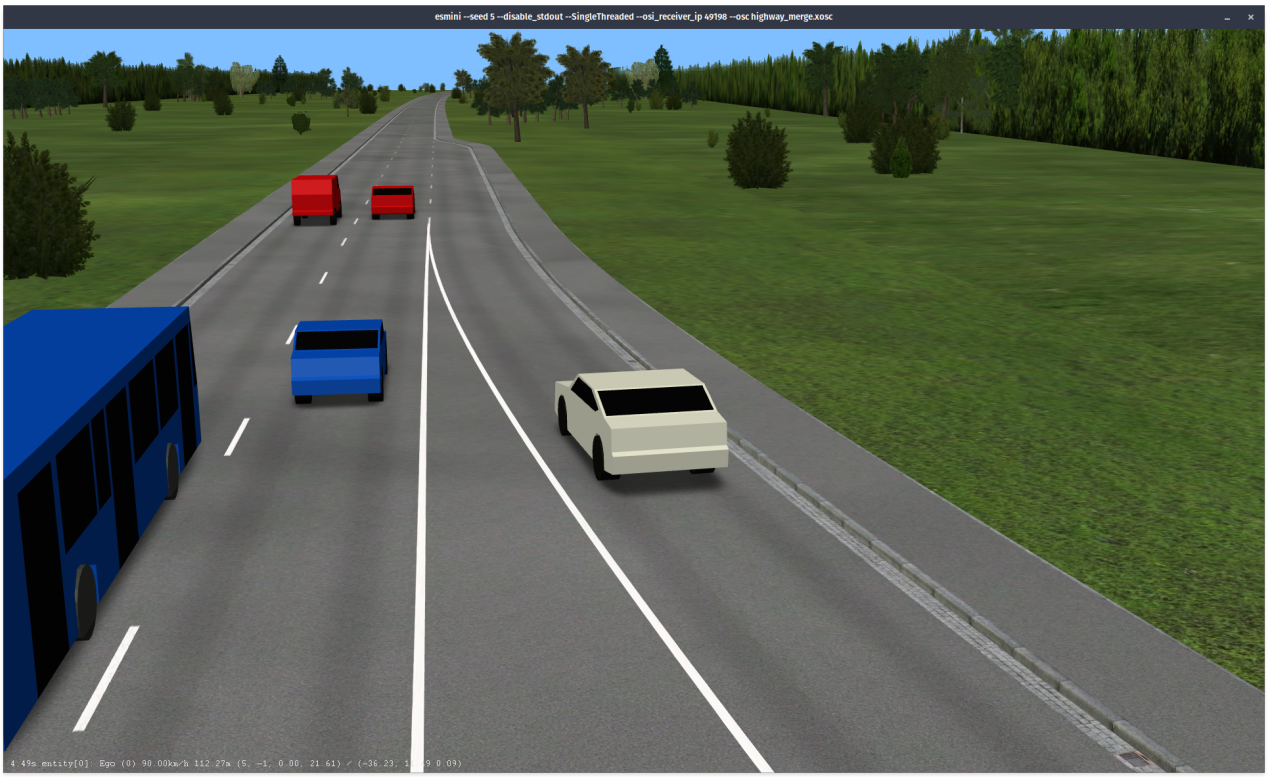
CHALMERS



The impact of programming language choice on execution time when performing virtual simulation with a driver model

A comparison of C++ and Python performance using the open simulation interface (OSI) in esmini

Master's thesis in Automotive Engineering

BURAK C. SOYDAŞ

Mechanics and Maritime Sciences
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2024

MASTER'S THESIS IN AUTOMOTIVE ENGINEERING

The impact of programming language choice on execution time when performing virtual simulation with a driver model

A comparison of C++ and Python performance using the open simulation interface (OSI) in esmini

BURAK C. SOYDAŞ

Mechanics and Maritime Sciences
Division of Vehicle Safety
CHALMERS UNIVERSITY OF TECHNOLOGY

Göteborg, Sweden 2024

The impact of programming language choice on execution time when performing virtual simulation with a driver model
A comparison of C++ and Python performance using the open simulation interface (OSI) in esmini

BURAK C. SOYDAŞ

Supervisor and Examinor: Associate Prof. Jonas Bärgman

Mechanics and Maritime Sciences
Division of Vehicle Safety
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone: +46 (0)31-772 1000

Cover:
Virtual simulation of an example merge manoeuvre on a highway, which is included with esmini and executed by it in this cover.

Chalmers Reproservice
Göteborg, Sweden 2024

The impact of programming language choice on execution time when performing virtual simulation with a driver model
A comparison of C++ and Python performance using the open simulation interface (OSI) in esmini

BURAK C. SOYDAŞ
Mechanics and Maritime Sciences
Division of Vehicle Safety
Chalmers University of Technology

Abstract

In the past decades the automotive industry increased the usage of computer simulations in diverse fields for the purposes like Verification & Validation (V & V), research, development and with progress made in the development of Automated Driving System (ADS) and Advanced Driver Assistant Systems (ADAS), the legislation. One form of computer simulation used for those purposes are virtual simulations. In the case of research for ADS, driver behavior and driver models (DM), a virtual world is created with the intention to recreated a real-world scenario for simulation applications. The many benefits it offers for this purpose can be improved by using more complex algorithms or larger-scale simulations. Based on the complexity and computation required for increasing the accuracy of those, the available performance might be a limiting factor. Therefore, the focus of this thesis was on comparing the performance of a compiled and an interpreted programming language driving a minimal reproducible example of virtual simulation as it could be implemented for the stated research applications. The open-source environment simulator esmini was used together with a DM based on the theory of predictive processing. Respectively, the main function execution times and statistics of the GNU time function were collected. Both parts of the simulation were implemented in C++ and Python. The results showed substantially faster times for the C++ DMs main function, and that wrapping the esmini library in Python using the ctypes library didn't affect the simulations main function execution. Statistics of GNU time showed better results for both, the DM and simulation, when C++ was used.

Keywords: Simulation, Driver Model, C++, Python, Performance, ADAS, ADS

Acknowledgements

The process of writing this thesis was a long one. Next to the academical and technical challenges, a lot about personal matters was learned due to personal challenges along the way. The completion of this thesis and the lessons learned along the way is mainly attributed to the understanding, support and genuine care from people, many of whom I consider friends. I can sadly not list all of them, but would like to express my gratitude to at least a few.

I would like to start with my supervisor Jonas Bärgman for making this thesis possible in the first place and for his guidance, understanding as well as his patience along the whole way. Without doubt, this thesis wouldn't have been completed without him. In the academic context, I would also like to thank the people at Volvo. Emil Knabe, who took the time to help me with questions regarding esmini, as well as Malin Svärd and Simon Lundell, who helped me with questions regarding the driver model after providing the source code to a C++ implementation of the driver model. On a more personal level, my special thanks go to my parents Sadeguel and Birol, as well as my sister Asena. They put me on this academic path and more importantly, always supported me along every step I took in any way they could. This is also true for my better half Jasmin, who together with my family always encouraged me to continue. Finally, I would like to thank Lars Lindén for listening to a lot of rant when things didn't work and helping me focus on what was important during the final stages of the project.

Burak C. Soydaş, Gothenburg, 2024

List of Abbreviations

Below is the list of abbreviations that have been used throughout this thesis listed in alphabetical order:

ABS	Anti-lock Brake System
ADAS	Advanced Driver Assistance Systems
ADS	Automated Driving System
AEB	Automated Emergency Braking
ALU	Arithmetic Logic Unit
API	Application Programming Interface
ASAM	Association for Standardization of Automation and Measuring systems
CAD	Computer Aided Design
CAE	Computer Aided Engineering
CFD	Computational Fluid Dynamics
CPU	Central Processing Unit
esmini	Environment Simulator Minimalistic
DiL	Driver-in-the-Loop
FCW	Forward Collision System
FEA	Finite-Element Analysis
FMI	Functional Mock-Up Interface
FMU	Functional Mock-up Unit
SSP	System Structure and Parametrization
HiL	Hardware-in-the-Loop
I/O	Input and Output
ICE	Internal-Combustion Engine
IDM	Intelligent Driver Model
IP	Internet Protocol
IQR	Interquartile Range
ISA	Instruction Set Architecture
Mil	Model-in-the-Loop
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistors
OEM	Original Equipment Manufacturer
openPASS	open Platform for Assessment of Safety Systems
OS	Operating System
OSI	Open Simulation Interface
PBA	Precrash Braking Assist
PCS	PreCollision System
R& D	Research & Development
SAE	Society of Automotive Engineers
SiL	Software-in-the-Loop
TCP	Transmission Control Protocol
TTC	Time-To-Collision
UDP	User Datagram Protocol
UML	Unified Modeling Language
V & V	Verification & Validation
ViL	Vehicle-in-the-Loop
VTD	Virtual Test Drive
XML	eXtensible Markup Language

List of Figures

List of Tables

Contents

1 Introduction

For many people living today, the primary mode of individual transportation are cars, mopeds, or other smaller traffic vehicles [1]. While those offer many advantages for personal mobility and transportation needs, they also create problems: In 2019, road traffic accidents caused 1.3 million deaths [2] and in 2018, it has been the leading g cause of death for young people aged 5 to 29 years as well as the eight leading cause of death generally [3][4].

In an effort to make road traffic safer for all participants and bring down the number of deaths and injuries, the automotive industry, academia and research organizations, governmental as well as non-governmental organizations [5] conduct studies and research in various fields. The research is utilized to improve passive safety (mitigating the consequences of crashes), active safety (preventing crashes) and the understanding of the human behaviour in participating in road traffic, especially behaviour leading to crashes and accidents. Advanced Driver Assistance Systems (ADAS) and Automated Driving Systems (ADS) are viewed as potential contributors to making traffic safer [6].

ADAS are already built into production cars for several decades with the introduction of Anti-lock brake systems (AEB) in 1978 by the Robert Bosch GmbH and have since increased in numbers and functionality [7] while efforts to reach higher driving automation levels according to the definitions of the Society of Automotive Engineers (SAE) for ADS are ongoing. The development of those systems require safety performance assessments of the functionality as well as driver acceptance [8]. A failure in either of these two aspects means that safety might be compromised, possibly with critical consequences. While it is directly deducted how a fail in functionality directly compromises the safety, a fail in acceptance of safety systems might lead to their deactivation by the driver [9]. This means that the safety system cannot perform any actions to affect safety, effectively resulting in the same level of safety of the same vehicle without the safety system equipped at all. Therefore, during numerous stages of development, engineers use various simulations, procedures and other tools/tests (e.g, real-world driving) to Verify and Validate (V&V) the functions and test the acceptance of the systems.

