

Figure 5.4: PreScan Simulator Footage [34]

I-VTD

VTD, distributed by Vires [35], has the big advantage of open standards for maps (OpenDRIVE [36]) and Scenarios (OpenScenario[37]). It also has suitable physics and graphics as visible in Figure 5.5. However a big disadvantage is the connectivity. While experience at the VIF shows that additional sensors and other functionality is quite easy to implement with tools like MATLAB, the interface itself is rather mysterious compared to OpenSource projects. The reason is that documentation is only available in a very sparse online wiki or directly via training courses.

As it was not possible to estimate the effort necessary to provide ROS connectivity, VTD was ruled out as an option.

Figure 5.5: Vires VTD Simulator Footage [34]

I-AutonoVi-Sim

AutonoVi-Sim is a light weight AD simulator, primarily designed for dynamic maneuver testing. The platform is ranked higher in this list, as it suits the basic idea of scenario testing of this thesis. However graphics were not suitable, as visible in Figure 5.6 and future development of AutonoVi-Sim was not foreseeable at the time of preliminary research. [38]

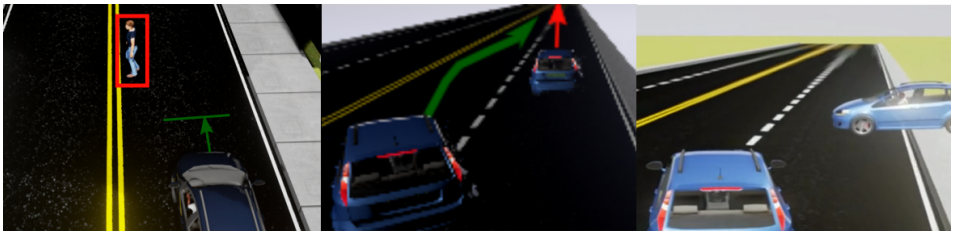

Figure 5.6: AutonoVi-Sim Simulator Footage [38]

I-Righthook

Righthook is similar to SynCity although its strengths are in the support of Artificial Intelligence (AI) training. Physics and Graphics are decent, as visible in Figure 5.7. Sensors are also including LIDAR. However, while there was no direct ROS support at the time of preliminary research, the factors ruling out Righthook were again the high software and hardware costs. [39]

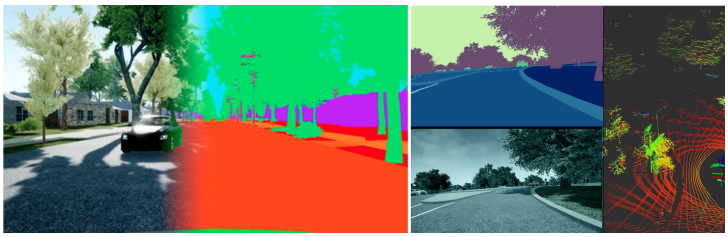

Figure 5.7: Righthook Simulator Footage [39]

I-CarMaker

CarMaker, distributed by IPG, is a simulation software designed to test vehicle dynamics. While it is a sophisticated platform to test vehicle dynamics and ADAS modules, it was in the process of adapting to AD testing methods at the time of preliminary research. Especially ROS support was not available at that time. [40]

As visible in Figure 5.8, graphics would have been optimal, with a physics layer close to realism. However, the missing ROS support ruled out CarMaker, because of better alternatives.

Figure 5.8: CarMaker Simulator Footage [40]

I-Racer

Racer is a free to use platform for non-commercial projects. As the last update seems to be from 2015, the project seems to be halted as of the time writing. Nevertheless, it has a strong physics simulation engine as visible in Figure 5.9 where the heat distribution of the break disc is shown. However, Racer lacks other features like LIDAR sensors or pedestrians. [41]

Figure 5.9: Racer Simulator Footage [41]

I-SCANeR

SCANeR, visible in Figure 5.10 has a similar background as PreScan and similar features as VTD. The reason this platform was not considered is due to high software and hardware costs and better alternatives as the aforementioned PreScan and VTD. [42]

Figure 5.10: SCANeR Simulator Footage [42]

I-AirSim

AirSim is an open-source project very similar to Carla. The main background of AirSim is a simulation platform for drones, as visible in Figure 5.11. However, at the time of preliminary research, AirSim lacked pedestrians and other cars. [43]

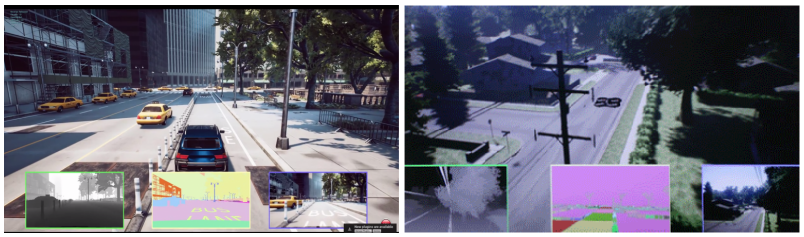

Figure 5.11: AirSim Simulator Footage [43]

I-rFpro

The platform rFpro started as a motorsport simulation project in 2007 within a Formula 1 team. Decent physics and graphics are clearly provided, as visible in Figure 5.12. Non-Ego cars, pedestrians, sensors and scenarios are also included. The only open question was ROS compatibility. But again, due to the high costs, this option was ruled out. [44]

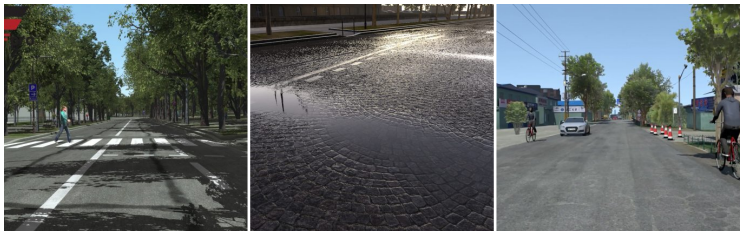

Figure 5.12: rFpro Simulator Footage [43]

I-Udacity

The open-source simulator from the online learning platform Udacity is a light weight version of Carla, as visible in Figure 5.13. Its primary target is to quickly teach people how to do the basics of AD. Beside being a light weight version of Carla, mediocre graphics and the uncertainty of pedestrian support during preliminary research ruled out this option. [45]

Figure 5.13: Udacity Simulator Footage [45]

I-Cognata

The strength of Cognata is its deep learning simulation engine. It is able to generate scenarios for a given vehicle model. Physics and graphics are decent enough for the requirements and also pedestrians and non-ego vehicles are available. ROS support is not explicitly stated, but might be available. However, it has not been further investigated due to cheaper and technically equal alternatives. [19]

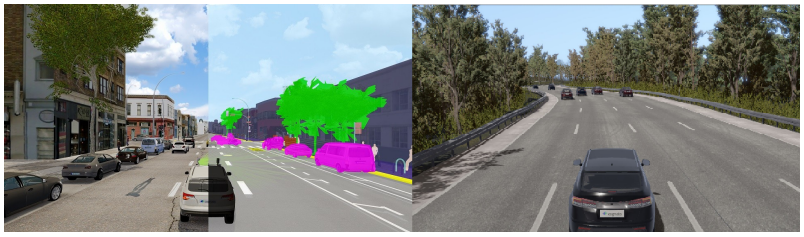

Figure 5.14: Cognata Simulator Footage [19]

II-Aimsun and Vissim

The combination of Aimsun and Vissim is for early stage but detailed simulation of traffic flow behavior. Thereby Vissim is the visual component with graphics comparable to VTD. Overall, the focus is on traffic simulation and does not support detailed sensors as required for this thesis. [46]

II-SiVIC

Simulator of Vehicle, Infrastructure and sensor (SIVIC) was created for ADAS prototyping, test and validation. The Software has decent graphics as visible in Figure 5.15, sophisticated physics similar to Vires VTD and IPG CarMaker. It is also important to note that in the early stages, focus was on sensors. Therefore SIVIC is one of the few simulators with radar support. [47]

However, even though SIVIC has pedestrians from infrastructure simulation, its expensive and does not support ROS out of the box.

Figure 5.15: SiVIC Simulator Footage [47]

II-Sumo

Sumo can be found in various research papers about traffic simulation, like [48]. It has no suitable graphical output for sensors, because it is mostly a top down representation of traffic, as visible in Figure 5.16. But when talking about simulators, it is important to mention Sumo, due to the amount of citations and usage in traffic simulation papers.

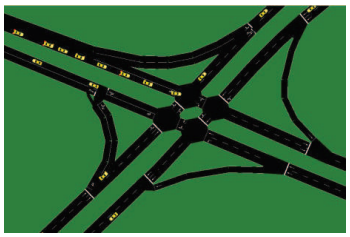

Figure 5.16: Sumo Simulator Footage [47]

II-RTMaps

RTMaps is a sensor simulation platform. It is designed to record and playback raw sensor data. RTMaps can be combined with other simulators, like the above stated SiVIC, but is no standalone simulator in the sense of generating some virtual world. [10]

II-CVIS

Similar to Sumo and RTMaps, Cooperative Vehicle Infrastructure System (CVIS) is a simulation tool for a specific part of ADS and ADAS research, development, verification and validation. CVIS can for example be combined with a visual simulation system as presented in [49]. However, for the task in this thesis, the system lacks pedestrians and is rather complicated.

II-RoadView

RoadView is no simulator that can create a virtual environment like Carla, Gazebo or other aforementioned platforms. RoadView simply uses Images, combined with Global Positioning System (GPS)-poses to provide the ADS or ADAS real world input as [15] states. Although RoadView is an interesting approach to validate ADS and ADAS, it is not suitable for this thesis.

II-USARSim

USARSim is a platform very similar to earlier versions of Gazebo. It was designed to provide a simulation platform for robotics, as stated by [50], and can therefore also be used for AD research. Due to the similarity to Gazebo and due to the inferior graphics and overall development state the USARSim platform was ruled out.

II-PELOPS

Programm zur Entwicklung längsdynamischer, mikroskopischer Prozesse in system relevanter Umgebung (PELOPS) is a traffic flow simulation program similar to Sumo, [51]. It is mentioned for completeness, as it could be combined with other visual simulators.

II-MORSE

The Modual OpenRobots Simulation Engine (MORSE) [52] is a platform similar to Gazebo and USARSim. As the simulator has inferior features compared to Gazebo, it has not been in-depth reviewed, due to a better alternative. However, considering completeness and future reference, it has to be listed here.

5.1.3 Decision

The choice finally fell on Carla [30]. The biggest factor in the decision making process was suitability, especially easy compatibility with ROS. About a third of the simulators already dropped out due to design concepts incompatible with the given task. For example RTMaps has its focus on sensors and does not explicitly render virtual environments. Similar to that, RoadView simply synchronizes real world images with GPS coordinates.
Furthermore, the majority of commercial simulators has been ruled out due to high costs not only for software, but also for hardware. At this point it should also be mentioned that the market for AD simulators is very active,

as ADAS and ADS get more and more complex and have to be validated to meet statutory provisions in different countries. In consequence the acquired software could restrict future projects of the VIF. With open-source versions the need for adaption is more prominent, but in the end it is cheaper as the market for simulated AD testing is still very young.

5.2 Scenario Generation

Before going into detail on options available for scenario-describing files, its worth mentioning [7]. In this paper, the fundamental termination of scenario related terms is discussed. The right usage of terms is helpful when talking about scenarios. The paper itself claims to be a guideline for discussions around scenarios, as some terms like scenario and situation can get mixed up.

For a better understanding of the Chapters 5.2.1-5.2.3 the importance of scenarios has to be pointed out. To achieve repeatability and reproducibility the behavior has to be the same in each execution. This can be achieved via storing the random seed of a simulation environment or by controlling each actor via a “script”. While a random seed is rarely a human readable format, a “script” can be built like a storybook. This storybook for a specific set of acts and events is called scenario.

5.2.1 Requirements

Generally speaking, a scenario format has to be lightweight and easy to create. The reason for that is simple. To provide test coverage of a specific amount of test kilometers, one does need a variety of scenarios. Each scenario needs effort to be created and each scenario needs storage space.

Easy creation goes hand in hand with easy to read. If some test case fails, an engineer should be able to backtrack the error in the scenario file. Simple, human readable representation saves time.

As requirements change over time, its important to provide a certain amount of flexibility. The format should be as independent of the simulator in use as possible. Meaning, if the simulator part is exchanged, the format should be reusable.

Last but not least, to fulfill the lightweight concept and to avoid having to change one thing in multiple locations, its important to have few to no redundancy. For example the scenario should be independent of configuration parts. It should not be the scenario file which defines detailed paths. And it should not be the scenario file which describes the used equipment. In the best case, the scenario file links to other files, which aid the simulator and scenario engine in calculating detailed routes and choose available equipment.

Summed up, a scenario has to fulfill the following parameters:

- lightweight
- easy to create
- easy to read
- independent reusability
- few to no redundancy

5.2.2 Existing Formats

Finding scenario formats in existing literature turned out to be more difficult than expected. The majority of papers deals with the creation of scenarios itself. For example [18] provides a general discussion on testing methods for the evaluation of autonomous vehicles. They come to the conclusion that evaluating the performance of autonomous vehicles is mostly a black box approach, due to the complexity of these systems. Because of that, they try to identify test cases for the performance boundaries of the system.

Other papers like [20] deal with the idea to provide open naturalistic driving scenario libraries. To get scenarios, the team behind the paper analyzed crash databases, as well as existing AD databases in various formats. Providing an open database in a consistent format is a good idea to aid smaller AD

development groups with testing. However, the overall format is not easily usable with a scenario interpreter.