In the context of ADAS systems and the goal to achieve complete autonomous driving, virtual simulations are becoming more important in the automotive development, research and testing processes [10][11]. As simulations are based on models and given that models always are a reduced and abstract representation of observations and measured data, the usefulness of virtual simulations highly depend on the accuracy of a given model in the context of interest [12]. Depending on the type and scale of simulation as well as the models accuracy, the required computational cost and simulation time can be high and result in high monetary and time costs. Therefore, the virtual simulation should also ideally execute as fast as possible. While virtual simulations have multiple aspects that could improve their performance like increasing the performance of the used hardware or using costly and power-hungry servers, this thesis deals with the effect of the chosen programming language used for the components of a virtual simulation. More precisely, this thesis examines the impact of using C++ or Python on the performance of running simulations using the Environment Simulator Minimalistic (esmini) application and a driver model connected to it via the Open Simulation Interface (OSI) on a personal computer. Execution times are measured repeatedly in a benchmark setup, and a performance factor describing how many times faster the C++ implementation was over the Python implementation was used to assess the performance.

1.1 Background

Studying and replicating drivers' behaviours is essential for ADAS & AD development and testing. Consequently, work is going on various projects, for example models from the UN regulation 157 in paragraph 5.2.5.2 and appendix 3 (Competent and Careful driver) [13], the Responsibility Sensitive Safety model by Intel/Mobileye and the Fuzzy Safety Model by Mattas et al. [14]. For the prospective assessment of traffic safety of ADAS & ADS in virtual simulations, the P.E.A.R.S. initiative develops methodology for quantitative assessment of crash avoidance [5].

The driver models and ADAS & ADS are used in virtual simulations, e.g. for safety assessments, in various phases of research and development, in which there are different requirements. In early phases, such as in the prototype phase, it is important to be able to quickly and easily write code and debug it, as the code changes often and is executed relatively little. This is the case when developing or tuning computational driver models or conducting research on driver modelling, where often the Python programming language is preferred

over compiled programming languages such as C++, which according to [15] are more complex than Python. On the other hand, the code is matured in later stages such as in product development and virtual safety assessments, meaning that it doesn't change much and is executed often and/or for long durations. The benefits of easy coding and debugging are less relevant and the benefits of fast execution times such when using C++ become more relevant, as it decreases time spent on the simulations. Those simulations are performed using simulation platforms. Some distributors of simulation platforms highlight their usage of C++ for the best possible performance [16] [17], indicating that C++ offers potentially the best performance available.

Generally speaking, the simulation platforms contain the core functionalities such as simulating the environment and stepping the simulation. Often they come with complementary tools to work with the simulations. In section 2.1.2, a selection of simulation platforms are presented. With this background, the aim of this research is to investigate how large the performance penalty is when using Python instead of C++ in the virtual simulation with environment simulator esmini. It is of interest to know how big the trade-off between ease of use and the performance when executed between both programming languages during the aforementioned phases is. Is it eventually feasible to conduct the later phases using Python, or would it be sensible to start the development using C++? Additionally to the language based performance difference, the research tried to identify other performance influencing factors, such as the load on the CPU in any given system state and the impacts of different data structure used by the driver model. While research and publications of performance differences of programming languages exists, they are usually looking at synthetic benchmarks, which can differ quite drastically from real-applications. Other publications look at specific applications, but to the knowledge of the author, no publications deal with virtual simulations for safety assessments, leading to this thesis. The conditions were the usage of the same driver model implemented in C++ and Python with esmini using the OSI.

1.2 Scope

Given a complex enough simulation, there might be multiple points of optimization that can influence the overall performance of a virtual simulation, starting with the hardware and the simulation itself. The scope of this thesis focuses on the analysis of the performance of a Python and a C++ driver model in the context of a virtual simulation environment, and does not deal with any other aspect that could be optimized to increase performance. It also assesses the impact of the choice of programming language for the simulation iteration wrapper. Consequently, the main research question here is:

> How much does the choice of programming language (between Python and C++) affect the execution time of virtual simulations, when the simulations include a driver model interfaced via the open simulation (OSI) interface?

Based on this research question, the following objectives have been derived:

- Design the evaluation experiment
  - Design virtual simulation framework based on the esmini simulation software, using OSI to provide an interface for the driver model
  - Design a basic conflict scenario using the OpenDRIVE [18] and OpenSCENARIO [19] standards to apply the model to
  - Implement a driver model as similarly as possible in both Python and C++
  - Choose and evaluate evaluation metrics to assess both the execution time of the main functionality (threat-assessment and decision-making for the driver model, stepping the simulation for the esmini wrapper) and the execution time of full simulation runs (start to finish)
- Collect and analyse data, reflect on the results

Note that the esmini wrapper is a C++ or Python program that utilizes the esmini API to initialize and step through the simulation scenario defined by the OpenDRIVE and OpenSCENARIO files as well as sending packages using the OSI interface. Completing the simulation setup is the driver model, which is a C++ or Python program running parallel and processing the received OSI GroundTruth to respond with vehicle control updates to esmini.

All driver models are written in an object-oriented style. The functional or procedural programming approaches are not included. As Python offers different data structures, an investigation of how big the choice of Python data structure impacts the execution time, was also performed. The data structures studied include:

- the standard list
- the standard class
- data class, which requires Python 3.10+
- the standard dictionary

All Python driver models are executed using Python version 3.10. There is only one C++ version of the driver model, which implements a structure to store the data and is compiled based on the C++ 17 standard with the optimization flag '-O3'. The flag tells the compiler to use the highest level of optimizations for execution performance, allowing for increased memory consumption. While memory consumption can be also used as a performance indicator, especially on embedded hardware where memory is limited, it was not considered in this research as the simulations are generally performed on general-purpose computers with plenty of memory available. Therefore, the execution times where used to evaluate the performances. Computationally, the thesis limited the scope of research to the most basic scenarios and an available accumulation-based driver model, conducting the research on a minimal working example. It is assumed that performance differences observable on this minimal working example would be also observable in more complex setups, most likely showing more distinct differences.

1.3 Outline

First, the introduction including the background and scope is presented, followed by the theory. The theory section aims to provide a general understanding about how and why virtual simulations generally became relevant in automotive research and development processes these days. A definition of Safety Benefit Assessment as well as a selection of existing simulation platforms designed for traffic simulations and AD & ADAS V&V are provided to help create an understanding of the context of this research. Additionally, it aims to provide an understanding about how software is running on a computer system and how abstraction has and is helping humans work with computer systems. Finally, the theory section also aims to also provide a quick overview of machine code, assembly, C++ and Python to highlight how they differentiate from each other.

The third section is the Methodology, where a high-level overview of the interaction and execution of the whole simulation and its parts, namely esmini and the driver model, is provided. The aim of this section is to provide a basic understanding of what is required to run a virtual traffic simulation.As it is the main focus point of this thesis, the driver model and how the performance is assessed is discussed in detail, too. Following the methodology, the results are presented and in the included discussion subsection, the results and what their implications are is discussed. The final section, conclusion, provides a conclusion and posses further research questions. An overview of used software, libraries/packages and their version to as well as a link to the source code with all created artefacts to conduct this thesis can be found in the appendixes.

2 Theory

This section aims to provide additional information building on top of the information provided in the introduction, creating the theoretical framework necessary to understand the how and why of this research. On the 2.X level, a general context/history is provided before going into detail at the 2.X.Y level.

2.1 Pre-crash safety benefit assessment

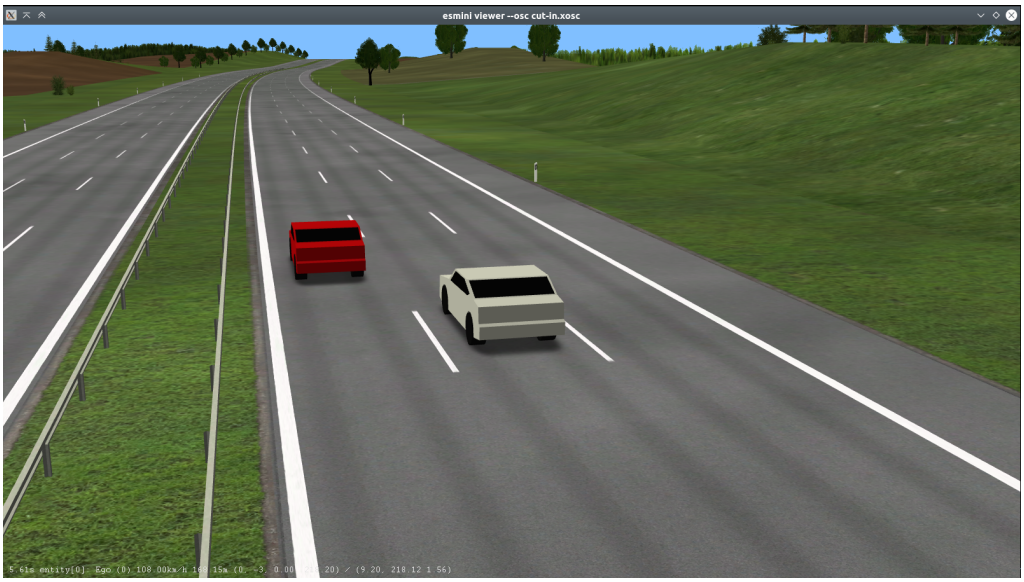


Figure 2.1: *An example virtual simulation using esmini*

For the research conducted at the Division of Vehicle Safety department at the Chalmers University of Technology, virtual simulations are used to test ADS & ADAS as well as driver models for pre-crash safety benefit assessments.

Generally speaking, simulations refer to the replication of a real-world process or systems over time for the purpose of training, entertainment, or system analysis [20]. Simulations can be physical, e.g. crash test, or digital, which is also known as computer simulations, but in both cases a model is used for the replication. A wide range of applications for simulations are found in many science fields and product development processes, but also in the form of games for entertainment purposes or very specific simulations for educational purposes. While physical simulations have and will continue to play an important role, computer simulations have taken an increasingly prevalent role in the development procedures of the automotive industry for the provided convenience, repeatability, safety and time efficiency in the form of quick iterations and easy variability provided by them [21]. All those advantages allow for the possibility of saving a substantial amount of money and speed up the development process, especially during early development stages. Additionally, new kinds of experiments can be conducted as computer simulations enable the usage of new technological possibilities [22].

Consequently, computer simulations are used in multiple disciplines and development stages in the automotive industry, such as computational fluid dynamics (CFD) for aerodynamic and internal-combustion engine (ICE) applications, or finite-element analysis (FEA) for structural and battery design applications [23]. For example, to understand how much a structural part deforms during a crash and if the deformation causes safety issues, it used to be necessary to design and manufacture a complete prototype and its manufacturing process, including the machinery and tooling. This prototype was then used in a crash test (physical simulation). With the gained insight, parts of the prototype might have been changed, and the whole process repeated, until the part fulfils its specified properties confirmed by the simulation. Naturally, this approach was time-consuming and expensive. Nowadays, such an iterative process starts digitally. The prototype is designed as a model using computer-aided design (CAD) software, and the model is then loaded into a FEA software to create a mesh model. The created mesh model is simulated with forces acting on it. After a simulation run, the FEA

software provides a complex dataset that can be used as a base to quickly optimize the design in iterations. This approach only requires some final physical testing for verification [22].

Virtual simulations are subsets of computer simulations, characterized by a virtual world the user or system-under-test interacts with [24]. As an example from this thesis, traffic simulations are virtual simulations with a virtual word (roads, traffic participants, sings, ...) and input from the user controlling an entity. The control of an entity could be automated, e.g. by using a driver model, as it the case in this thesis. As the computer simulations are based on models, they are by definition not a perfect representation of reality. The cause of this lies in not (yet) having a complete understanding of cause and effects of phenomenons, by not having the financial or technical possibilities of capturing all attributes affecting the behaviour of a real system or being limited by computational power (e.g. to reduce computational cost by removing parts of the model irrelevant for the question of interest or by using approximations for complex calculations). This leads to the results of the simulation being more of an approximation.

Therefore, the usefulness of results provided by simulations are highly dependent on the accuracy of the used model, as the models always have a level of abstraction and assumptions. The accuracy of the model, and successively the simulation, is increased by more advanced and computationally more complex models by capturing more attributes of the real system. Figure 2.2 visualizes how a process for building and improving a computer model can look like.

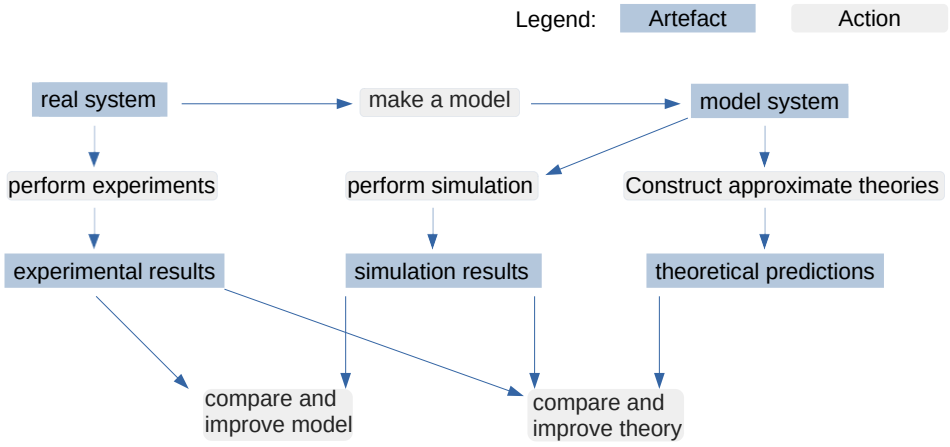


Figure 2.2: *Creation and improvement procedure of a model for simulations, based on [25]*

Hardware-in-the-Loop (HiL), Model-in-the-Loop (MiL), Software-in-the-Loop (SiL), Driver-in-the-Loop (DiL), and Vehicle-in-the-Loop (ViL) simulations complement the simulations already commonly used in the automotive development procedures or are actively researched, developed, and implemented [26].

2.1.1 Safety Benefit Assessment

ADAS and ADS, consisting of hardware and software, aim to improve road and traffic safety for all participants. During research and development as well as after production, it is of interest to be able to quantify the effectiveness of those systems using statistical methods to arrive at key performance indicators such as saved lives and reductions in injury risk. Generally speaking, there are two kinds of safety benefit assessments [27]:

- **retrospective approach**: The system is implemented on vehicles and real-life data is collected for various databases, such as naturalistic driving data or crash databases. It allows concluding the effect of the system in the real world, but takes a long time from the start of this endeavour until enough data is obtained to arrive at a well-founded conclusion.

- **prospective approach**: The assessment is performed before the system is actually implemented in the vehicle (or when there is not enough data from real traffic crashes), giving an indication of the effectiveness

of a system before its deployment on public streets and real world data is available at sufficient quantities to allow for relevant assessment.

As this thesis deals with the prospective approach, the rest of this subsection expands on the prospective approach: As mentioned in section 2.1, physical (or real-world) testing is expensive with regard to time and cost, making it used towards the end of development to test if the system is up to specifications and passes V&V. They provide a high fidelity, but it is difficult to include driver variability in the assessment, especially in cases where the vehicle is remotely controlled, and other traffic participants are replaced by dummies.

Counterfactual, and traffic simulations are prospective approaches. They are based on virtual simulations, as they consist of a virtual environment that can be interacted with running on a computer system. Counterfactual simulations explore "what-if?" scenarios. To be more precise, they recreate a scenario that has happened without the influence of the system-under-test and simulate how the system-under-test would have influenced the outcome of the recreated scenario. The P.E.A.R.S. initiative develops a methodology to harmonize the assessment of effectiveness of ADAS & ADS [5]. Counterfactual simulations utilize existing (pre-)crash data to recreate a scenario.

Examples of data sources used are the SHRP2 Naturalistic Driving Study in the United States of America, UDRIVE in Europe and CNDS in Canada [28]. Instead of counterfactual scenarios, artificial scenarios are also used for research and development purposes. Traffic simulations aim to recreate real-world traffic conditions and behaviour and focus on the interactions of the traffic participants [29]. The prospective approaches have the same benefits as general computer simulations. On the other hand, there is a need to model the virtual vehicle including the virtual system-under-test and other parts, which have an interconnected relationship as well as the virtual environment, which includes specific crash scenarios and/or human behaviour models. The result is only as good as the accuracy of the modelling itself. Those criteria can result in big and complex models and simulations, which require high performance computing. Generally speaking, higher performance benefit virtual safety benefit assessments in different ways:

- simulations can be run on less powerfull computers
- each simulation runs faster, and more simulations can be run in the same time span, which results in more data generation in the same time
- given a hardware, it allows for more complex simulations and therefore, deliver better results.
- given a hardware, the simulation can be scaled up more. This benefits especially traffic simulations, where multiple entities exist.

The increase in performance for computer simulations can be achieved by improving either the hardware that the simulation is run on or by increasing the performance of the simulation. Of course, modern simulations and their software consist of big and complex code, that can be optimized in many ways. This thesis focuses on a performance of a driver model, which could be used in counterfactual simulations.

2.1.2 Specific Tools for Virtual Safety Benefit Assessments

2.1.2.1 Environment Simulator Minimalistic (esmini)

Esmini is an open-source, multi-platform scenario player for the OpenSCENARIO standard, developed as a result from the Swedish collaborative research project Simulation Scenarios. Supported platforms include Windows, Mac, Linux as well as Android. It is written in C++, focusing on tool integration and portability. This allows the usage of esmini in native C++ applications as well as other frameworks such as Unity3D (C#) and MATLAB/Simulink and resulted in esmini being used in other applications and test platforms. It consists of the following two libraries, composed of one or multiple modules:

- esminiRMLib: This library provides an interface to OpenDRIVE files to parse road network descriptions and create the virtual world. It does not contain the full scope of OpenDRIVE.
- esminiLib: This library provides an interface to OpenSCENARIO files and parses scenario descriptions to generate entities used in the scenario and control the dynamic behaviour of non-user controlled entities. It also includes a viewer using OpenSceneGraph for a visual representation of the scenario, a gateway to centrally collect OSI data, and finally, implements ideal sensors.

Esmini provides 3D models support, comes with a few predefined controllers for various use-cases, as well as the option to connect a costume controller. It exposes an interface based on OSI to other simulation tools in various ways:

- API functions to fetch the OSI structures
- Saving to OSI trace file
- Sending serialized OSI data over TCP-IP/UDP-IP to an external program

The creators implemented an API to expose some internal functions to applications using the provided esmini libraries, e.x. stepping the scenario or creating the socket to send OSI messages [30]. The methodology section provides more detailed information about how to use esmini for custom simulations. Figure 2.3 how esmini works internally with a UML sequence diagram.

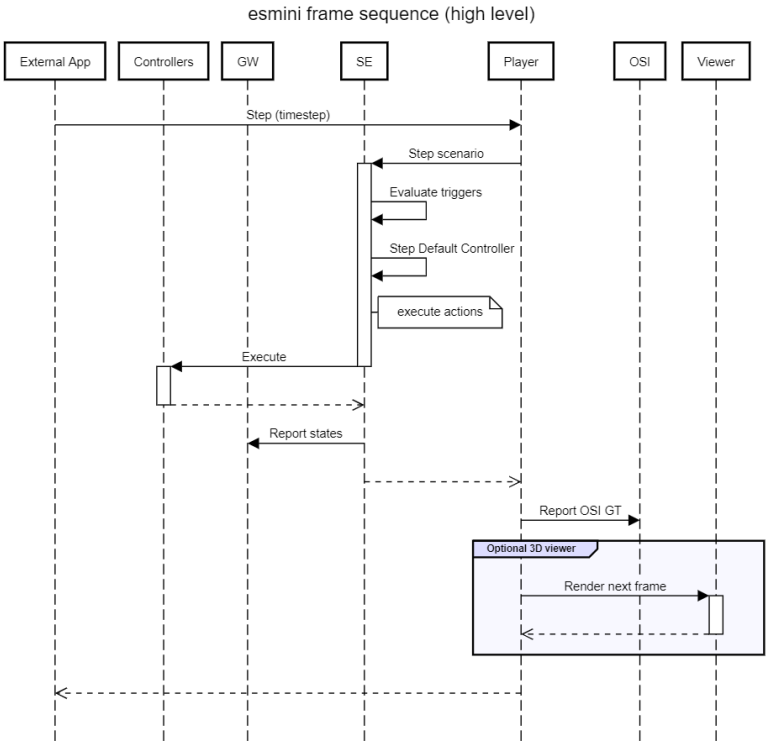


Figure 2.3: *Sequence Diagram showing the internal working of esmini [31]*

2.1.2.2 openPASS

The Eclipse Foundation's openPASS Working Group develops the open Platform for Assessment of Safety Systems (openPASS) under the Eclipse Automotive Top-Level project. Similarly to esmini, it is open-source, can simulate traffic via its SimulationCore and incorporates the OpenSCENARIO, OpenDRIVE and Open Simulation Interface standards. Additionally, it works with the Functional Mock-Up Interface (FMI) and System Structure and Parametrization (SSP) standards.