Table 5.2 provides an overview of the found scenario formats with suitable requirements. The symbols used for rating are (-) for poor fulfillment of a requirement and (+) for good fulfillment. A more detailed analysis is provided below.

Scenario Format	LW	Create	Read	Reuse	Unique
Script	+	+	+	-	+
OpenScenario	-	+	+	+	+
SDFormat	-	-	+	+	+
MBSS	-	+	-	+	+

Table 5.2: Scenario formats with respect to suitability

Script

The easiest form to create a scenario is a script. Since version 0.9.1, Carla [30] provides a scenario loading module which takes and executes Python files. Each Python file contains the logic of the scenario. It spawns the vehicles, lets them drive a specific route and finally removes them at the end. While the script is easy to create and easy to read, it is not reusable. Even after an update of Carla, it might happen that specific code lines have to be changed in multiple files.

OpenScenario

The OpenScenario format [37] is the logic consequence of the OpenDRIVE format [36]. While OpenDRIVE files describe road networks, OpenScenario files describe what happens on these roads. Vires states on their homepage that the OpenScenario format has been designed together with influential business partners. They also state that they try to make OpenScenario an industry standard, as soon as it is released.

While the format itself is rather complex, its Extensible Markup Language (XML) background makes it easy to parse. As soon as Editors for this format are provided it should be easy to create these files. Even without editor, one can easily read the storybook provided. Additionally the format grants the ability to reuse huge parts of scenario elements like maneuvers. These parts can be stored in external library files.

SDFormat

SDFormat [53] is an XML file format to describe environments for robot simulators. It is the main format to describe environments and robots in ROS. With some effort it can also be used to describe scenarios. However, due to the complexity and the different scope, it is not the best format to describe scenarios.

Model Based Scenario Specification (MBSS)

An interesting format is the model based scenario specification (MBSS) discussed in [21]. The authors analyzed the different parameters necessary for a scenario and derived a domain model from it. With the aid of a database, one could create scenarios. This concept seems to be a good solution for huge scenario databases, as redundancy is kept on a minimum. However, the complexity of the system exceeds the scope of the project.

5.2.3 Decision

The decision finally fell on OpenScenario. The main reason for that is future expectation. As the standardization organization ASAM adopted OpenDRIVE and announced to accept OpenScenario as well, the future usage of this format is very likely [54]. The other mentioned possible scenario types are not suitable. A script is to inflexible, similar to SDFormat and the MBSS far to complex.

6 Prerequisites

For the whole setup procedure, the software repo [55] of this master thesis also contains a script (InstallAll.sh) which installs and explains all required components. Nevertheless its important to point out some tricky parts of the software stack.

6.1 Ubuntu

Carla [30] is developed for Linux distributions like Ubuntu but can also be built for Windows. For the implementation described in Chapter 8, other software tools are needed which are only available for Linux based systems. Due to the recommendations of the Carla developers and the developers of the additionally needed software stacks, the decision finally fell on Ubuntu 16.04 LTS [56].

Depending on the required performance, it is advisable to run Ubuntu in a native environment and use a dedicated graphics card. Other than that, a basic installation is sufficient.

6.2 Robot Operating System

ROS is a software environment administered by the Open Source Robotics Foundation [8], as already mentioned in Chapter 2. ROS is very widespread in prototyping robotics. For a quick and good structured overview of ROS, [9] is a good start. For this thesis the used ROS version is ROS Kinetic Kame.

6.3 Drive By Wire Kit

In Chapter 3 it is already stated that the final software should support specific drive-by-wire (DBW) messages in ROS, provided by Dataspeed. [57].

6.4 Carla Simulator

Carla [30] is one of the main pillars for the working result of this thesis. For the final implementation, version 0.9.3 is used together with version 4.21 of the Unreal Engine. At the end of this thesis, the Carla development team started to provide frequent and regularly updates to their software. As pointed out later in Chapter 7, the architecture of the software created within this thesis is modular to support future upgrades. In other words, as soon as the Carla simulator itself supports a feature, this feature can be deactivated in the provided software, to use more compatible and community maintained code.

6.5 Miscellaneous Python Libraries

The software created in Chapter 8 needs two important packages which are: (i) the prctl (python-prctl) library, to set meaningful thread names in process viewers like top and (ii) the xmlschema library to validate and parse XML files. These two packages play a key role during implementation and are therefore emphasized here.

7 Concept

To recap, the main pillars for the software of this thesis are: (i) a 3D simulation engine capable to provide physics and graphics close to realism, (ii) a light-weight scenario format capable to describe complex formats and (iii) a software loading and playing scenarios on the simulator while providing an interface between simulator and the AD software under test.

In the previous Chapter 5.1.3 the Carla simulator was found to provide the 3D simulation engine. The advantages of Carla are its origin from a research project and its ongoing development. At the time writing it can also be said that during the development phase of this thesis, the Carla team got on a similar path concerning scenario handling.

The scenario format, chosen in Chapter 5.2.3, is OpenScenario. The light-weight XML format has the advantage that for a proof-of-concept software, only parts of the XML file have to be parsed.

With the support of the first two pillars, the concept of the software can now be laid out. The software, shown in Figure 7.1 of Chapter 7.1 is called Scenario Loader. It will assist the communication between the AD software and the simulation software. It will also make use of OpenScenario files, which are the screen play for the simulator.

Also shown in Figure 7.1 is the Carla development block. The introductory Chapter 7.1 will continue with a brief overview in reference to this Figure. The Chapters 7.2 and 7.6 will then go into more detail on the semantics and the behavior.

Finally, Chapter 7.7 will present four different scenarios which shall be supported by the final software. These scenarios are also designed to support agile development of the software.

7.1 General Overview

Right from the beginning the software was designed to be modular and flexible. During the development phase it turned out to be a good decision as the Carla team started to move in a similar direction concerning scenario aided testing. The Carla development block, shown in Figure 7.1, indicates that some features have been removed from the Scenario Loader as they were introduced as a Carla feature. The first one was the ROS support of sensor messages.

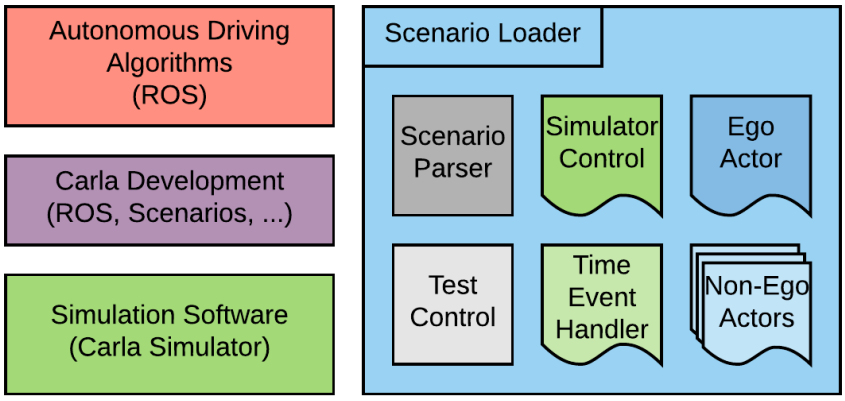

Figure 7.1: Scenario Loader Concept

The concept of the Scenario Loader relies on basic game theory. Games have a description of the scene. This is on the one hand the map, and on the other hand the actors. The configuration of this scene is provided by the scenario file, which is handled by the Scenario Parser block. The Scenario Parser is in charge to analyze a given scenario and provide the information.

Next the Test Control takes over, which handles all actors of the scene. There are the Simulator Control, which is the game master — setting the scene in the simulator. Meaning the Simulator Control loads the map and sets weather as well as the time of day. After the Simulator Control has set the scene, the Test Control activates the main actors. These are on the one hand

the ego vehicle as Ego Actor and on the other hand, all non-ego vehicles as Non-Ego Actors.

When the scene is set, the Test Control activates the Timed Event Handler, which synchronizes all actors and starts to step through the scenario, time stamp by time stamp, meaning simulator step by simulator step.

7.2 Scenario Parser Details

Coming from a general perspective, the Scenario Parser can be described with the aid of Figure 7.2. The Scenario Parser is either prepared for a specific scenario format, or takes a scenario format description file. With the knowledge about the scenario format, the Scenario Parser is able to validate the provided scenario file and parse the content. From the parsed content, the Scenario Parser extracts the scenario screenplay information and distributes it to the scenario controlling entities.

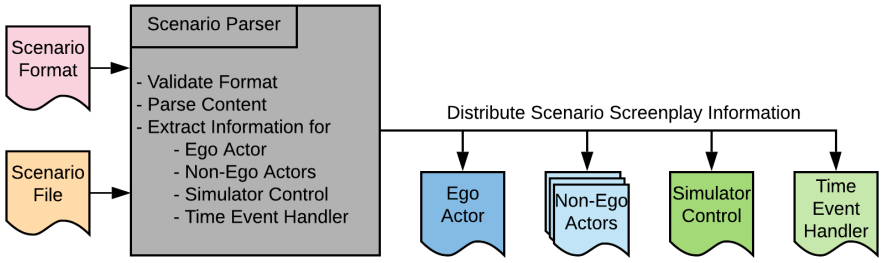

Figure 7.2: Scenario Parser Concept

7.3 Simulator Control Details

Controlling the simulation environment is the main task of the Simulator Control as shown in Figure 7.3. During the initialization phase, the Simulator Control is fed with information on how to connect to the simulation server.

In the same process the Scenario Parser provides the Simulator Control with environment setup information. This information consists of (i) the basic environment initialization — like the map, but also weather and time of day — as well as (ii) the event triggered environment changes in form of state events. State events are events triggered by the change of the server state. For example this can be *on startup* or *on scenario end*.

During run-time the simulator control may receive events from the Time Event Handler or one of the actors. For example: the scenario requires a time of day change after two minutes of simulation time, or a weather change when the ego vehicle arrives at a specific location.

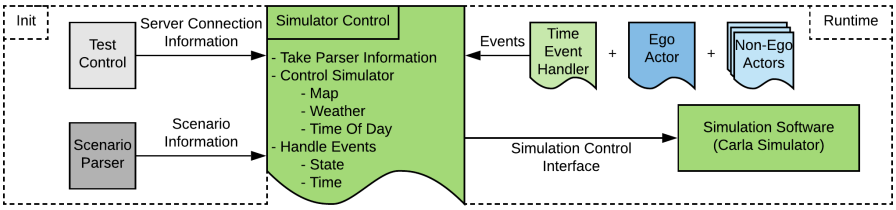

Figure 7.3: Simulator Control Concept

7.4 Test Control Details

The Test Control provides an interface to control the test environment as shown in Figure 7.4. It wraps the initialization and the cleanup process, as well as starting and stopping of the threads necessary for the simulation.

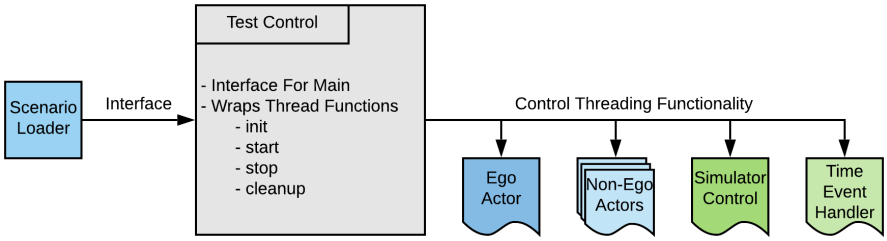

Figure 7.4: Test Control Concept

7.5 Time Event Handler Details

Time events are events at specific simulation times. At initialization the Scenario Parser fills the Time Event Handler with time events, as shown in Figure 7.5. Additionally the Test Control sets the reference time. The reference time is important to save computation time, as the simulation will not be reloaded. The approach is thereby similar to the mechanism discussed in Chapter 4.6.

Then, during run-time, the Time Event Handler gets an update of the simulation time at every tick of the simulator. The update is provided by the Simulator Control. At each tick, the Time Event Handler checks the time events. If an event is triggered, the event is sent either to the Simulator Control, or to one of the Actors.

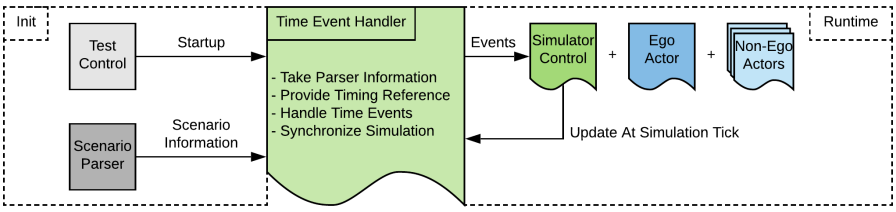

Figure 7.5: Time Event Handler Concept

7.6 Actor Details

Ego Actors and Non-Ego Actors have the same initialization behavior, as shown in Figure 7.6. Both get their scenario related information from the Scenario Parser and are triggered by the Test Control. During run-time they also share similar behavior as each of them handles their own set of entity events. If an entity event is triggered it may either be executed within the current actor, or it is sent to the corresponding actor. The check of the events, as well as the calculation of the trajectory is done after the reception of a simulation tick.

However, before sending a vehicle control message via the Simulation Control, the behavior differs: the Non-Ego Actor calculates the current pose with the aid of the scenario description, while the Ego Actor loops to the SIL-tested AD-Algorithms.