One difference to esmini is the specific aim to enable the prospective evaluation of safety systems (ADAS and ADS) through stochastic traffic simulation, event based scenario modelling or crash re-simulation. It is modular and allows the integration of self-developed modules to the pre-existing modules. This flexibility comes from its MantleAPI, allowing the exchange of scenario engines, map converters and environment simulators [32]. Figure 2.4 provides an overview about the openPASS Platform and its modules.

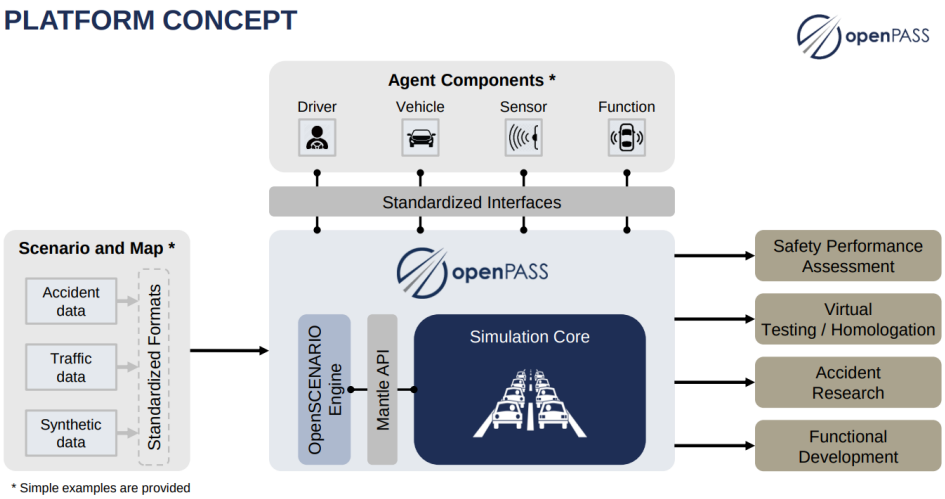


Figure 2.4: *the openPASS platform concept [33]*

2.1.2.3 Simcenter Prescan

Siemens develops and distributes the Simcenter product line, offering their own proprietary software solution for simulations in various domains, e.g. CFD for fluid and thermal applications or computer aided engineering (CAE) for mechanical ones [34].

As part of the Simcenter product line, Prescan focuses on the domain of ADAS and ADS development. Simcenter Prescan provides a full-stack simulation helping in the concept, development as well as test and validate phase. Its functionality contains a physics-based simulation platform, tools to create and modify the simulation environment including roads and the weather, sensor fusion, V2X and more. PreScan offers interfaces for control systems to design and verify algorithms for data processing, sensor fusion, decision-making, and control as well as parameter variation. The testing can range from component-level to whole traffic simulations and additional HiL, MiL, SiL, DiL, and ViL solutions. A test suite for physical testing completes the Simcenter Prescan package [35]. Simcenter Prescan uses the source-available commercial game engine Unity, which provides the physics and graphical functions.

2.1.2.4 Virtual Test Drive (VTD)

VTD is a software platform for developing ADAS & ADS systems to train, test and validate them. Many open ASAM standards (ASAM OpenCRG® , ASAM OpenDRIVE®, ASAM OpenSCENARIO®, ASAM OSI®) are utilized in the functions to create, configure and animate test scenarios in virtual environments. According to ASAM, it is the most used platform for the previously mentioned use cases [36].

It offers a complete tool chain and with its modular design and co-simulation feature its functionality can be extended. VTD includes dynamic algorithms, functions for parametrization, scripting and automation functionalities via Python as well as real-time-analytics that integrate with provided data analysis tools. Additionally, various sensor models are part of VTDs scope. Its intended use cases are MiL, SiL, DiL, HiL and ViL [37].

2.1.2.5 CarMaker

CarMaker is a commercial virtual test drive software with focus on ADAS & ADS, but also powertrains and vehicle dynamics. It is developed IPG Automotive GmbH and was first released in 1999. A real-time capable vehicle model, called virtual prototype, consists of all vehicle subsystems and is freely configurable via open interfaces. Combining the vehicle model with a driver model, traffic model and a virtual environment enables reproducible and automated tests. MovieNX enables CarMaker to represent the scenario photorealisticly and allows the testing of object detection as it is used in ADAS & ADS. The publisher highlights that CarMaker integrates seamlessly in the areas MIL, SIL, HIL and VIL. It is capable of parallel processing locally and remotely, where the included TestManager or Python API can be used to control the remote test automation. The integrated IPGControl allows for real-time data monitoring. Alternatively, the data can be stored to be used by third-party programs. Standard interfaces and standard formats, such as FMI and OpenDRIVE, are

supported and allow the integration of CarMaker into existing toolchains. A set of custom interfaces allows the usage of CarMaker as a central integration platform. Supported platforms are Linux and Windows [38][16].

2.1.3 Open Standards for Virtual Safety Benefit Assessments

The driver models are evaluated in a virtual simulation. A virtual simulation uses models to describe the characteristics/behaviour of a system, while the simulation itself represents the evolution of the model over time. In this case, the system-in-simulation is a world, which consists of a road network and traffic participants. Therefore, a need to describe the road network including traffic signs and the participants with their dynamic behaviour, as well as a need to step the simulation through time, exists. Individual companies and software developers have come up with their own, proprietary solutions for these challenges.

The issue with those are high cost, limitation of interoperable tools and that it generally makes it difficult to cooperate, based on variations in data structures, input, and output formats. This leads to slowing down development progress, especially in complex cases such as ADS & ADAS development, as they cannot easily exchange results, data, and software since they are limited by the proprietary nature of their dependencies. Thankfully, the developers and researchers have realized this and many companies and researches bodies have and continue to provide efforts in developing open-standards for simulation-based research and development to enable efficient communication and integration for a variety of simulation tools and platforms [39]. With this in mind, open-source software and open standards have been used for the purpose of this thesis and the following subsections provide an overview of used standards, which are used to describe the whole simulation environment for the driver model performance evaluation.

2.1.3.1 OpenDRIVE

OpenDRIVE is a standard published and developed by the Association for Standardization of Automation and Measuring Systems (ASAM). It defines the road network description and logic influencing entities using extensible markup language (XML) syntax and includes geometry of roads, lanes and road marks as well as static objects such as traffic signals, traffic signs and obstacles influencing the course. OpenDRIVE files have the ".xodr" file extension. The aim of the standard is to provide a road description, which can be exchanged between simulations to develop and validate ADAS & ADS features for effective representation and exchange of simulation scenarios. Well-known manufacturers are contributing to its development and make use of it in their development of ADAS & ADS [18][40].

2.1.3.2 OpenSCENARIO XML

OpenSCENARIO XML is a complementary standard to OpenDRIVE, developed by ASAM in cooperation with manufacturers. It provides the description of dynamic entities participating in the simulation and their appearance, environmental conditions as well as complex, synchronized manoeuvres that involve multiple entities in the form of traffic participants, such as vehicles and pedestrians. Similar to OpenDRIVE, it uses XML syntax and is stored in ".xosc" files. With OpenSCENARIO user-defined controllers can be implemented, and it provides reuse-mechanisms with parameters and catalogues. The focus of the standard is on facilitating the effective representation and exchange of simulation scenarios. In March 2024, ASAM released a version using domain-specific language called OpenSCENARIO DSL [19][41]. Figure 2.5 shows a high-level structure for a scenario using OpenSCENARIO.

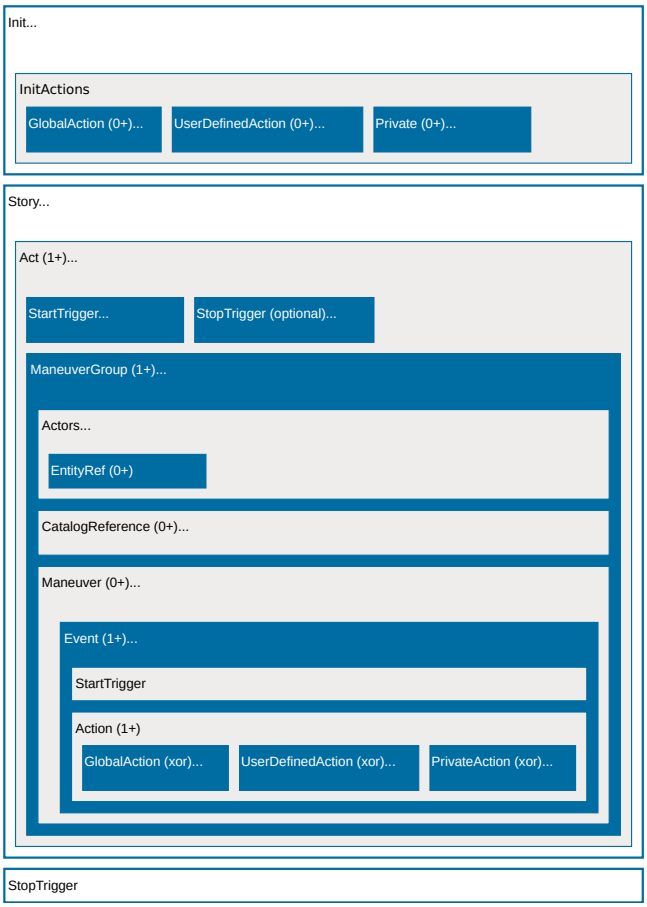


Figure 2.5: *Structure of an OpenSCENARIO scenario description [42]*

The numbers in parentheses show the minimum amount of an element needed in its encapsulating element. The structure consists of two parts:

1. Init: The definition of initial parameters using actions to describe the initial states of traffic participants
2. Story: The actual scenario description, consisting of multiple parts, nested in themselves
    - Act: Each scenario needs at least one act, even if it is empty. Its function is to regulate the story depending on time. It uses triggers to achieve this
    - ManeuverGroup: Links the entities acting out the manoeuvre by defining them as actors.
    - Maneuver: Consists of one or more events, which define what is happening by using a trigger and actions.

There are three categories of actions that are used in the scenario to describe how traffic participants behave or even directly affect the traffic participant's state:

- GlobalAction: Actions regarding non-entity-related properties, for example the weather condition
- UserDefinedAction: Actions defined by the user
- PrivateAction: Actions regarding entities, such as position and speed

In the initialization phase, the actions are used to set up the initial states, while in the later phases the actions are triggered based on events.

2.1.4 Open Simulation Interface

Like OpenSCENARIO and OpenDRIVE, the Open Simulation Interface has been developed by ASAM and companies working in the automotive sector. Based on Googles Protocol Buffers, a language and platform neutral mechanism for serialization, OSI provides a serialization for the simulation data to share it in real-time between models and its components as well as with other computers, simulation tools and/or software (Simulation Data Exchange). This is made possible by integrating the FMI and Functional Mock-Up Unit (FMU) standards into OSI.

OSI consists of three main layers:

- Application Layer: Handles the interaction of the simulation software and the OSI Interface
- Communication Layer: Handles low-level aspects of the communication channels between interconnected tools, such as details related to message formatting, data transmission protocols, and network connections.
- Tool Layer: Handles high-level aspects of the communication of interoperable tools by abstracting details of individual tools and simulation models and providing a unified interface.

OSI provides an additional example for abstraction: It hides low-level details and defines a high-level interface, allowing the developer to create interoperable software more easily and focus more on the conceptual details, rather than dealing with low-level implementation details [43][44].

2.2 Programming Languages and Abstraction

The history of the term computer and devices used to aid with computations is an interesting, but also long one. To focus on the part that is relevant for the context of this research, this section provides a simplified overview of the workings of a microprocessor built from transistors, as it is used in current day digital computer as a central processing unit (CPU). Most commonly, metal-oxide-semiconductor field-effect transistors (MOSFET) which have three terminals, namely the source, the drain and the gate, are used. By controlling the voltage at the gate, the conductivity between the drain and the source are controlled to amplify or switch electronic signals. In the context of microchips, the transistors are used as switches between two states – on and off. This enables the building of basic logic gates by combining transistors. Those logic gates are capable of Boolean functions, that is, performing logical operations on binary input.

Combining the logic gates enables to build more complex logic circuits from multiplexers, registers, arithmetic logic units all the way to microprocessors. Microprocessors are composed of at least an Arithmetic Logic Unit (ALU) and a control logic unit. The control logic unit controls the execution of the instructions. It ensures that the machine command received by the instruction is decoded by the command decoder and executed by the correct unit and the other components of the computer system. To achieve this, the command decoder uses an instruction table to translate binary machine commands into corresponding microcode, which activates the circuits required to execute the command [45].

The usage of transistors is the reason behind the need to use binary code to program microprocessors, as it directly translates to the states of the transistors. Of course, with current microprocessors having up to billions of transistors, controlling it on its lowest level is next to impossible. This, combined with the differing design of different microprocessors resulting in not interoperable code, a need for abstraction was quickly recognized.

2.2.1 Abstraction

Abstraction can be defined as managing complexities by highlighting what is relevant for the user, and cover what is not [46]. Figure 2.6 shows the level of abstractions used to program a microchip and their human readability as well as program execution speed depending on the language level. Therefore, with each layer of abstraction, the human-readability can improve, but it tends to add overhead and a performance penalty. Briefly summarized, high-level programming languages focus on manipulating conceptual features, while low-level programming languages use commands close to processor instructions.

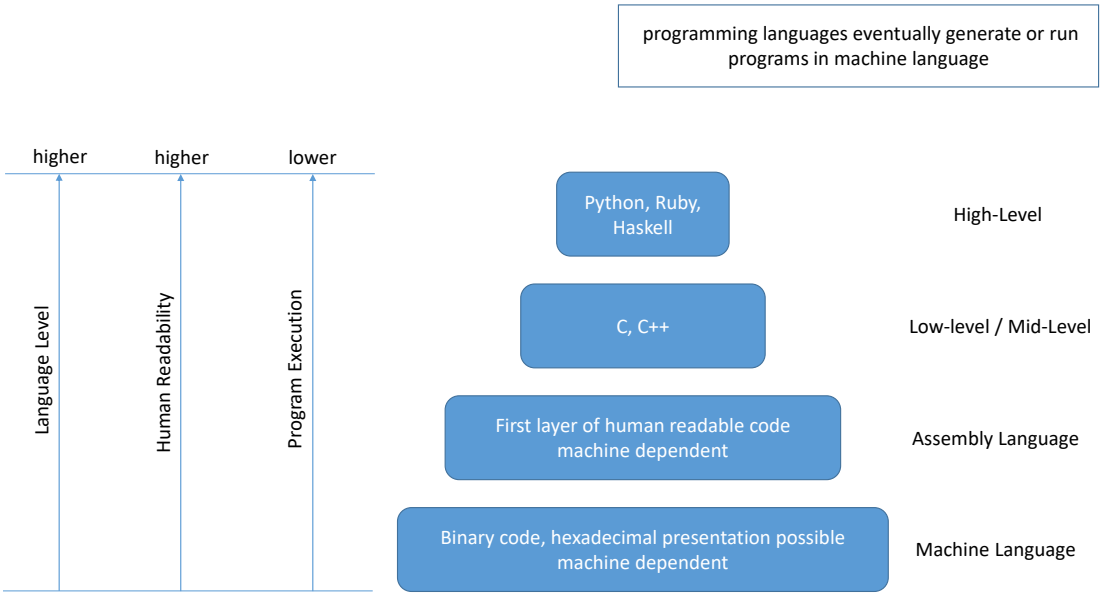


Figure 2.6: *Layers of abstraction and their trade-offs regarding performance and human-readability, based on [47]*

To provide an example, a user requires calculating the sum of 2 plus 2. The user starts the calculator application and enters the wanted calculation via the user interface. There are many levels of abstractions involved to make it that easy for the user, as the kernel takes care of all the necessary background tasks. As a more programmatic example in Python, the programmer doesn't have to deal with the implementation of I/O to get input from the user, all the programmer needs to know is the usage of the 'input' function. With benefits like these, high-level languages increase software productivity, reliability, comprehensibility, and simplicity. According to Fred Brooks, the increase in productivity is by a factor of at least five [48].

2.2.2 Machine Code and Assembly

Since CPUs are based on logic using the states on and off, or true and false, the language they "understand" is a base-2 numeral system, called binary. An instruction in this format contains the opcode and, if required, any related data. The opcode is interpreted and microcode executed by the CPU.

Microcode is usually stored in read only memory directly in the CPU for performance reasons, and it controls the signals which in turn control the flow of logic. As previously mentioned, binary code is not human-readable and the first layer of abstraction was introduced in the form of hexadecimal numbers (base-16 numeral system), making it more human-readable, while it still represents the same information. It fulfills the same purpose, as hexadecimal is easily converted to binary. Both the binary and the hexadecimal code are machine code, as they are directly understood by the CPU based on physics. Therefore, machine code has no overhead and results in the highest performance possible. Since the programmer has to deal with a pure numerical language and technical details on the lowest level (hardware), it is error-prone and tedious to work with.

Each CPU uses a specific instruction set, called instruction set architecture (ISA). An ISA defines an important aspect of the operation of a CPU, such as instructions and data types. This means, that each CPU can only work with machine code written for its ISA and portability is only given for systems using the same ISA.

Addressing the human-friendliness aspects, assembly was developed with an extremely close match between the language's instructions and the target ISA's machine code instructions [49]. Assembly allows for a less error-prone and tedious workflow using a layer of abstraction by introducing mnemonics. Mnemonics are symbolic names and represent a single opcode. The assembly code is eventually converted into machine code using the assembler. As machine code is specific to a particular computer architecture, the assembly code is also for a specific ISA. This means, that portability is fundamentally limited to the used ISA.

2.2.3 C++

In 1985 the creator of C++, Bjarne Stroustrup, released it for the first time as a super set of the programming language C to make use of classes. It is a general-purpose, multi-paradigm and compiled programming language. Compiled means that the source code written in C++ gets translated into object code before it is linked to a full program which can be executed by the system. It is a high-level programming language (offering further abstraction with classes) with low-level features (offering hardware access). In 1998, the International Organization for Standardization standardized C++ for the first time. More C++ standards have followed, and the latest standard release today is C++ 20. C++'s design had systems programming and embedded software in mind and aimed for performance and readability among others [50].

Today, C++ is among the most popular languages used in applications, where performance and resources are limited or critical for the system, such as operating systems, embedded systems, search engines, and many more[51]. The automotive industry makes strong use of C++ for embedded systems.

2.2.4 Python

Guido van Rossum, the creator of Python, published it for the first time in 1991. Python 2.0, released in 2000 and discontinued in 2020, was the successor. Today, the in 2008 released Python 3 and its iterations are the stable and maintained releases. This thesis makes use of Python 3.10 and its features. Python is a general-purpose, multi-paradigm language with emphasis on code readability [52].

CPython, the reference implementation of Python, translates the source code into intermediate byte-code, which is platform-independent code and differs from machine code. The byte code is then interpreted by the Python virtual machine line-by-line. This way, Python code can be run platform independently and has advantages in debugging processes, as it is easier to identify the source of the error but also lowers performance compared to not interpreted languages. Figure 2.7 shows the execution steps of Python source code using the reference implementation Python interpreter.

Python uses whitespace indentation, English keywords and omits semicolons after statements, resulting in better readability compared to other languages such as Java and C++. This results in the semantic structure being represented by the program's visual structure [53] and a very simple and consistent syntax, which makes it beginner-friendly. Additionally, the provided large standard library helps in focusing on the problem, rather than on the semantics and implementation of certain functions. Python is consistently ranked in the top 10 programming languages in regard to popularity [54] and the most popular programming language in 2021 [55]. It is especially popular in machine-learning and for rapid-prototyping, based on the large community and a variety of existing, specifically tailored for machine learning libraries. These influential factors are amplified by the flexibility and interoperability offered by Python.