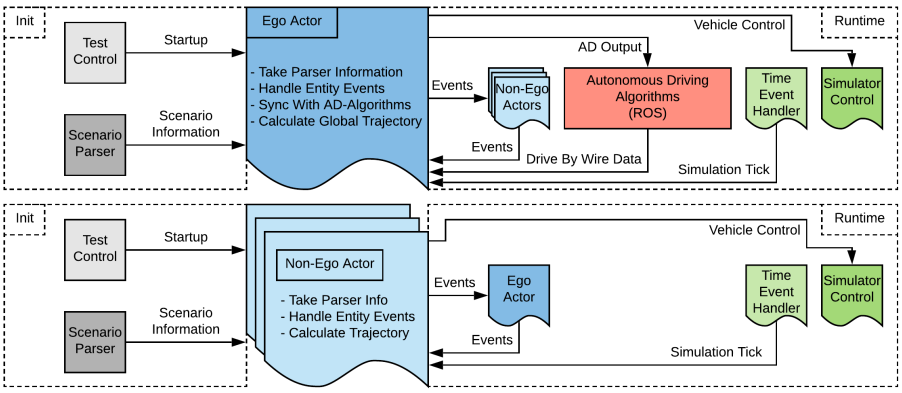

Figure 7.6: Actor Concept

7.7 Conceptional Scenarios

Before the implementation step in Chapter 8 it was important to define some basic scenarios which should be supported. Therefore shown in Figure 7.7 are four scenarios which represent the fundamental support of a scenario

loading engine. The advantage of these scenarios is also the potential for iterative implementation. By targeting the support of one specific scenario at a time, the software can be developed in an agile manner.

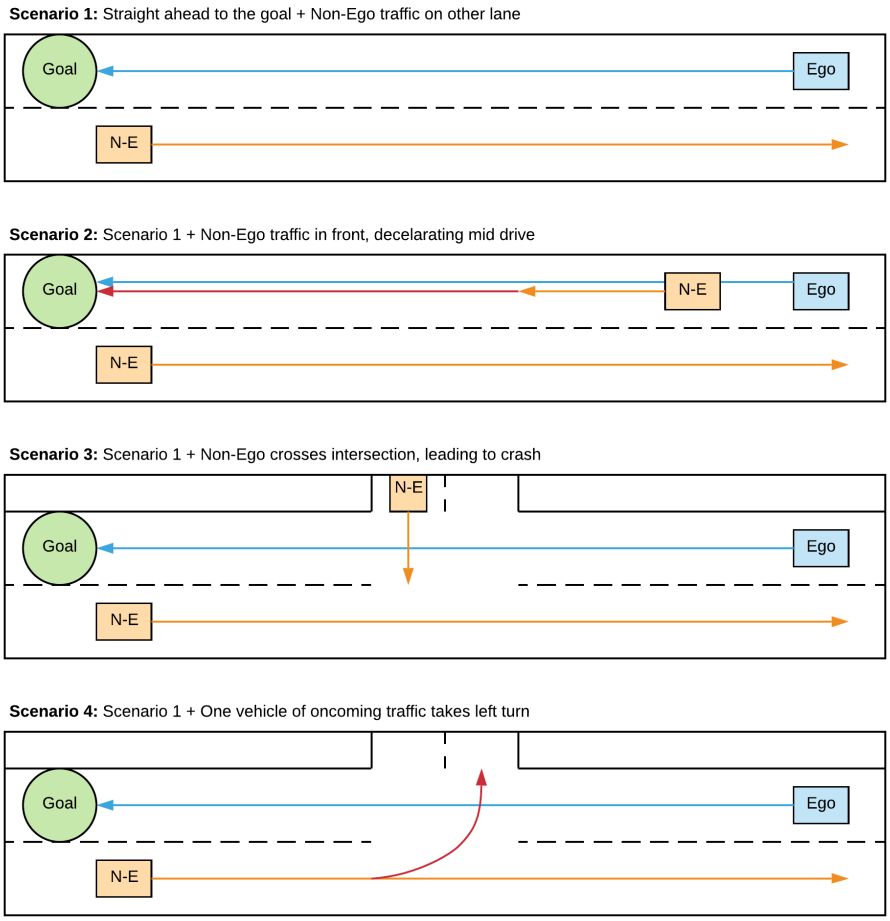

Figure 7.7: Four Fundamental Scenarios

To briefly describe the scenarios shown in Figure 7.7 one might start with scenario one. This scenario features a simple ego vehicle which should reach a goal. In the next step, the non-ego vehicle on the oncoming lane can be implemented. Reaching a step further to scenario two, one can see an additional non-ego vehicle which decelarates at a specific point.

Scenarios three and four then feature an intersection. In scenario three a vehicle from the right crosses when the ego vehicle reaches a specific position. Without intervention from the AD algorithms a crash should be guaranteed. A similar approach is taken by scenario four. However, the non-ego vehicle takes a left turn at the intersection to provoke a crash.

8 Implementation Details

The flexible and modular design presented in Chapter 7 is the foundation for the software design presented within the next Chapters. The flexibility is also necessary for agile development as (i) the Carla simulator is still in a young development state, (ii) the OpenScenario standard is too complex to be fully implemented from scratch and as (iii) the Scenario Loader may be further developed by others in the future.

To get a better overview on the implementation, Chapter 8.1 will start with a more detailed outline of the concept. Chapter 8.2 will then provide a brief overview by presenting the code structure as a class diagram. Finally, Chapters 8.3-8.5 will provide an in-depth overview of explicit parts of the code.

8.1 General Overview

Before further details are explained, the following Chapters present the principles of the designed software, while referring to the concept presented in Chapter 7. Thereby Chapter 8.1.1 recaps the modules of the Scenario Loader software itself. Next Chapter 8.1.2 continues with a more detailed explanation of the communication between the Scenario Loader software and the AD software, as well as the simulation environment. Closing by Chapter 8.1.3 which presents a sequence diagram of a single run through.

8.1.1 Basic Modules

Looking at the Scenario Loader, one can distinguish the main method and six different modules, as shown in Figure 8.1. These modules are used to process the key functionality of the Scenario Loader as Chapter 7 already pointed out. While the main method triggers start up and console procedures. The following paragraphs should provide a quick overview before the next Chapters go into more detailed implemented behavior.

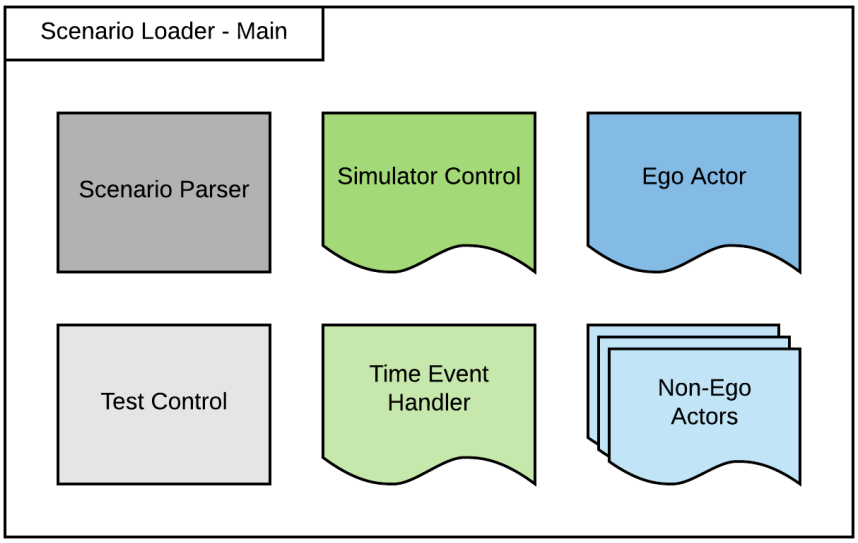

Figure 8.1: Scenario Loader: Basic Overview

Scenario Loader Main handles the command line parameters and the scenario file lookup. It also uses the modules scenario parser and test control to process each scenario.

Scenario Parser takes a scenario description as input and creates the actor classes, as well as the event management system as specified by the file.

Test Control manages the test of a single scenario. It triggers the start-up procedure of the threads created by the scenario parser. After the initialization is complete, it triggers the run procedure. The main thread is then waiting on a signal flag, which can be signaled by either one of the actors, or a keyboard thread. After wake-up, test control takes care of the scenario shut-down procedure.

Simulator Control is the controlling thread for the simulator. In the current version it is able to start and stop the simulator. But it is prepared for future development, as functionality is designed to take initialization parameters. These parameters can be the name of the map, or weather configurations.

Time Event Handler has two functionalities: (i) it is the central synchronization of all actors and (ii) it handles timing events. Ideally a simulator has a synchronization mode. Meaning the simulator does one calculation step at a time. In other words, at a so called tick, the simulator provides output, takes input and continues after a signal. Timing events are in close connection to the synchronization ability. A timing event can be triggered at specific simulation times.

Ego Actor and Non-Ego Actors are basically the same. Their purpose is to control the vehicles in the simulator. The difference is simply that the ego actor uses the commands coming from the AD implementation, instead of directly following a route provided by the scenario description.

8.1.2 Communication System

This Chapter provides an overview of the communication system, meaning the complete loop starting from the AD software implemented in ROS via the Scenario Loader to the Carla Server and back. The following paragraphs will reference Figure 8.2 which provides a graphical representation.

To get a functional loop, the ROS AD implementation of the Ford Mondeo and the Carla server should already be running. As the previous Chapter

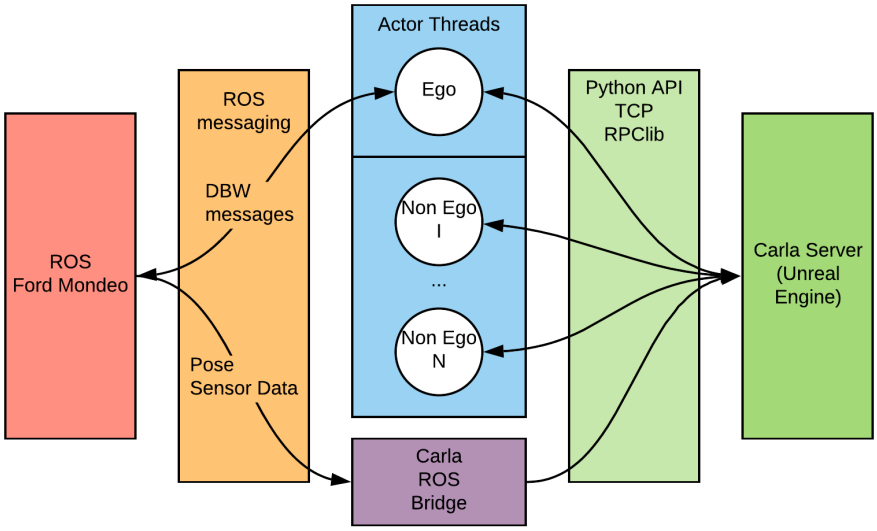

Figure 8.2: Scenario Loader: Communication System Overview

8.1.1 indicated, the Scenario Loader software initializes the actor threads and the simulator control connects to the Carla server.

After start-up, the actor threads go into a sleep mode and wait for a wake-up signal. Each server tick is a calculation step of the server. When the server tick callback is triggered, the Time Event Handler wakes up all actor threads. The reason for this wait and signal implementation is the synchronized mode coming in one of the future Carla releases.

After wake-up, the actor threads query their current position. Now the behavior of the ego and non-ego actors differ. The ego thread sends its current position and other DBW commands as ROS messages to the ROS implementation of the Ford Mondeo. The non-ego threads however calculate their actions for their own.

The DBW message sent by the ego actor thread is then processed by the ROS Ford Mondeo AD software. The software also gets sensor and pose information via the Carla–ROS bridge. At this point it is important to mention that in previous versions of the Scenario Loader software the ego

vehicle thread provided non-ego pose and sensor information itself. But due to ongoing development, the ROS bridge of Carla made additional processing obsolete.

Generally the ROS Ford Mondeo software does its calculations according to the ROS time stamp, which is the current simulation step of the simulator. Therefore the ROS part is always synchronized. So now the calculations made by the ROS software can be read by the ego actor thread which sends them to the Carla server and goes to sleep to wait for another wake-up.

Simultaneously the non-ego actors use their routing information — which they get from their events — to calculate the position and speed matching the current simulation step. After the calculation each non-ego actor sends the new positions of their car to the simulator and goes to sleep, similar to the ego actor.

8.1.3 Run Through

The following paragraphs provide an overview of the basic internal functionality of the Scenario Loader from time perspective. The sequence diagram shown in Figure 8.3 will be the main reference.

On Scenario Loader program launch, the command line parameters are analyzed and the scenario files are extracted. Next, the main method goes into a loop where each step handles one scenario file.

In each loop the scenario file is sent to the scenario parser. The scenario parser checks the file integrity and parses the contents. The gained information is used to initialize the simulator control, actor thread and the time event handler.

After the important threads have been created, the main method of the Scenario Loader utilizes the test control module to set-up and start-up the previously created threads.

As one can see in Figure 8.3, each thread does the job it was designed for. After each shown loop, they go to sleep, until they are woken up by the time event handler.

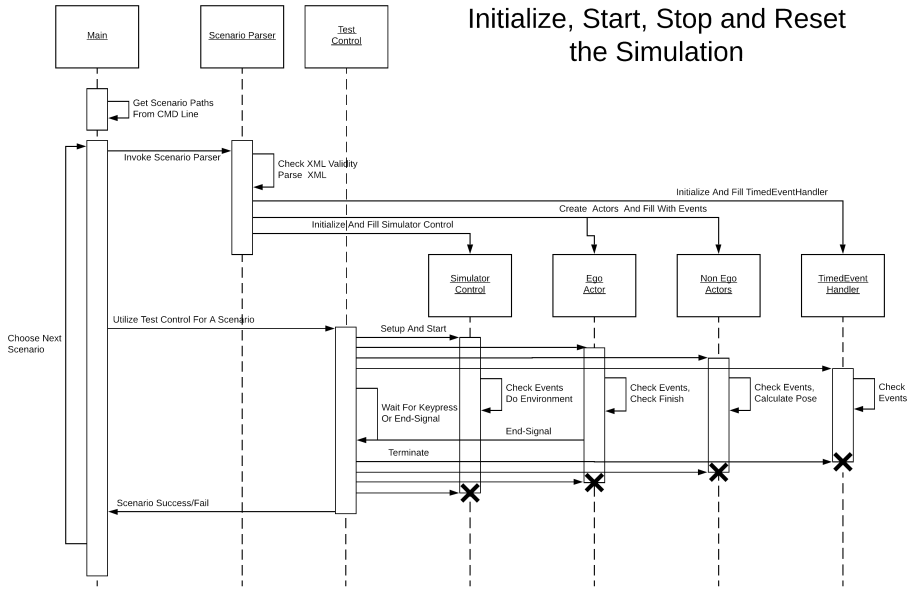

Figure 8.3: Scenario Loader: Sequence Diagram

Termination of the inner loop can either be triggered by the keyboard thread spawned by the test control thread, or if an actor thread terminates. An actor thread may terminate on error, or if a collision is detected.

When the end signal is signaled, the main thread is woken up in the test control module. The main thread then sends end signals to each thread and joins them. After successful termination the flags in the actors are analyzed. If none of them indicate a failed test, the test is a success.

In any case, the test control module also resets respectively deletes the threads and returns. Then the next scenario can be handled the same way. Except if the keyboard press — which stopped the last scenario execution — was a termination signal. Or if there is no scenario left. In these cases the program simply terminates.

8.2 Code Structure

The implementation of the Scenario Loader follows the structure proposed in Chapter 7. Figure 8.4 shows the main functionality of the Scenario Loader, which are (i) the scenario parser, (ii) the test control, (iii) the simulator control and (iv) the time event handler. The Scenario Loader software was also designed to be modular and as flexible as possible. A classical approach to achieve modularity is to use inheritance, which leads to behavior as the *Adapter* design pattern suggests.

Looking at the class diagram shown in Figure 8.4, one can see that the OpenScenarioParser implementation has the ScenarioParser class as base class. The ScenarioParser class — similar to the SimulatorControl class — provides an interface for other more specific implementations. Speaking in terms of code, the user can change the initialized class, because the underlying implementation is always controlled via the base class.

With the *Adapter* pattern it is also possible to be open for future scenario format changes. One idea leading to this pattern was the uncertain stage of the OpenScenario 0.9.1 format. Vires, the company behind OpenScenario, was announcing an upgrade to version 1.0 since the end of August. The idea was to support the old version 0.9.1 and the new version 1.0 with very few changes to the code. This could have been possible, with two separate OpenScenarioParser implementations.

Going a step further, the implemented actors are also designed as an *Adapter* pattern, as shown in Figure 8.5. The Actor class however, also implements the *Observer* pattern. As proposed in Chapter 7, the time event handler is in charge of the synchronization steps. As mentioned in Chapter 8.1.3, the actor threads go to sleep after they sent their new commands to the server. When every actoring thread is waiting, the TimedEventHandler class triggers the next synchronized simulation time step. After updating the simulation time, the TimedEventHandler triggers the *Observer* update method of every subscribed actor.

To ensure the implementation of patterns, the metaclass system of Python3.x has been used. Metaclasses basically define additional functionality of a class. One can use it for logging or registering classes at creation time. But

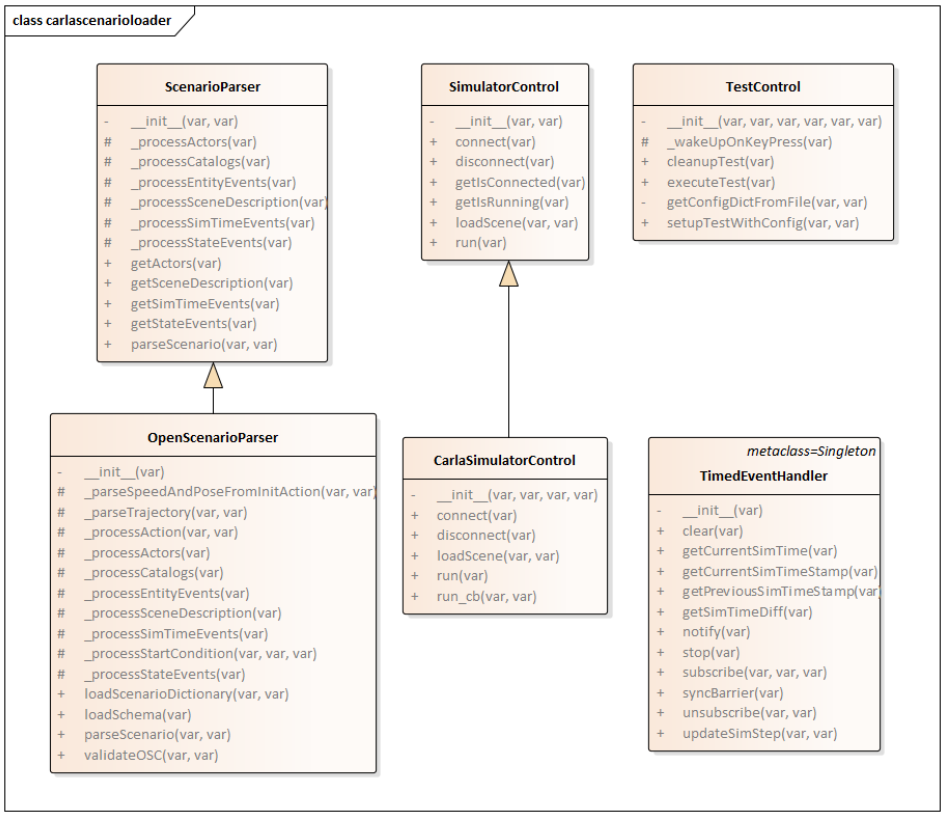

Figure 8.4: Scenario Loader: Class Diagram

in case of the Scenario Loader, it is used to define singletons and abstract base classes to ensure specific implementation behavior.

Before stepping further into the details of the Scenario Loader implementation, it is important to point out that also the events are created as inheritance models. Per definition, an event has a condition which triggers the event and an action which is the result of the event. Combining the knowledge of related research from Chapter 4 and the far reaching specification of OpenScenario 0.9.1, one can pin down three different types of events. These events are: (i) events triggered by entities, (ii) events triggered by the simulation time and (iii) events triggered by the state of the simulator (i.e. start-up, weather change, ...)

The Scenario Loader software is prepared to take these events, but other events than EntityEvents will raise *NotImplemented* errors. The all over policy of the implementation is to print warnings to the console if parts of the implementations are skipped, but not directly necessary and to raise concrete errors, if program execution can not be guaranteed due to missing implementations. As also Chapter 9 and 10 will point out, the time needed to fully implement a complete scenario standard and its effects on the simulator is a task for a team of developers.

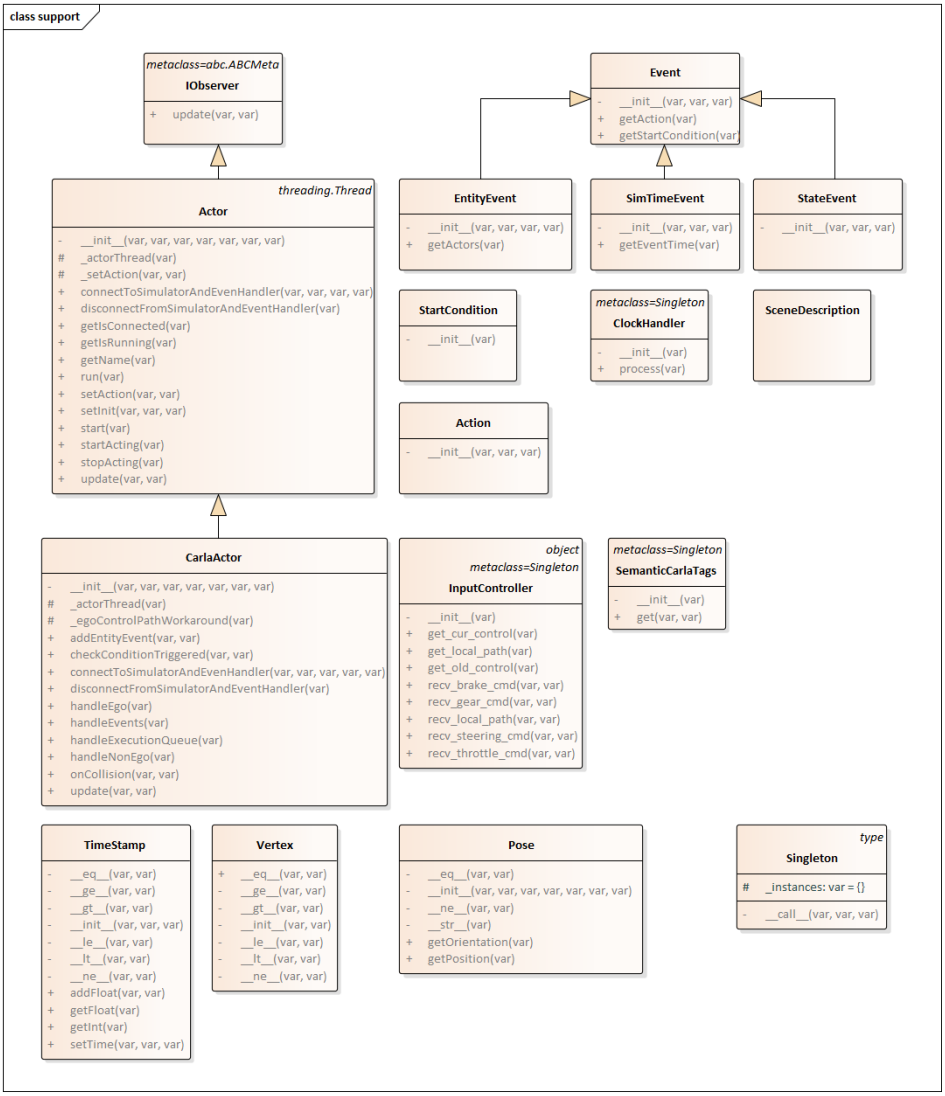

Figure 8.5: Scenario Loader Support: Class Diagram

8.3 Scenario Parsing

The used OpenScenario standard in version 0.9.1 is a sophisticated XML standard designed by the company Vires. As mentioned in Chapter 5.1.2, Vires also published the frequently used OpenDrive standard. Similar to the creation of OpenDrive, Vires discusses and reviews the OpenScenario standard with industrial partners as can be gathered from the publications at [37].

To handle the XML-file, the *XMLSchema* Python library is used. *XMLSchema* is fed with the OpenScenario XML Schema Definition (XSD)-file in version 0.9.1 to initialize the parser. The parser is then able to validate a provided OpenScenario file. After validation the OpenScenario XML file is parsed by the *XMLSchema* library which results in a nested Python dictionary.

Due to the complexity of the dictionary, respectively the OpenScenario standard itself, the best approach to implement functionality is to select a specific set of features and iteratively implement them. The most simplistic start is to get the actors of the scenario from the file. After parsing and setting their initial parameters, more complex elements can be parsed, like events and trajectories. With every additional parsed element, the rest of the Scenario Loader functionality has to be improved too.

In case of unsupported functionality, the same policy applies as mentioned in Chapter 8.2. Critical components, which directly influence existing behavior, will raise *NotImplemented* exceptions. Missing implementations for less critical components result in a warning on the console log.

8.4 Event Handling

Independent of the type, an event always consists of an action and a triggering condition as shown in Figure 8.6. The starting condition can either be a pythonic *None* or a set of conditions. For example: some acting vehicle has to be within a specific distance to another acting car. Besides that, the three different types of events can be distinguished by their purpose:

EntityEvents are triggered by one or multiple entities. A list in the event contains all actors required to trigger the start condition. If one actor triggers its own start condition, it has to check with all other actors if they also fulfilled their part of the start condition. In other words, each actor has a list of events to check every simulation step.

SimTimeEvents are triggered by the simulation time. Therefore the starting condition is normally a pythonic *None*. But conforming to the OpenScenario definition it is possible to hit a specific simulation time and afterwards deal with the start condition. In this case, the SimTimeEvent is converted into an EntityEvent with the aid of the StartCondition.

StateEvents are triggered by the state of the simulator. If the simulator, respectively the scenario is starting, stopping, or reaching another defined state, the event is triggered. Similar to SimTimeEvents, StateEvents can fall back to SimTimeEvents or EntitiyEvents after they are triggered.