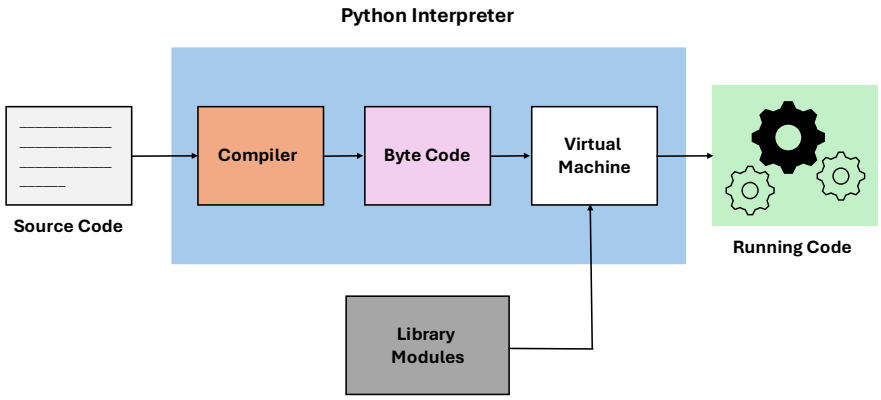


Figure 2.7: *The execution of Python source code using the reference implementation Python Interpreter*

2.3 Driver Model

The purpose of a driver model is to replicate human behaviour while controlling a vehicle and are used to develop a better understanding of driver behaviour, interactions with other traffic participants and vehicle control. Driver models are an important part of simulations for ADAS & ADS development and focus on specific aspects of driving are not general-purpose.

For example, the intelligent driver model (IDM) by Treiber et al. [56] is an agent-based single-lane car following model with a predefined maximum deceleration. On the other hand, Kusano and Gabler [57] assessed the safety benefits of a precollision system (PCS) components for forward collision warning (FCW), precrash brake assist (PBA), and autonomous emergency braking (AEB). The PCS is based on Time-To-Collision (TTC) and tries to minimize ΔV, which is the change of velocity of a vehicle during a crash.

Another approach based on ecological psychology and driven by perceptual indicators is used by Makkula to provide a computational framework that aligns well with the theory of predictive processing, which states that human behaviour can be regarded as prediction minimization. The framework combines the neuroscientific concepts of evidence accumulation, sensory prediction and motor primitives. Evidence accumulation describes a cognitive process for decision-making on the level of neurons. Noisy sensory input is sequentially sampled until enough evidence to make a decision has accumulated. Control intermittency is a concept to describe sustained human motor control. Sensory prediction describes how the brain predicts the sensory feedback, originating from an action taken or successive events occurring, and plays an important role in human behaviour. Motor primitives describe how the brain uses basic movements to create complex motor behaviour. The underlying idea is that the brain organizes movement in modular and reusable patterns, which it adjusts depending on the conditions [58].

The driver model implemented in the simulation is based on previous work by Svärd et al. [59] [60], which is based on the described framework and uses looming as a visual indicator. Looming is the optical size of an object, in this case a lead vehicle, and its expansion when the distance to the object decreases. Visual indicators are also used in the non-psychological based approach, but the way they are used differ. In the non-psychological based approach, they are used as a threshold, initiating actions based on the crossing of the threshold. The approach used by Svärd et al. is based on evidence accumulation over time.

3 Methodology

This section covers the approach chosen to research the performance of the driver models and the wrapper. Starting with an overview of the whole simulation setup, each component used in the setup is presented. This includes a description of their functionalities and how they have been implemented. Successively, how the benchmarking has been performed and evaluated is explained.

3.1 Simulation Setup

Table 3.1: Information about the used computer

Name of System Part	Used
Operating System	PopOS_!20.04
CPU	Intel® Core i7-4800MQ
GPU	Intel® HD Graphics 4600
RAM	16 GB

Initially, it was decided to limit the hardware to one specific personal computer using an i7-4800QM (x86-architecture) CPU, 16 GB RAM and the integrated graphic chip on the CPU. As the OS, PopOS_! 20.04 was used. PopOS_! is a Linux-distribution based on Ubuntu, which in turn is based on Debian. Those two limitations allowed to limit variability of performance-influencing factors. Further efforts to limit the performance-influencing factors were made by using one version of Python (3.10), the C++ Standard (14) and compiler (gcc 9.4.0) with optimization parameter -O3, as well as keeping the version of the standards and simulation software consistent during the research. Assuming the simulation software performance is constant, only the driver models themselves should have affected the performance benchmarking.

The aim for the simulation setup was to create a minimal working example. A minimal working example is usually used to demonstrate a bug or problem, but in this thesis it refers to an example implementation, which contains the core functionality to perform a virtual simulation without additional complexity. From the start of the thesis, the usage of esmini as the simulation software was determined, since it is open-source and utilizes open standards to describe the static and dynamic aspects of scenarios. It was also a given to utilize the OSI functionality of esmini for data exchange, allowing a modular simulation setup instead of a monolithic one. Two wrappers for esmini were written, one using C++ and one using Python. The Python wrapper was written to see if a completely Python based simulation setup is feasible. With working esmini wrappers, a scenario needed to be described to be used in the simulation. Using the OpenDRIVE and OpenSCENARIO standards, a cut-in scenario was described. At this point, a simulation scenario could be simulated, but the ego vehicle still needed a driver model to control its actions. For this, a driver model based on the theory and previous work described in section 2.3 was implemented using C++ and Python. No language-specific optimizations were performed, and the code in both languages was written to match line by line as much as possible. To see if the choice of data structure has a noteworthy impact, four Python driver models using different data structures to store the relevant OSI data were written.

Figure 3.1 visualizes the interaction of all selected parts abstracted, and table 3.1 presents the information of the used system compactly.

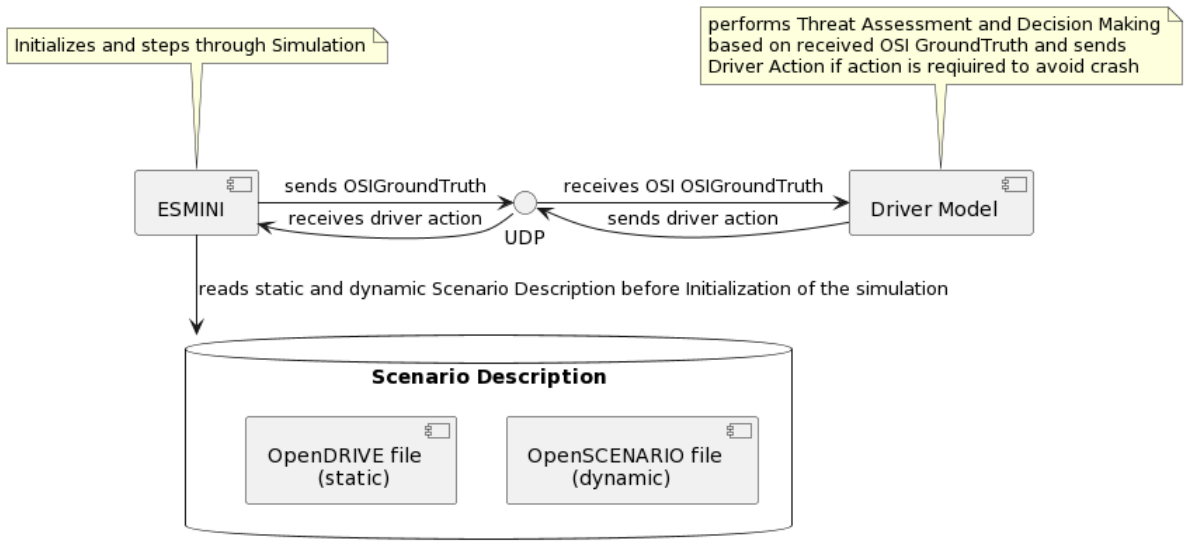


Figure 3.1: *Files and software used for the research*

3.1.1 esmini

Esmini is packaged with some scenarios and example application, which did not fit the needs of the benchmark. Therefore, simple wrappers for the included libesmini.so shared library were written in C++ and Python to utilize the esmini API. For the core functionality of initiating a scenario and stepping the scenario while using OSI, only three API function calls were required:

- SE_InitWithArgs: Initialize the scenario by passing the path to an OpenSCENARIO file
- SE_OpenOSISocket: creates socket that then sends OSI GroundTruth messages
- SE_StepDT: The simulation is stepped by the provided time step

The wrappers contained argument parsing in their functionality of the code to enable the user to change the scenario, the time step, choosing the benchmarking mode, changing the output file name of the benchmark and to toggle the headless mode via command line arguments. Headless in this context means without a visual representation, except textual updates in the standard output of the shell sharing information about the current state of the simulation.

To be able to use the esmini API in the Python wrapper, the ctypes module had to be imported. Using the function "ctypes.CDLL()", the location of the shared library to use was provided and a handle created. Using this handle, all API functions could be called from inside Python. As the shared library was written in C++, and therefore, used static typing, the return types and argument types of the used API functions needed to be defined. For this, ctypes provides classes to use as C data types.

3.1.2 Scenario

To test the driver model in a safety-critical scenario, a cut-in situation was chosen. The ego vehicle accelerated on its lane to full speed and aimed to keep that speed as long as no interference that affected its safety on its path was perceived by the driver model. On its left lane, the target vehicle drove with a fixed speed, which was substantially lower than the top speed of the ego vehicle, and would initiate a cut-in into the lane of the ego vehicle when the distance between both vehicles was lower than a defined distance. Since the ego vehicle accelerated the same way from the start of the simulation, the specified distance between both vehicles was always reached at the same time step and thus, the scenario was repeatable.

For the scenario definition, a OpenDRIVE file was written, which contained the definition of the static parts of this scenario. It contained the definition of a road with two lanes per side and the corresponding lane markings. As there is a maximum length that can be assigned to the roads and one road was not long enough for the whole manoeuvre, two roads were defined and connected to each other. The dynamic properties of

the traffic participants were defined in a OpenSCENARIO file, which contained the property definitions of the vehicles, such as full speed and acceleration amongst others. Here, vehicles predefined in the catalogue provided by the esmini software package were chosen.

The definition of the vehicles also contained the controller to be used. In the case of the ego vehicle, the external controller defined in the controller catalogue was selected and configured via parameters. The configuration included the selection between synchronous or asynchronous modes and the port to be used, which corresponded to the port that the driver model used to send data to. For the target vehicle, the esmini default controller was used, which followed the dynamics defined for the target vehicle in the OpenSCNEARIO file.

The definition of the traffic participants was followed by the initialization of them at the start of the scenario. This included the position of the traffic participants and their velocity. In the case of the ego vehicle, the controller had to be activated instead of defining its velocity, as that vehicle was controlled via the external controller. Finally, the dynamic evolution of the scenario itself was defined by defining two manoeuvres contained in acts. The first one was active from the start of the scenario and just kept the target vehicle at a fixed speed. The second one was activated when the distance between both vehicles were under a certain threshold. Then, the target vehicle would start to cut-in from the left lane to the lane in which the ego vehicle was moving. The scenario would stop after a timer defined for the second act ran out, which then ended the simulation.

3.1.3 Driver Model

This subsection provides language-independent information about the implementation of the driver models with images using C++-style syntax. For more theoretical details of the driver model, refer to the paper by Svärd et al [59][60].

While the main question was how the performance of C++ and Python implementations differ, the author was also interested if there are any observable differences when different data structures are used in the same programming language. Therefore, additionally to one C++ driver model, four different Python driver model variants have been implemented, only differentiating in the data structure used internally by the driver model to store the relevant GroundTruth data from the received message. These were:

- Class: The standard class used by Python.

- Data class: First described in PEP 557 and introduced in Python 3.7. They use standard class definition syntax and can be used with inheritance and other Python class features, but differentiate by implementing special methods such as __init__() and __repr__(). Type annotations are used to define data classes. In this implementation, the parameter "slots" is set to true, which prohibits the instantiated objects from being extended, thus saving some overhead.

- Dictionary: The standard dictionary implementation of Python.

- List: The default list implementation of Python, representing a basic array.

For the C++ implementation, a structure was used to store the relevant GroundTruth information processed by the driver model. The driver models were class based, separating the threat-assessment and decision-making functionality in a class referred to as the main driver model class and the User Datagram Protocol (UDP) data exchange functions in a class referred to as UDP class. The UDP class was integrated in the main driver model class as a composition to provide the data exchange functions to the main driver model class. Figure 3.2 shows a UML class diagram including the most important functions and properties of the complete driver model.

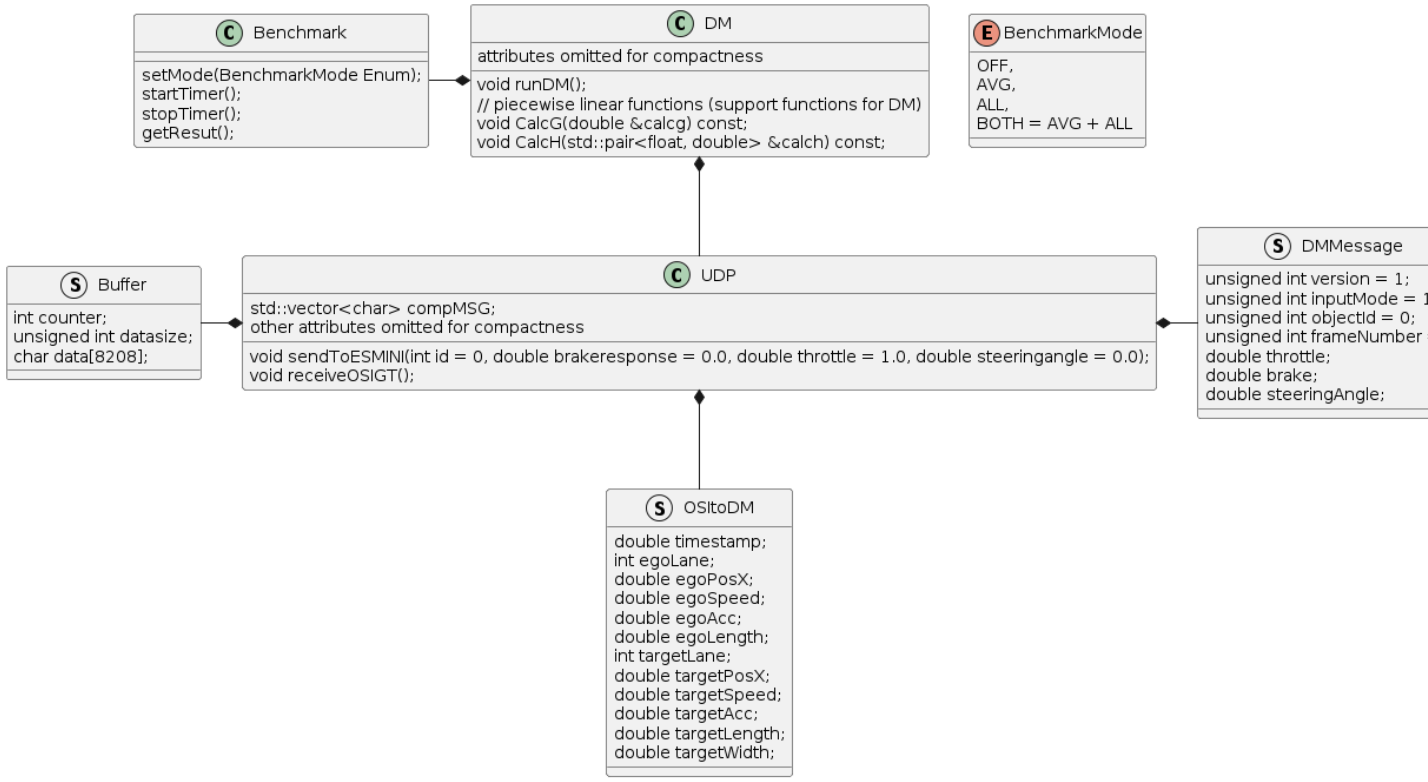


Figure 3.2: *The structure of the driver model*

3.1.3.1 UDP Class

The UDP class were constructed of multiple parts and provides two functions, one to send vehicle control actions and one to receive OSI GroundTruth data between the driver model and simulation respectively. The parts the UDP class consisted of data structures for a buffer, the data decoded from the serialized GroundTruth in the form of a class object and a data structure to package the response of the driver model.

First, a connection to the simulation needed to be created. For this, a UDP socket was used, as it was the only option for synchronous connection at the start of this work. UDP sockets are connectionless sockets and exchange datagrams. They are primarily used for asynchronous functionality, but can be used for synchronous functionality as well. In the latter case, the users have to implement those themselves. The created UDP socket was used for both, ingoing and outgoing datagrams between the driver model and the simulation. As the data received from the simulation is serialized, the receiver needs a way to deserialize the binary data back into language-objects. For this purpose, the OSI libraries were used by the UDP class, as it contains the definition of the OSI GroundTruth sent by esmini and functions to deserialize a complete message.

There were two points to consider when receiving the serialized data before the OSI deserializing functionality could be used. Since the received datagrams have a maximum size (in this case 8208 bytes, 8192 bytes data and 16 bytes for the header, which consists of two integers), one OSI GroundTruth message can be split across multiple datagrams. This necessitated a way to put multiple datagrams together to form a complete GroundTruth message at the receiving side. The other point to consider was to have a check to make sure that no data was lost on the way. Both points have been handled by integrating those functionalities from an example Python script kindly provided by the authors of esmini in the software package. The Buffer structure stored the received serialized data in three properties:

- counter: Allowed to deduct if a message is a complete GroundTruth or a split GroundTruth. In the case of split GroundTruth, it also allowed to check the order of received messages, where a negative number would indicate the last part of a message. Extracted from the header of the received datagram.
- data size: Contained the information about the actual size of the received GroundTruth data, extracted from the header of the received datagram.

- data: A character array with the pre-allocated size of 8208 bytes. The data of the received datagram without the header was stored here.

In the case of split GroundTruth, the received packages were stored in an array (called vector in C++), until a negative counter indicated the last part of the GroundTruth was received. After the last part of the GroundTruth was received and stored in the array, the deserializing function of the OSI library "ParseFromArray" was used to convert the complete serialized message to a human-readable format. Data that would be processed by the driver model main functionality was stored in a OSItoDM object.

By default, the driver model would populate the response data structure with full acceleration value and no brake value before sending it in serialized form to the simulation. If the driver model deemed a brake action as necessary, it would instead set the acceleration value to zero and the brake value to a value it determined to be necessary to avoid a crash.

3.1.3.2 main driver model class

This section provides insight about how the accumulator driver model works. To respect the intellectual property, only the physical aspects will be explained in detail, while the other aspects will be described abstractly. Figure 3.3 presents a schematic view of the working of the complete driver model.

The driver model class was written to provide argument parsing, a logging function as well as a function to evaluate the OSI GroundTruth and perform the threat assessment and decision-making. To be able to perform the threat-assessment and decision-making, the piece-wise linear functions H(t) and G(t) were realized. The accumulator driver model used the distance between the two vehicles and calculated the optical size of the target vehicle on the retina based on the formula (3.1).

$$\theta = 2 * \arctan \frac{widthOfTargetVeh}{2 * Distance} \tag{3.1}$$

The calculated value was stored to be used in the next step to calculate the change in optical size. To obtain the change of optical size on the retina, the difference between the current optical size and the optical size of the previous time step were divided by the time step according to formula 3.2.

$$\dot{\theta} = \frac{\theta - prev\theta}{timestep} \tag{3.2}$$

To be able to perform this calculation in a sensible way, the previous optical size was initiated as infinity in the first call of the threat-assessment and decision-making function. The expansion rate of the optical size, also known as looming, was computed as

$$\tau^{-1} = \frac{\dot{\theta}}{\theta} \tag{3.3}$$

The predicted looming, which is calculated at the end of each run of the driver model function, is subtracted from the calculated looming value to determine the prediction error.

$$\varepsilon = \tau^{-1} - \tau_{pred}^{-1} \tag{3.4}$$

This prediction error was used in an accumulator to determine if the breaking model should be activated. When the accumulator was over a set threshold, the breaking model was activated and new values for the piece-wise linear functions were added to their respective arrays in every loop the accumulator was over the threshold. After the brake model was activated, the piece-wise linear functions iterated over their respective arrays as their input, a brake response was calculated, and a brake delay was increased. The moment the brake delay counter passed a set threshold, the current brake response current brake response was sent to esmini. At the end of the function, the looming prediction for the next cycle was calculated based on a weighted prediction error history.

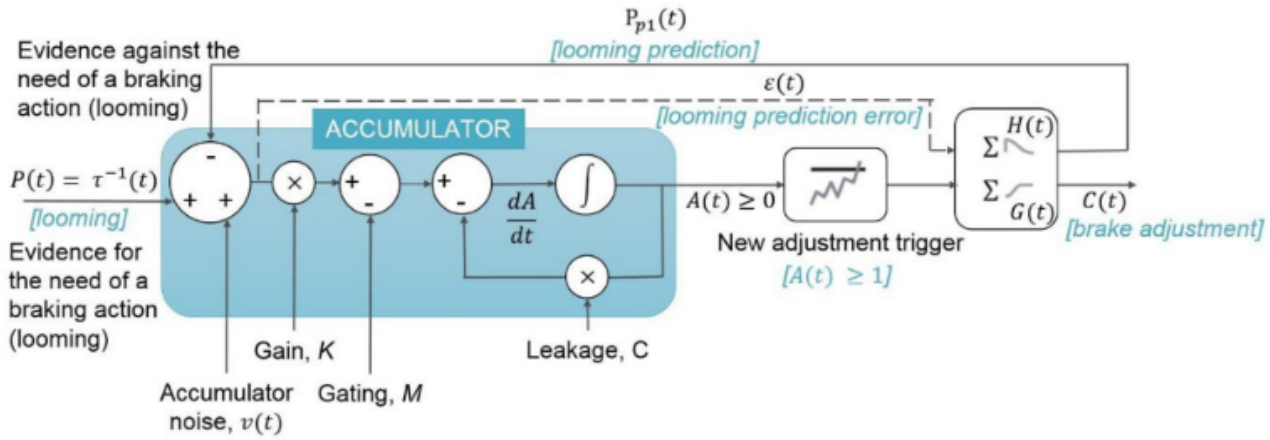


Figure 3.3: *Schema of the way the accumulator driver model works [60]*

3.2 Benchmark

To be able to evaluate the performance, execution times of the driver model and esmini wrapper were of interest. In order to get a detailed insight of the performance differences, two different execution times were measured for each part of the simulation setup.