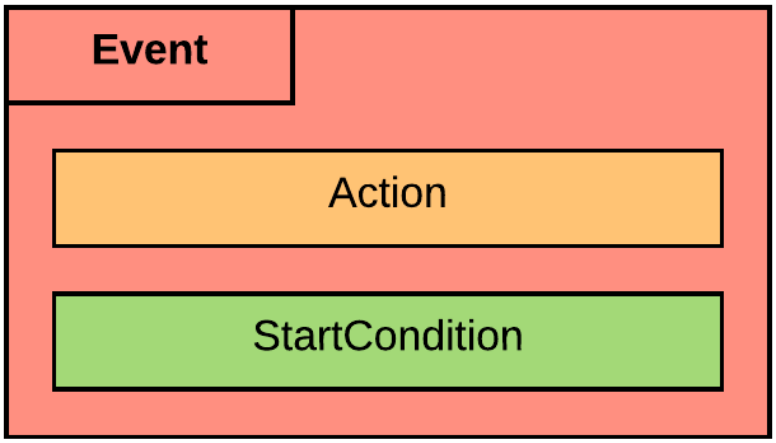

Figure 8.6: Scenario Loader: Event Layout

The event handling itself is implemented in reduced form. This is due to several reseaons: (i) the SimTimeEvents are not useful enough for the application cases of the VIF, (ii) the SimTimeEvents are not applicable as long as the synchronized mode is missing for Carla, (iii) the StateEvents do currently not add any value to the used short scenarios.

Since only EntityEvents are used in the current software state, the potential conversion of SimTimeEvents and StateEvents are not implemented. However, the design of the event handling architecture of the actors can easily deal with that in the future, as (i) the actors already manage their own events, (ii) the TimedEventHandler is ready to deal with SimTimeEvents and as (iii) the SimulatorControl is designed to be equipped with functionality to handle StateEvents.

Each actor has (i) an event queue with EntityEvents, (ii) an action queue with Actions and (iii) an execution queue with poses mapped to time stamps as shown in Figure 8.7. The idea of queues is that they can be filled from the right side and popped from the left. Therefore it is possible that at each simulation step, each actor steps from the left to the right through its event queue and checks if the StartCondition is fulfilled. If it is fulfilled, the actor takes the reference to the affected actor and calls its setAction method to transmit the action contained in the event.

Depending on the overwrite parameter of the action, the setAction method either appends the action to the right of the action queue or overwrites the action queue with the provided action. In any case, the setting of the action queue itself is synchronize, but the actor threads are unsynchronized with each other. Therefore the affected actor may be already past the check of the action queue described below.

After dealing with its event queue, the actor checks if the action queue is empty. If the action queue contains elements, the actor takes the actions to overwrite or extend its execution queue. The decision depends on another overwrite flag. For example: a new trajectory can either be put at the back of the current execution queue, or replace the whole queue.

The execution queue is the last element shown in Figure 8.7. The execution queue contains the trajectory of the actor and is calculated from the actions, as Chapter 8.5 will elucidate.

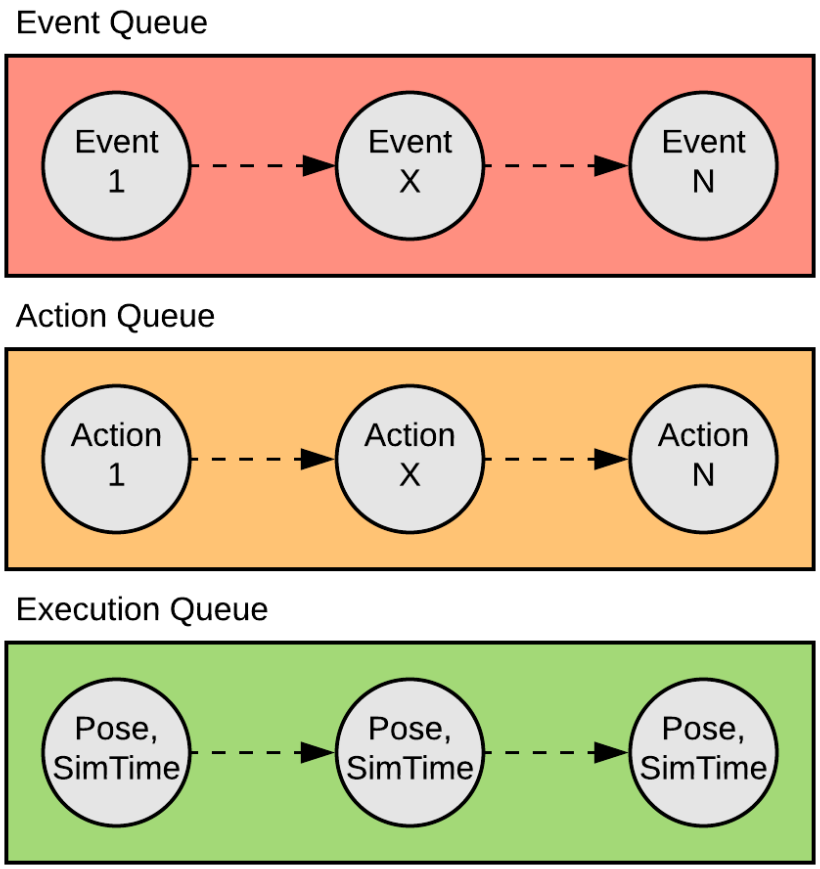

Figure 8.7: Scenario Loader: Event Queues

8.5 Actor Routing

Important to mention here is the fact that the trajectory is not fully stored in the scenario file. Only changes in the trajectory are stored. Each actor takes the action points (explained in Chapter 8.4) and calculates its own trajectory. The resulting trajectory is stored as waypoints in the execution queue.

A waypoint has a pose, a velocity and a time stamp. Currently the distance between two poses is ten centimeters. If no action is provided to calculate a new trajectory, the current desired speed is taken to calculate 100 waypoints straight ahead. If the desired speed is zero the execution queue is set pythonic *None*. The time stamp of a waypoint is the simulation time at which the pose is valid.

After the calculation of the trajectory, the behavior of the ego actor is different to the non-ego actors. While not implemented in the current development state due to time reasons, it is planned that the execution queue will be published as trajectory for the ROS AD implementation. The software under test is then able to follow the route prepared by the scenario.

The non-ego actors however, are different. As mentioned above and shown in Figure 8.8, the execution queue has a reference to the simulation time. Each actor uses the sorted queue of waypoints to jump to the specified pose. Thereby the next pose is popped from the left of the execution queue. Then the current time stamp of the simulator is compared to the time stamp of the popped pose and to the time stamp of the left most element in the execution queue. If the current time stamp is somewhere inbetween, the new pose is interpolated between the points. Else the next pose is popped from the execution queue. If the queue is empty, the previous step — where the action queue was checked — is repeated to fill the execution queue again. The execution queue stays empty, if the desired speed from the last action is zero. In this case, the actor goes into standstill

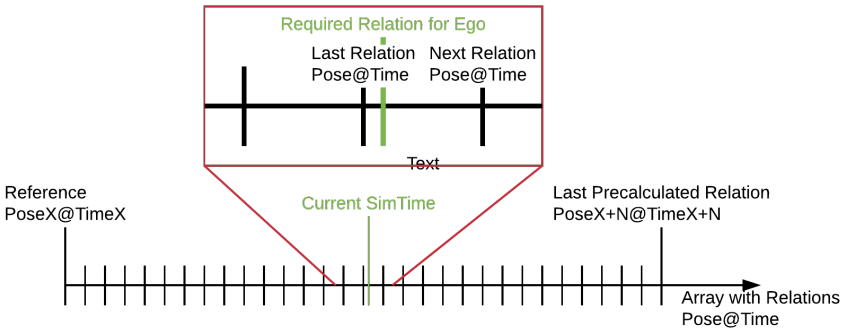

Figure 8.8: Execution Queue: Actor Routing

9 Evaluation

The evaluation of this thesis concentrates on three factors:

(i) a review of the used simulator and scenario format in Chapter 9.1, 9.3 and 9.4. Here the main external elements are checked for their suitability for the requirements in Chapter 3. The performance of the simulator is tested in Chapter 9.1 and compared to the principles of SIL testing as stated in Chapter 1. The scenario format is then reviewed by experts in Chapter 9.3 and analyzed with a view on industrial applicability in Chapter 9.4.

(ii) an analysis of the architecture and performance in the Chapters 9.2 and 9.5. These Chapters review the Scenario Loader software from a software architectural perspective and compare its features to the requirements in Chapter 1 and 3. While Chapter 9.2 concentrates on the simulation loop as a whole, Chapter 9.5 evaluates the performance of the different key components.

(iii) a look on the internals, consisting of scalability in Chapter 9.6. This Chapter sums up existing issues and brings them in perspective to scalability and future development. It also provides some ideas for software architectural improvements, which is a basis for the conclusions made in Chapter 10.

For the evaluation of the functionality and the performance, the set of four basic scenarios — defined in Chapter 7.7 and shown in Figure 7.7 — have been used. The four scenarios represent scenario groups. In other words, these basic scenarios were additionally tested in reduced form, for example without oncoming traffic.

The scenarios were modeled with OpenScenario 0.9.1, chosen in Chapter 5.2.3. The format itself is rather complex. A basic overview is shown in Figure 9.1, for a better understanding. In context of this thesis, the two most

important sections of the scenario format are the *Entities* element and the *Storyboard* element. The *Entities* element contains a description of all objects (vehicles and pedestrians). The current version of the *Scenario Loader* needs simply the definition of a vehicle to create an actor. Sub-elements like the bounding box or axles are not further parsed.

Within the *Storyboard* element, the *Init* element defines the initial position of the vehicles. The *Scenario Loader* uses that information to spawn the actors. The *Story* element contains the logic of the scenario. From top to bottom it consists of *Acts*, *Sequences*, *Maneuvers* and *Events*. *Events* themselves consist of *Actions* and *StartConditions*.

The running version of the *Scenario Loader* currently supports only one element of a kind, except for *Sequences*. Considering for example scenario four in Figure 7.7 the non-ego vehicle has (i) a *Sequence* for the first part as oncoming traffic, (ii) a *Sequence* for the turn and (iii) a *Sequence* for driving straight ahead after the turn. Taking the full logic of OpenScenario aside, the *Sequences* can be broken down to three different *Events*: (i) *Event* straight ahead oncoming, (ii) *Event* turn, (iii) *Event* straight ahead after turn.

The *StartCondition* of the first *Event* is triggered by the ego vehicle as it moves by a specific point (calculated via speed and distance before the scenario). The second and third *Event* are triggered by the non-ego vehicle itself. At first the non-ego vehicle reaches a specific position and drives the turn. And if the ego vehicle successfully prevents the collision, the non-ego vehicle also reaches the end-point of the turn and continues straight ahead after triggering the third *Event*.

This scenario could also be modeled in other ways, but specific guidelines are not yet published by the OpenScenario team.

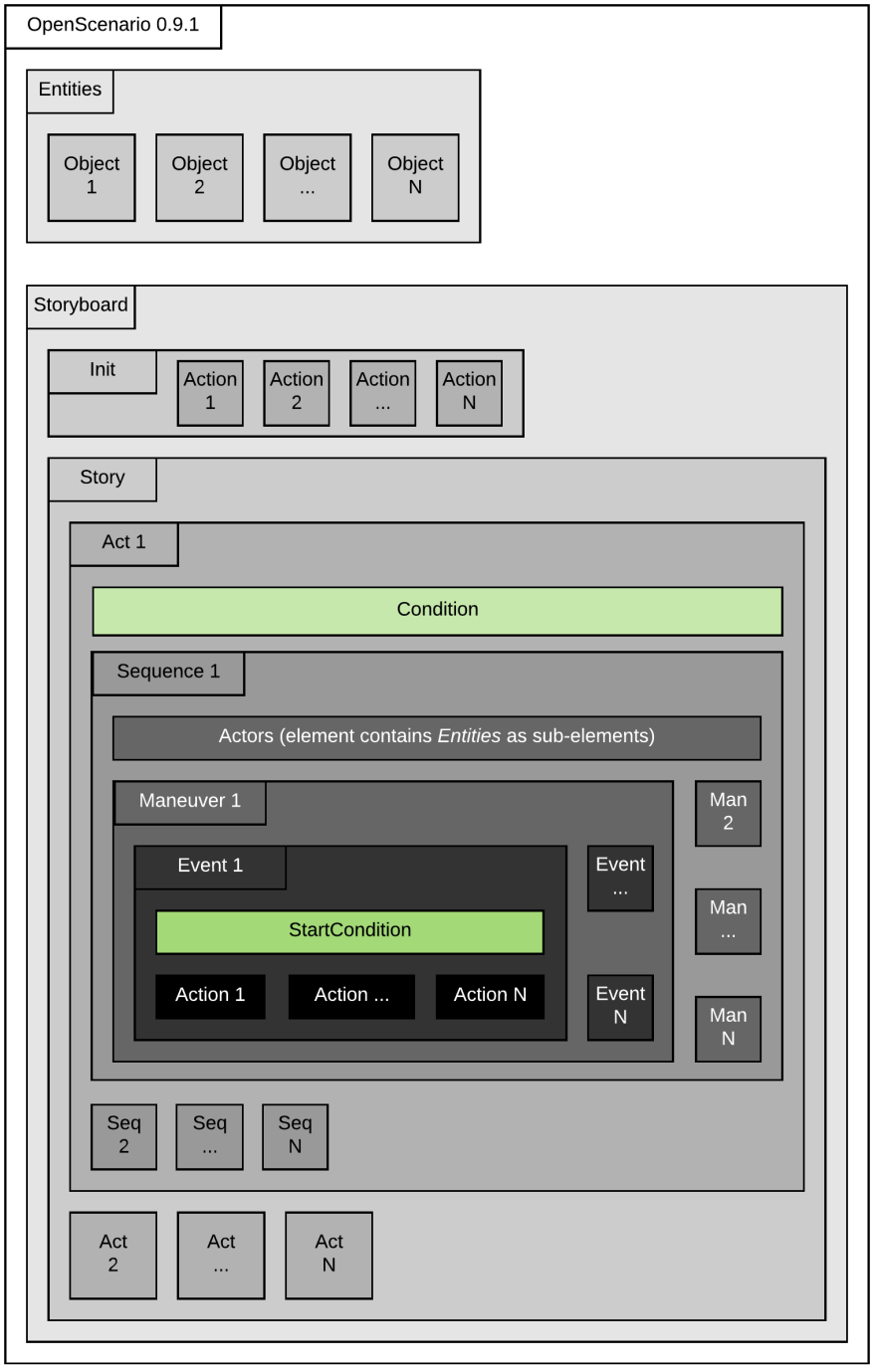

Figure 9.1: Basic Layout of OpenScenario 0.9.1

9.1 Carla Simulator

First of all it has to be pointed out that two different Carla versions were used during this thesis. In the beginning, the stable version 0.8.x provided a *Protocol Buffer* interface. However, the available features were limited, as there was no possibility to directly influence the pose of non ego vehicles or even pedestrians. Additionally the performance was critically limited due to the *Protocol Buffer* interface used in Python.

The other Carla version, 0.9.x implemented the remote procedure call (RPC) library into the communication stack. Although RPC was first criticized as performance bottleneck, it turned out that the Unreal Engine will most likely be the first performance limiter. The new interface also allowed to directly control non-ego vehicles and promised direct control over pedestrians. However, the new Carla versions lacked the synchronized mode, which later turned out to be necessary, as the Python implementation of the Scenario Loader got timing issues as outlined in Chapter 9.5.

In general the performance of Carla has a lot of potential, when compared to its competitors. Especially as the underlying Unreal Engine is a well developed game engine. However, as the previous paragraph stated, the synchronized mode is still missing in the current version 0.9.3. While the simulator provides the most basic requirements stated in Chapter 5.1.1, the missing synchronized mode has a strong influence on the reliability and replayability of the whole Scenario Loader loop. To be more specific: Carla does not guarantee a fixed time between each simulation step. Carla simply calculates the graphics and physics as fast as possible. Additionally the Scenario Loader software can sometimes not guarantee fast enough responses. The Carla engine and the actors of the Scenario Loader can therefore reach a highly asynchronous state. This in consequence may lead to serious deviations as quantified below.

To quantify the derivation, Scenario 2 and 3 — defined in Chapter 7.7 and shown in Figure 7.7 — were used in a reduced form. The tested scenarios contained the ego vehicle and one non-ego vehicle to provoke a crash. For both scenarios, the ego vehicle was defined to travel with 50km/h. The Scenario Loader software was run on two different machines. Machine 1 was the development machine which ran Ubuntu 16.04 LTS in a VirtualBox

on a Windows 10 host, with a Intel i7-8550U. Machine 2 was for comparison and ran native Ubuntu 16.04 LTS and had an Intel i7-7800X.

Scenario 2 was set up as shown in Figure 9.2. The ego vehicle triggers the start of the non-ego vehicle. The non-ego vehicle immediately drives 40km/h. When the ego-vehicle reaches the next trigger point, it immediately reduces its speed to 20km/h. The speed reduction will then cause a crash.

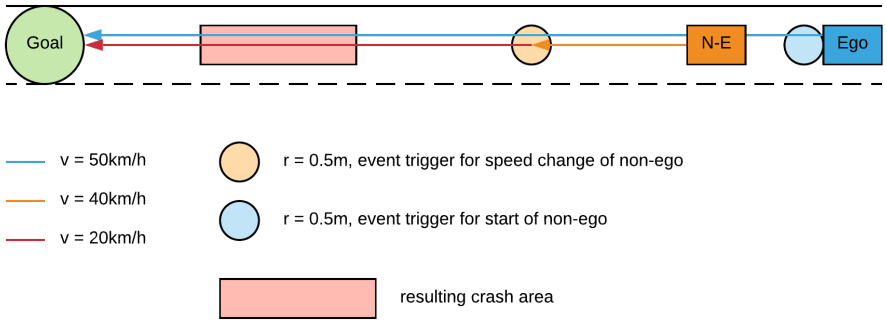

Figure 9.2: Deviation Quantification: Scenario 2 Setup

Scenario 3, shown in Figure 9.3 features a non-ego vehicle which is triggered by the ego vehicle at a specific point. The non-ego vehicle then drives straight ahead with 10km/h which results in a crash.

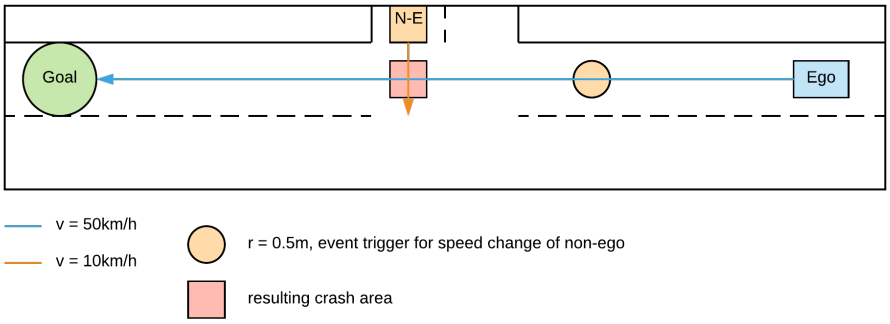

Figure 9.3: Deviation Quantification: Scenario 3 Setup

The results of the deviations, presented in Table 9.1, are based on 10 run-throughs on each Machine. The deviations in the table are thereby measured to the average center of the measurement set. The results show, the longer the scenario, the higher the error. Overall, the results showed that the asynchronous character creates great deviations if the scenario module is to slow. Additionally, the asynchronous nature of the simulation creates a negative feedback with the controller of the SIL environment, which leads to great variations in the crash area, even though the trigger points are quite accurate.

Furthermore, as the deviations are also highly influenced by the computational load of the machine — which was observed during development — it should be stated that no commercial simulator runs in an asynchronous mode. At this point it should also be pointed out that deviations in physics simulations are normal, especially if controllers are used. Deviations depending on the type and complexity of the simulation, as well as the tested scenario can in most cases be inquired from the simulation vendor.

Deviation Type	Avg M1	Range M1	Avg M2	Range M2
S2: Ego Event Trigger	103	52–168	47	21–83
S2: Non-Ego Event Trigger	107	63–201	51	27–85
S2: Crash Area	10700	6900–17400	6900	4300–9900
S3: Ego Event Trigger	17	9–31	9	5–19
S3: Crash Area	3400	2500–5200	2100	1200–4400

Table 9.1: Deviations from the average center of the set (values in mm)

Summarizing the deviation issue, one can estimate that a deviation of 2-5 meters, as in scenario 3, can be the difference between a hit or miss of two actors. In other words, it could lead to an accidental fail or success of the tested scenario. Figure 9.4 shows for example that scenario 3 could be a frontal or side impact with that kind of deviation.

Moving on to scalability, the performance of Carla is limited to the performance of the computer running the simulation server. When using only pose information from the simulation, the server runs with solid 50-70 frames per second (FPS). The used hardware is an NVIDIA GTX 1060 6GB, 32GB RAM and an Intel i7-7800X. With this setup it is no problem to connect three

Figure 9.4: Carla Simulator: Crash Deviation (f.l.t.r.: Frontal and Side Impact)

Scenario Loader clients for tests on one map. It is also possible to control 10-20 cars without major frame drops below 40 FPS. However, the FPS drop significantly when more than 3 cameras and a complex LIDAR sensor are used on the mentioned hardware. But as with other simulation platforms, the used hardware is the main factor for possible sensor computations.

9.2 Simulation Loop

Pointed out in Chapter 3 the Scenario Loader software should be able (i) to load and play a scenario, (ii) to synchronize the AD implementation with the simulator and (iii) to report if a scenario failed or succeeded. While it can be said that all of these requirements are fulfilled, the next few paragraphs go into detail on the level of support:

Scenario loading and playing is implemented iteratively. Basic functionality as vehicle definition, spawn location and simple routes can be read from the file. If something is not yet supported, a console message is printed with a warning or error message. The decision for the message type is thereby made by the criticality of the not yet parsable information. Both elements, loading as well as playing, are dependent on each other. If a feature to load is implemented, it also has to be implemented on the playing side and vice versa. The Scenario Loader is also able to handle multiple scenarios, as each scenario file is handled for its own. The Scenario Loader takes either single files or multiple folders as parameters for the scenarios.

Scenario synchronization is the bridge between the ROS AD software and the simulator. The main problem here is the missing synchronization mode of Carla. The Scenario Loader software is already able to synchronize new incoming data from the server with the AD implementation. However, the server does not wait for a response and floods the client with new calculation steps, as soon as they are available.

Scenario reporting is implemented very light-weight. The scenario is defined to be a success if the ego-vehicle reaches a specific target point and stops there with zero velocity. A scenario can be failed, if the Carla server detects a crash with the ego-vehicle. One can already see that there are other possibilities for the scenario to fail. For example when the ego-vehicle takes too long to reach the target. Or when the ego-vehicle disobeys traffic rules. All of these features can easily be implemented as checks in the ego actor. However, completeness of scenario success checks was not the target of this thesis.

Summarizing the above statements, evaluation showed that the implemented logic works as expected. This is mainly due to the reason that the desired behavior was immediately tested after its implementation. However, although the basic logic of the simulation loop is working — as tested with the required scenarios defined in Chapter 7.7 and shown in Figure 7.7 — there are two important major issues: (i) the simulator has no synchronized mode and (ii) the simulator needs steering angles for the vehicles to provide realistic physics.

The missing synchronized mode not only leads to the deviations expained in Chapter 9.1. But the overall performance of the system suffers due to necessary synchronization overhead combined with a weakness of Python, as analyzed in Chapter 9.5. The physics problem, also described in Chapter 9.5 might not be an issue with a working synchronized mode. However the effect is still important to mention to support more accurate simulations.

9.3 OpenScenario Reviewed By Experts

To get a better impression of the practicability in everyday research and development, two experts of the VIF have been asked to review the format and state their opinion:

Bernhard Hillbrand uses the scenario generator of Vires VTD to create test scenarios for internal verification scenarios. Overall he pointed out that it is generally a complicated task to fully specify a scenario. Additionally, the editing is easier if a graphical user interface (GUI) can be used.

For the review, Bernhard was presented Scenario 3 shown in Figure 7.7 in form of a hand drawn sketch and in form of the raw OpenScenario XML. He first analyzed the sketch and said that he never opened the underlying scenario description files of VTD, as the GUI has everything he needs for simple scenarios. However, he then built the simple version of Scenario 3 with the VTD scenario creation tool to compare the formats.

During the comparison of both scenario description formats he recognized similarities between the Vires VTD scenario format and OpenScenario. As Vires is the main developer of the OpenScenario standard, this fact is understandable. The biggest similarity he found, is the usage of libraries. Both scenario formats use libraries for the definition (i) of vehicle attributes and (ii) of specific trajectories. Although the scenario was very short, he said that one can assume that the underlying description logic is very similar. However he pointed out that the XML tags and the nesting logic show the main differences.

Kailin Tong uses a scenario generation tool for the traffic simulation platform SUMO. Kailin was provided the same setup as Bernhard. He got the sketch of the scenario as well as the raw OpenScenario XML. At the beginning of the interview Kailin pointed out that he uses a GUI to design the principle of the scenario. Afterwards he sets specific parameters directly in the resulting XML files. When he compared the SUMO format to the OpenScenario format, he could find similarities in the modular approach of both scenarios. However, as he uses rather simple scenarios for his simulations, the complexity of the event logic in OpenScenario was rather new to him.

Then, after a more detailed analysis he pointed out that SUMO simulates the vehicles different than what is expected from simulators using OpenScenario. While SUMO uses parameters to control the vehicles, OpenScenario grants the opportunity to trigger specific routes for a target vehicle, as soon as a condition is met. He continued to explain that he usually does not need event triggered behavior within his SUMO simulations, but he supports the idea of the underlying concept for complex simulations.

Both interviews turned out to be consistent with the impression that Vires is developing the OpenScenario format with industrial and research partners. The main reason for that is the versatile complexity of the OpenScenario format. As Bernhard said, the format allows to create complex and detailed scenarios. And it is thereby also flexible and modular as Kailin mentioned with his expertise.

9.4 OpenScenario Format

The chosen scenario description format OpenScenario fulfills the basic requirements outlined in 5.2.1. Especially modular approaches to use libraries for vehicles and maneuvers avoids redundancies. Also the nested Syntax of XML makes parsing and validation relatively easy. During implementation, version 0.9.1 was used. Due to the pending release, there is currently no editor with a GUI available. A change in the XML file of the scenario has therefore be made by hand.

Comparing the OpenScenario format to other approaches in literature, one can see that the OpenScenario format has strong similarities to the database format presented by [21]. The basic principle of the possibility to reuse parts of the scenario (like the vehicle definition or specific trajectories) is the major feature of both approaches. Although there are no editors for the format, the modularity of OpenScenario is the key feature for a scenario library.

One can estimate from the features of the format, as well as the two interviews provided in Chapter 9.3 that the OpenScenario format has the potential to become a standard for file based scenario databases. As Vires showed with the OpenDrive format, they are motivated to provide a common basis

with the help of industrial and research partners. But a lot depends on competitors. For example potential competitors may also provide scenarios in database form, as [21] proposed.

9.5 Performance

To explicitly test the performance, the scenarios from Figure 7.7 have been used. In the first attempt, all scenarios had three oncoming non-ego vehicles as traffic. This immediately lead to the first critical problem: the Scenario Loader is slower than the calculations of the Carla server. This resulted in lagging vehicles on the simulator. After deep analysis of the code, it turned out that the locking mechanism used to synchronize the threads creates roughly one locking call per 5-10ms. Python is not designed to access the hardware level at such a frequent rate. After talking to Matthias Scharrer, a Python expert at the VIF and doing some desk research, the issue was confirmed. At this point it can be said that, as the Python prototype is working, the software should be reimplemented in C++ to guarantee execution speed.

After discovering this underlying software performance issue, the tested scenarios were reduced to simulate only one additional non-ego vehicle. With only the ego vehicle and one non-ego vehicle left, the Scenario Loader stayed slightly below maximum processing load. At this point it also has to be mentioned that a sync mode of Carla would fix this problem, as the server would not be able to flood the Scenario Loader with new calculation steps. The Scenario Loader would then be able to take as long for thread scheduling as necessary.

The overall performance of the Carla simulator has already been evaluated in Chapter 9.1. The Unreal Engine is generally a fitting simulation platform for the Scenario Loader, with the exception of the missing synchronous mode. This missing feature also leads to small jitters of the non-ego cars in the simulation. The loose asynchronous connection therefore prevents smooth car placement. The non-ego placement, as explained in Chapter 8.5, is generally applicable. However, due to the asynchronous state, the current timestamp does not always match the time stamp of the server.

Therefore, setting the forward speed of the non-ego vehicle is like applying a smoothing filter to the problem.

Even though the vehicle movement can be fixed to a large extent, scenario 4 in Figure 7.7 is not working. In all other three scenarios the non-ego vehicle drives straight ahead. When the position of the vehicle is set, the forward speed is taken and the simulator can calculate logical physical effects. However, when the non-ego vehicle in scenario 4 takes a left turn, the vehicle is slightly rotated at each point. Due to a missing steering angle of the wheels and due to missing rotational speed, the vehicle is not placed smoothly. The most important thing to get this scenario to work, is the synchronization mode. Additionally, as soon as the Carla team provides access to the physical model of the vehicles, the relation between steering angle and steering float value [0..1] can be set, which should smoothen the movement similar to setting the forward speed.

9.6 Scalability And Expandability

Summing up the insights of the previous Chapters 9.1-9.5, it can be said that the Carla simulator based on the Unreal Engine is highly scalable. It is able to handle multiple clients, as well as multiple simulated vehicles. Additionally the Scenario Loader can easily handle single scenario files, up to big databases consisting of scenario files in folders.

The modular architectural concept of the Scenario Loader software, presented in Chapter 7, guarantees easy implementation of additional features. These features may be additional parameters contained in the OpenScenario file, up to new features of the Carla simulator. It is even possible to exchange the scenario format, or the simulator with minimal effort, as the controlling classes implement the adapter pattern.

However, iterative implementation lead to one issue: the actors also based on the adapter pattern started to contain code redundancy and are too integrated into parts of the communication to the server. Overall, if one considers to refactor the actor design, it is recommended to also consider switching to a more embedded programming language. For example C++,

which has the big advantage of a more shallow connection to the locking mechanisms of the operating system.

A missing feature is the integration of the routing algorithm for the ego vehicle. As elaborated in Chapter 8.5, the non-ego vehicles already have a route in their execution queue. The same calculations have to be done for the ego-vehicle. But in contrast to the non-ego vehicle, the ego route has to be published as path message, for the AD SIL to follow. This step should guarantee that the ego vehicle also takes a specific route in the scenario.

10 Conclusion

The Scenario Loader designed within this thesis combines the strengths of an existing simulator and an existing scenario format with the scenario handling principles of multiplayer game engines. The performance of the architecture turned out to be reasonable, although it struggles with Python as the chosen programming language. The Chapters 10.1–10.3 will continue to discuss and summarize the resulting solution, respectively point out improvements and possible future applications.

10.1 Discussion

The best way to analyze the outcome of this thesis is to discuss the three main components introduced in Chapter 3.

The Simulation Engine fulfills the main requirements stated in Chapter 5.1.1. During the development phase, the missing capability to simulate pedestrians was a problem. However, the interfaces of the Scenario Loader are ready and can easily be adapted to also use the pedestrian models which were reintroduced in the newest Carla version. The only thing absolutely missing in Carla is a synchronized mode, as stated multiple times in Chapter 9.

The synchronized mode was no requirement in Chapter 5.1.1. However, all important simulation platforms support that out of the box. Even Carla supported it until version 0.8.4. Since version 0.9.0, also the Carla community is waiting for the reimplementation. But generally speaking, the decision to utilize the Carla simulator was the best. While it was time intensive to adapt to the version changes, it saved costs for a platform with closed source.

The Scenario Format OpenScenario is still promising for the future and combines the knowhow from industry and research, as stated in Chapter 9.4. The format provides a good basis for straight forward, as well as very complex scenarios. However, the standard is not yet fully released. The only accessible version is OpenScenario 0.9.1. Missing documentation and potential deprecated code also prevented the creation of a simple GUI to create scenarios.

Controlling Software is based on the design of multiplayer game engines, as presented by Chapter 7. The implementation, explained in more detail in Chapter 8, has an additional focus on modular flexibility. Both, the underlying design as well as the software architecture, support iterative development approaches. This way it is possible to exchange the used simulation engine as well as the scenario format with minimal effort. However, the biggest flaw in the resulting Scenario Loader software is the used programming language Python. While Python was a good choice, as it allowed for rapid development, it massively lacks performance when it comes to time critical synchronized threading. As analyzed in Chapter 9.5, Python renders multi non-ego vehicle simulations impossible if mid range processing hardware is used.

All in all, the Scenario Loader fulfills the expectations and is simply lacking manpower to further implement the OpenScenario standard.

10.2 Known Issues And Recommended Improvements

The overall architecture and implementation of the Scenario Loader software fits the requirements stated in Chapter 3. However, the proof of concept still contains some issues as outlined in the previous Chapter 9. These issues and other recommended improvements are listed in the following:

Python should be exchanged for a more embedded programming language like C++. As elaborated in Chapter 9.5, the synchronization of the

actor threads creates a major performance issue: due to the high level scripting approach of Python, the access to the atomic locking mechanism takes too much computational power.

Synchronous mode is a must have for future iterations. The missing synchronous mode in Carla 0.9.3 creates major issues not only for the performance, but also for the accuracy. As soon as the synchronous mode is implemented, most of the existing issues listed below can be easily fixed, or may even be obsolete.

Turning vehicles can only be simulated when they drive themselves, as elaborated in Chapter 9.5. When the vehicles are placed via time progressive positioning, the physics engine can not handle rotating vehicles due to the missing steering angle. However, this issue is mainly created by the missing synchronous mode. Additionally it can not be fixed, as the physical models of the simulated vehicles are not accessible. In other words, the Scenario Loader software can not create a relation between the simulated physical steering angle values and the steering float value [0..1], which has to be provided.

Jump backs in server time are also caused by the asynchronous mode of the Carla server. The Carla server mostly provides a constant time delta between the simulation steps. However, within a timeframe of 1-5 seconds, the server provides the same simulation time twice. This issue gets more prominent when the computer, running the Carla server, is under heavy load. The origin of this issue is yet unknown. It might be an Unreal Engine problem, a Carla problem, or even a problem triggered by the Scenario Loader software.

Deadlock on scenario end happens sometimes on bad scheduling. The test control tries to disconnect the simulator control. At the same time it may happen that the run_cb method is currently active. Due to resource accesses, the program runs into a deadlock. The issue can be fixed with a simply lock. However, this lock would create additional overhead, as it

would be checked at multiple locations. This leads in further consequence to timing destabilization in combination with the Python locking mechanism and the missing synchronization mode.

Refactor the actor class to reduce code redundancy. Due to iterative implementation, the adapter pattern architecture of the actor is not fulfilled. For example event handling and execution queue handling are too integrated into the CarlaActor. Additionally the handling of events and the execution queue could be converted into a strategy design pattern. This approach would provide even more flexibility

Actor routing for the ego actor is missing. The ego actor should publish the calculated execution queue as route for the AD SIL. This would work best in combination with an interface pattern, to be more flexible on the used SIL. Additionally, the refactoring of the actor class might come in handy, as the strategy pattern could easily differentiate between ego and non-ego execution queue calculation.

10.3 Future Work

After the scientific part of this thesis one might consider the following points:

Analyse different scenario loading approaches of existing simulators. As stated in Chapter 4, most research is done in the field of different forms of HIL and different forms to represent scenarios. For both research applications it seems to be necessary effort to load and play scenarios. However, especially in game design there are different possibilities to handle stories/scenarios. In other words, it would be interesting to apply different methodologies of story handling from game design to AD SIL testing.

Testing multiple autonomous cars at once might grant additional testing possibilities. The common perception of X in the loop is one element in the loop. In case of this thesis, this is SIL: meaning the AD software of one vehicle in the loop. However, as Carla already provides the simulator material to test multiple ego cars, it should be investigated if simulating multiple ego cars at once gathers benefit. While this approach seems to generate non-imitable situations at first, it is a great possibility for AD software to deal with calculation mistakes from other AD implementations. Compared to human standards, AD SIL testing would be driving school up to years of experience. Testing multiple autonomous vehicles with injections of small situations, like in human traffic, could be compared to every day traffic or to monkey testing approaches in software testing.

Reimplement the software in C++ to get access to the full performance. A similar idea is already planned, as this thesis will be made publicly available to the Carla development team. Although they will most likely redesign the software for their needs, the architecture and experience contained in this thesis might help them to avoid specific pitfalls.

Appendix

Acronyms

ABS anti-lock braking system. 12
ACC Adaptive Cruise Control. 1
AD automated driving. 3, 6, 9, 15, 16, 18, 22, 25–29, 31, 35–37, 43, 45, 47, 48, 58
ADAS advanced driver assistance systems. 1, 4–7, 11–13, 16, 21–23, 29, 33, 34, 36
ADS automated driving systems. 1, 3–5, 7, 9, 11, 21, 23, 34, 36
AEB automated emergency braking assistance. 12
AI Artificial Intelligence. 28
AR Augmented Reality. 6

CVIS Cooperative Vehicle Infrastructure System. 34

DBW drive-by-wire. 40, 47, 48

ECU electronic control unit. 3, 5, 12

FPS frames per second. 62

GPS Global Positioning System. 34, 35
GUI graphical user interface. 62, 63

HIL hardware in the loop. 5, 6, 18, 21
HMD head mounted display. 6

ICARSC International Conference on Autonomous Robot Systems and Competitions. 11
ICRA International Conference on Robotics and Automation. 11
IEEE Institute of Electrical and Electronics Engineers. 23
IROS International Conference on Intelligent Robots and Systems. 11
ITSC International Conference on Intelligent Transportation Systems. 11

IV Intelligent Vehicles Symposium. 11

LIDAR light detection and ranging. 22, 25, 28, 29, 62
LKA Lane Keeping Assistant. 1

MBSS model based scenario specification. 39
MIL model in the loop. 5, 21
MORSE Modual OpenRobots Simulation Engine. 35

OEM Original Equipment Manufacturer. 1

PELOPS Programm zur Entwicklung längsdynamischer, mikroskopischer Prozesse in system relevanter Umgebung. 35

RHIL robot hardware in the loop. 18
ROS robot operating system. 7, 18, 21, 22, 25–29, 31–33, 35, 39, 40, 43, 47, 48, 58
RPC remote procedure call. 61

SAE Society of Automotive Engineers. 1, 5, 9
SIL software in the loop. 3–7, 9, 13, 19, 21
SiVIC Simulator of Vehicle, Infrastructure and sensor. 33

TJA Traffic Jam Assistant. 1

VIF VIRTUAL VEHICLE Research Center. 22, 23, 25–27, 36, 56, 62
VANET vehicular ad hoc network. 15
VeHIL vehicle hardware in the loop. 6, 13
VIL vehicle in the loop. 6, 13, 21

XML Extensible Markup Language. 39, 41, 43, 54, 62, 63
XSD XML Schema Definition. 54

Bibliography

[1] S. of Automotive Engineers SAE, *Taxonomy and definitions for terms related to on-road motor vehicle automated driving systems*, Jan. 2014. DOI: https://doi.org/10.4271/J3016_201401. [Online]. Available: https://doi.org/10.4271/J3016_201401 (cit. on pp. 1, 3, 7).

[2] Groovecar. (Mar. 2019). Autonomous parking - image, [Online]. Available: https://www.groovecar.com/media/images/articles/2016/09/tech-out-my-new-car/automatic-parking-new-advances-from-chrysler/automatic-parking-new-advances-from-chrysler-1.jpg (cit. on p. 2).

[3] A. Miglani, "User interface design for highly and fully automated driving: Designing for system trust and comfort in non-driving related tasks," PhD thesis, Aug. 2017 (cit. on p. 3).

[4] A. GmbH. (Jan. 2019). Alp-lab homepage, [Online]. Available: https://www.alp-lab.at/ (cit. on p. 4).

[5] D. J. Verburg, A. C. M. van der Knaap, and J. Ploeg, "Vehil: Developing and testing intelligent vehicles," in *Intelligent Vehicle Symposium, 2002. IEEE*, vol. 2, Jun. 2002, 537–544 vol.2. DOI: 10.1109/IVS.2002.1188006 (cit. on p. 8).

[6] T. Bokc, M. Maurer, and G. Farber, "Validation of the vehicle in the loop (vil); a milestone for the simulation of driver assistance systems," in *2007 IEEE Intelligent Vehicles Symposium*, Jun. 2007, pp. 612–617. DOI: 10.1109/IVS.2007.4290183 (cit. on p. 8).

[7] S. Ulbrich, T. Menzel, A. Reschka, F. Schuldt, and M. Maurer, "Defining and substantiating the terms scene, situation, and scenario for automated driving," in *2015 IEEE 18th International Conference on Intelligent Transportation Systems*, Sep. 2015, pp. 982–988. DOI: 10.1109/ITSC.2015.164 (cit. on pp. 8, 17–19, 37).

[8] O. S. R. Foundation. (Jan. 2019). Ros-homepage, [Online]. Available: http://www.ros.org/ (cit. on pp. 9, 41).

[9] A. Hellmund, S. Wirges, Ö. Ş. Taş, C. Bandera, and N. O. Salscheider, "Robot operating system: A modular software framework for automated driving," in *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*, Nov. 2016, pp. 1564–1570. DOI: 10.1109/ITSC.2016.7795766 (cit. on pp. 9, 41).

[10] D. Gruyer, S. Choi, C. Boussard, and B. d'Andréa-Novel, "From virtual to reality, how to prototype, test and evaluate new adas: Application to automatic car parking," in *2014 IEEE Intelligent Vehicles Symposium Proceedings*, Jun. 2014, pp. 261–267. DOI: 10.1109/IVS.2014.6856525 (cit. on pp. 13, 35).

[11] M. R. Zofka, S. Klemm, F. Kuhnt, T. Schamm, and J. M. Zöllner, "Testing and validating high level components for automated driving: Simulation framework for traffic scenarios," in *2016 IEEE Intelligent Vehicles Symposium (IV)*, Jun. 2016, pp. 144–150. DOI: 10.1109/IVS.2016.7535378 (cit. on p. 13).

[12] J. E. Stellet, M. R. Zofka, J. Schumacher, T. Schamm, F. Niewels, and J. M. Zöllner, "Testing of advanced driver assistance towards automated driving: A survey and taxonomy on existing approaches and open questions," in *2015 IEEE 18th International Conference on Intelligent Transportation Systems*, Sep. 2015, pp. 1455–1462. DOI: 10.1109/ITSC.2015.236 (cit. on pp. 14, 15).

[13] S. Khastgir, S. Birrell, G. Dhadyalla, and P. Jennings, "Identifying a gap in existing validation methodologies for intelligent automotive systems: Introducing the 3xd simulator," in *2015 IEEE Intelligent Vehicles Symposium (IV)*, Jun. 2015, pp. 648–653. DOI: 10.1109/IVS.2015.7225758 (cit. on p. 14).

[14] M. Feilhauer and J. Häring, "A multi-domain simulation approach to validate advanced driver assistance systems," in *2016 IEEE Intelligent Vehicles Symposium (IV)*, Jun. 2016, pp. 1179–1184. DOI: 10.1109/IVS.2016.7535539 (cit. on pp. 15, 16).

[15] C. Zhang, Y. Liu, D. Zhao, and Y. Su, "Roadview: A traffic scene simulator for autonomous vehicle simulation testing," in *17th International IEEE Conference on Intelligent Transportation Systems (ITSC)*, Oct. 2014, pp. 1160–1165. DOI: 10.1109/ITSC.2014.6957844 (cit. on pp. 15, 35).

[16] M. R. Zofka, R. Kohlhaas, T. Schamm, and J. M. Zöllner, "Semivirtual simulations for the evaluation of vision-based adas," in *2014 IEEE Intelligent Vehicles Symposium Proceedings*, Jun. 2014, pp. 121–126. DOI: 10.1109/IVS.2014.6856593 (cit. on p. 15).

[17] F. Michaeler and C. Olaverri-Monreal, "3d driving simulator with vanet capabilities to assess cooperative systems: 3dsimvanet," in *2017 IEEE Intelligent Vehicles Symposium (IV)*, Jun. 2017, pp. 999–1004. DOI: 10.1109/IVS.2017.7995845 (cit. on p. 17).

[18] G. E. Mullins, P. G. Stankiewicz, and S. K. Gupta, "Automated generation of diverse and challenging scenarios for test and evaluation of autonomous vehicles," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, May 2017, pp. 1443–1450. DOI: 10.1109/ICRA.2017.7989173 (cit. on pp. 18, 38).

[19] Cognata. (Jan. 2019). Cognata-homepage, [Online]. Available: https://www.cognata.com/ (cit. on pp. 18, 33).

[20] D. Zhao, Y. Guo, and Y. J. Jia, "Trafficnet: An open naturalistic driving scenario library," in *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*, Oct. 2017, pp. 1–8. DOI: 10.1109/ITSC.2017.8317860 (cit. on pp. 18, 19, 38).

[21] J. Bach, S. Otten, and E. Sax, "Model based scenario specification for development and test of automated driving functions," in *2016 IEEE Intelligent Vehicles Symposium (IV)*, Jun. 2016, pp. 1149–1155. DOI: 10.1109/IVS.2016.7535534 (cit. on pp. 19, 40, 76, 77).

[22] RoboCup. (Feb. 2019). Robocup-homepage, [Online]. Available: https://www.robocup.org/ (cit. on p. 20).

[23] I. TU-Graz. (Feb. 2019). Ist-homepage, [Online]. Available: http://www.robocup.tugraz.at/?p=1553 (cit. on p. 20).

[24] K. Takaya, T. Asai, V. Kroumov, and F. Smarandache, "Simulation environment for mobile robots testing using ros and gazebo," in *2016 20th International Conference on System Theory, Control and Computing (ICSTCC)*, Oct. 2016, pp. 96–101. DOI: 10.1109/ICSTCC.2016.7790647 (cit. on p. 20).

[25] A. Martin and M. R. Emami, "An architecture for robotic hardware-in-the-loop simulation," in *2006 International Conference on Mechatronics and Automation*, Jun. 2006, pp. 2162–2167. DOI: 10.1109/ICMA.2006.257628 (cit. on p. 20).

[26] K. Bousmalis, A. Irpan, P. Wohlhart, Y. Bai, M. Kelcey, M. Kalakrishnan, L. Downs, J. Ibarz, P. Pastor, K. Konolige, S. Levine, and V. Vanhoucke, "Using simulation and domain adaptation to improve efficiency of deep robotic grasping," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, May 2018, pp. 4243–4250. DOI: 10.1109/ICRA.2018.8460875 (cit. on p. 21).

[27] C. Pilz, "Realistic simulation of a time of flight camera for autonomous parking," Master project at University of Technology Graz, Aug. 2017 (cit. on p. 23).

[28] A. E. Gómez, T. C. dos Santos, C. M. Filho, D. Gomes, J. C. Perafan, and D. F. Wolf, "Simulation platform for cooperative vehicle systems," in *17th International IEEE Conference on Intelligent Transportation Systems (ITSC)*, Oct. 2014, pp. 1347–1352. DOI: 10.1109/ITSC.2014.6957874 (cit. on p. 24).

[29] K. S. Swanson, A. A. Brown, S. N. Brennan, and C. M. LaJambe, "Extending driving simulator capabilities toward hardware-in-the-loop testbeds and remote vehicle interfaces," in *2013 IEEE Intelligent Vehicles Symposium (IV)*, Jun. 2013, pp. 122–127. DOI: 10.1109/IVS.2013.6629458 (cit. on p. 24).

[30] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An open urban driving simulator," in *Proceedings of the 1st Annual Conference on Robot Learning*, 2017, pp. 1–16 (cit. on pp. 26, 36, 39, 41, 42).

[31] C. Aguero, N. Koenig, I. Chen, H. Boyer, S. Peters, J. Hsu, B. Gerkey, S. Paepcke, J. Rivero, J. Manzo, E. Krotkov, and G. Pratt, "Inside the virtual robotics challenge: Simulating real-time robotic disaster response," *Automation Science and Engineering, IEEE Transactions on*, vol. 12, no. 2, pp. 494–506, Apr. 2015, ISSN: 1545-5955. DOI: 10.1109/TASE.2014.2368997 (cit. on p. 26).

[32] O. S. R. Foundation. (Jan. 2019). Gazebo-homepage, [Online]. Available: http://gazebosim.org/ (cit. on p. 27).

[33] CVEDIA. (Jan. 2019). Syncity real-world simulator for autonomous applications, [Online]. Available: https://syncity.com/ (cit. on p. 27).

[34] tass international. (Jan. 2019). Prescan-homepage, [Online]. Available: https://tass.plm.automation.siemens.com/prescan (cit. on pp. 28, 29).

[35] V. S. GmbH. (Jan. 2019). Vires -homepage, [Online]. Available: https://vires.com/ (cit. on p. 28).

[36] ——, (Jan. 2019). Opendrive-homepage, [Online]. Available: http://www.opendrive.org/ (cit. on pp. 28, 39).

[37] ——, (Jan. 2019). Openscenario-homepage, [Online]. Available: http://www.openscenario.org/ (cit. on pp. 28, 39, 61).

[38] A. Best, S. Narang, D. Barber, and D. Manocha, "Autonovi: Autonomous vehicle planning with dynamic maneuvers and traffic constraints," *CoRR*, vol. abs/1703.08561, 2017. arXiv: 1703.08561. [Online]. Available: http://arxiv.org/abs/1703.08561 (cit. on p. 29).

[39] R. Inc. (Jan. 2019). Righthook-homepage, [Online]. Available: https://righthook.io/ (cit. on pp. 29, 30).

[40] I. A. GmbH. (Jan. 2019). Ipg automotive carmaker-homepage, [Online]. Available: https://ipg-automotive.com/de/produkte-services/simulation-software/carmaker/ (cit. on p. 30).

[41] D. B.V. (Jan. 2019). Racer-homepage, [Online]. Available: http://www.racer.nl/ (cit. on pp. 30, 31).

[42] AVSimulation. (Jan. 2019). Scaner-homepage, [Online]. Available: https://www.avsimulation.fr/ (cit. on p. 31).

[43] Microsoft. (Jan. 2019). Airsim-github, [Online]. Available: https://github.com/microsoft/airsim/ (cit. on pp. 31, 32).

[44] rFpro. (Jan. 2019). Rfpro-homepage, [Online]. Available: http://www.rfpro.com/ (cit. on p. 32).

[45] Udacity. (Jan. 2019). Udacity self-driving-car-sim - github, [Online]. Available: https://github.com/udacity/self-driving-car-sim/ (cit. on pp. 32, 33).

[46] J. Olstam and R. Elyasi-Pour, "Combining traffic and vehicle simulation for enhanced evaluations of powertrain related adas for trucks," in *16th International IEEE Conference on Intelligent Transportation Systems (ITSC 2013)*, Oct. 2013, pp. 851–856. DOI: 10.1109/ITSC.2013.6728338 (cit. on p. 33).

[47] D. Gruyer, O. Orfila, V. Judalet, S. Pechberti, B. Lusetti, and S. Glaser, "Proposal of a virtual and immersive 3d architecture dedicated for prototyping, test and evaluation of eco-driving applications," in *2013 IEEE Intelligent Vehicles Symposium (IV)*, Jun. 2013, pp. 511–518. DOI: 10.1109/IVS.2013.6629519 (cit. on pp. 34, 35).

[48] K. G. Lim, C. H. Lee, R. K. Y. Chin, K. B. Yeo, and K. T. K. Teo, "Simulators for vehicular ad hoc network (vanet) development," in *2016 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia)*, Oct. 2016, pp. 1–4. DOI: 10.1109/ICCE-Asia.2016.7804797 (cit. on p. 34).

[49] W. ShangGuan, H. Guo, P. Liu, B. Cai, and J. Wang, "Research of interactive visual simulation method based on cooperative vehicle infrastructure system," in *17th International IEEE Conference on Intelligent Transportation Systems (ITSC)*, Oct. 2014, pp. 121–126. DOI: 10.1109/ITSC.2014.6957677 (cit. on p. 35).

[50] S. Carpin, M. Lewis, J. Wang, S. Balakirsky, and C. Scrapper, "Usarsim: A robot simulator for research and education," in *Proceedings 2007 IEEE International Conference on Robotics and Automation*, Apr. 2007, pp. 1400–1405. DOI: 10.1109/ROBOT.2007.363180 (cit. on p. 36).

[51] F. K. mbH Aachen. (Jan. 2019). Pelops whitepaper, [Online]. Available: https://www.fka.de/images/pelops-whitepaper.pdf (cit. on p. 36).

[52] M. Team. (Jan. 2019). Morse-homepage, [Online]. Available: https://www.openrobots.org/morse/doc/stable/what_is_morse.html (cit. on p. 36).

[53] O. S. R. Foundation. (Jan. 2019). Sdformat-homepage, [Online]. Available: http://sdformat.org/ (cit. on p. 40).

[54] M. Software. (Jan. 2019). Vires announces transfer of opendrive® standard to asam e.v., [Online]. Available: http://www.mscsoftware.com/node/8467 (cit. on p. 40).

[55] C. Pilz. (Mar. 2019). Carlascenarioloader - git repo, [Online]. Available: https://github.com/MrMushroom/CarlaScenarioLoader (cit. on p. 41).

[56] C. Ltd. (Jan. 2019). Ubuntu-homepage, [Online]. Available: https://www.ubuntu.com/ (cit. on p. 41).

[57] D. Inc. (Jan. 2019). Dataspeed-bitbucket-homepage, [Online]. Available: https://bitbucket.org/DataspeedInc/dbw_mkz_ros (cit. on p. 42).