One execution time measured was the execution time of the threat-assessment and decision-making of the driver model and the simulation stepping function of the esmini wrapper for each call in a simulation run, looking at an isolated part of the whole execution chain. This was done in order to have a purely language-dependent performance difference with limited external influences from the state of the system and operating system influences. A separate class was written for this, referred to as custom-made benchmark.

Another execution time was measured using the GNU time command, which is built-in in many Linux distributions. The measured execution time covered the whole execution chain, from the start of the execution to its termination. This way, the effects of all driver model external factors have been captured. Using the GNU time command provided statistics about the real-world time passed, the time the process used the CPU actively in user space and system space (also referred to as Kernel space).

The simulation setup was run in different configurations, referred to as benchmark configuration. The idea behind this effort was to see effects of different parameters on the execution time. The following subsections go into more detail of the the benchmark configurations, the custom-made benchmark and the GNU time command.

3.2.1 Benchmark configurations

Each of the five driver models was benched in four modes:

- visual: Esmini visualized the simulation in 3D. The driver model performed its standard functionality without logging.
- visual with logging: same as the visual mode with driver model logging activated, capturing the influences of an additional external system call and I/O operations.
- non-visual: Esmini ran headless, meaning without visualizing the simulation. The driver model performed its standard functionality without logging.
- non-visual with logging: same as non-visual mode with driver model logging activated, capturing of an additional external system call and I/O operations.

The idea behind those configurations is to vary parameters of the simulation setup and get some degree of control over the context in which the execution times were measured.

Measuring the driver model function in the headless benchmark configurations provided a lightweight, but almost purely language dependent performance measurement. No high-level system calls like printing to the

shell or getting the current time from the operating system were invoked. The measurements still included system calls that the programming language needed to run its functionality, for example, managing memory in the case of the C++ driver model.

Toggling logging intended to capture any noteworthy difference in execution times caused by a system call invoking an I/O operation. Since the computational cost of the driver models' main function is quite low for a modern day processor, a system call would be expected to show impact. In these two benchmark configurations, the simulation caused minimal load on the CPU.

The same effects were tried to be observed, but in a system state with higher load, caused by the simulation setup. For this, the visualization of the minimal simulation scenario increased the load on the CPU and the driver model performance was measured in a more uncertain system state, with increased probability of system calls and context switches. While the increased load was still minimal in the context of the available total CPU performance, it represents a use case which could be useful for driver model and driver behaviour research by abstracting the data yielded during the simulation and presenting the scenario simulated visually.

A bash script was used to loop each benchmark configuration with combination of driver model and esmini wrapper a hundred times and perform repeated measurements. Figure 3.4 shows the effect of the benchmark configuration on the esmini wrapper and simulation variant, as well as showing the relation of the past simulation world time to real-world time in headless and visual mode of esmini. It also shows the increased CPU load caused by the simulation setup.

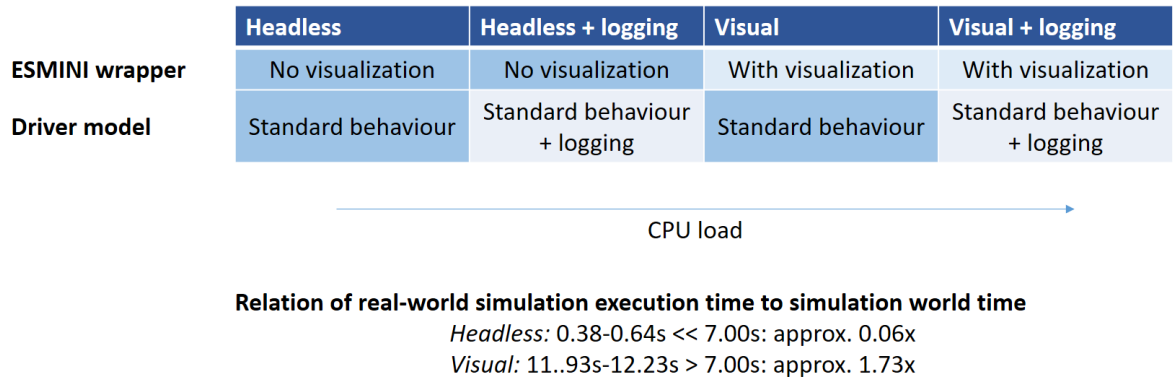


Figure 3.4: *The behaviour of the simulation setup components in each benchmark mode and the relation of real-world execution time to simulation-world time: headless simulation were approximately 17x faster than real-time, while simulations including visualization took almost twice as long to execute as real-time.*

3.2.2 Custom-made benchmark

The custom-made benchmark measured the time between the start of the function call and the moment it returned, effectively measuring the execution time of the driver models' threat assessment and decision-making or the esmini wrappers' simulation stepping. This benchmark was in both cases, Python and C++, defined as a separate class and initiated as an independent object. The core functionality consisted of getting an unreferenced time value by using the time (Python) or chronos (C++) libraries and calculating the difference to a later retrieved time value, which represented the duration and therefore, the execution time. The class contained the following methods:

- setMode: select one of the different modes to benchmark
- getResults: Based on set mode, getResults displays the average of all measurements and/or creates a file with all measured execution times

The times measured by f the custom-made benchmark were suspect to outliers caused by the scheduler of the operating system and the calculated averages of the custom-made benchmark were disregarded. Instead, as described in section 3.3, the stored times of each loop of a simulation run were filtered and the average calculated based on the filtered times.

Table 3.2: The different modes offered by the benchmark class

mode	description
OFF	No benchmarking was performed
AVG	For each loop, a start time value and an end time value was retrieved and their difference stored in an array, which was used in the getResults method to calculate and display the average of all measurements
ALL	For each loop, a start time value and an end time value was retrieved and their difference stored in an array, which was used in the getResults to create a file with all the measured times
BOTH	performs both, AVG and ALL

3.2.3 GNU time command

Additional measurements were performed using the Linux built-in GNU time command, which is used by providing the program to be measured as an argument (including the arguments the program should be run with) and arguments to specify the output format. While the custom-made benchmarks look at a specific sequence, the time command was used to provide data of the complete execution chain of the simulation setup in a dynamic system, based on the multitasking nature of the operating system. As some shells, such as bash used in this work, have their own time command implementation, it is important to note that the GNU time command was used. This was done by calling "\time" in the bash script, as without the backslash, bash would have used its own time command. It was used on both, the driver model and the simulation software, yielding the statics for both presented in Table 3.3.

Table 3.3: Overview of what is measured by the time command

GNU time measurement	description
real time	the real-world time passed between the start of a process and it's termination
user time	The time the CPU spent on executing the programs' functions in user-space, like arithmetic calculations
system time	The time the CPU spent on executing operating system functions invoked by the program (aka system calls), like using a socket and I/O

In an ideal system with only one core, the sum of the user and system time should equal the elapsed real time. Since modern CPUs are multicore, a sum bigger than the elapsed total time indicates the usages of multiple cores, while a sum less than the elapsed time indicates that the process spent time waiting.

As the GNU time commands measurements were only printed to the shell using standard error, they had to be written to a file using the >> appending redirection operator in the bash script. The file descriptors are referenced by numbers, where standard output is file descriptor 1 and standard error is file descriptor 2. There are more file descriptors, but those two were the only ones needed for the task on hand. Using redirection to write to a file didn't affect the performance of the simulation setup, as the operations for the redirection are executed before the process measured is invoked and the timer started. The listings 3.1 shows an example code of the bash scripts used to run the simulation in one configuration once. In this case, the C++ driver model with logging toggled on combined with the Python esmini wrapper running in visual mode. In appendix C. more bash scripts are shown as they were combined to run all driver models in all benchmark configurations the same amount of times by calling just one bash script.

```
#!/bin/bash
{ \time -f "%e, %U, %S " python ../Simulation/scenarioStart.py -v -b 2 -m VL-PyCls 1>/dev/
    null ; }  2>&1 | cut -d"y" -f2 >> ../output/timeEMPyVL-PyCls.csv &
{ \time -f "%e, %U, %S " python DMCls.py -b 2 -n 648 -l -m VL-Py ; } 2>> ../output/
    timePyClassVL-Py.csv
```

Listing 3.1: Example bash script starting the simulation and DM parallel via two separate calls of the "time" command and redirecting output to files

Esmini always communicates updates about the simulation in the way as described in the explanation of headless mode. Those updates would have ended up in the files containing the measurements, so esminis standard output was redirected to /dev/null to avoid the updates ending up in the measurement files. /dev/null is a null device file in Linux, which discards anything redirected to it.

As the GNU time command uses the standard error for its output, it was redirected to file in the case of the driver model. The appending redirection operator would create the file to write to if it didn't exist and append to it runs where it already existed.

In the case of using the GNU time command with esmini, some extra steps were performed. First, standard error was redirected to standard output, so that it could be piped to another command. As the redirection was to a file descriptor, `>&` had to be used. The piping operator | followed the redirection to send the output generated to the cut command as input. This was necessary because esmini would print a warning message ("Invalid MIT-MAGIC-COOKIE-1 key") to standard error when run in visual mode on this setup. Using the cut command, the warning message was removed and only the GNU time command output was written to the file by redirecting the output of the cut command to the file.

All driver models were combined with both versions of the esmini wrapper and benchmarked in each benchmark configuration 100 times. Each combination yielded 64800 individual execution times as well as 100 executions times from start to finish via the GNU time command for each, the driver model and the simulation. Figure 3.5 visualizes where in the execution chain the custom-made benchmark performed measurements as well as the GNU time commands' measurement during the whole process execution.

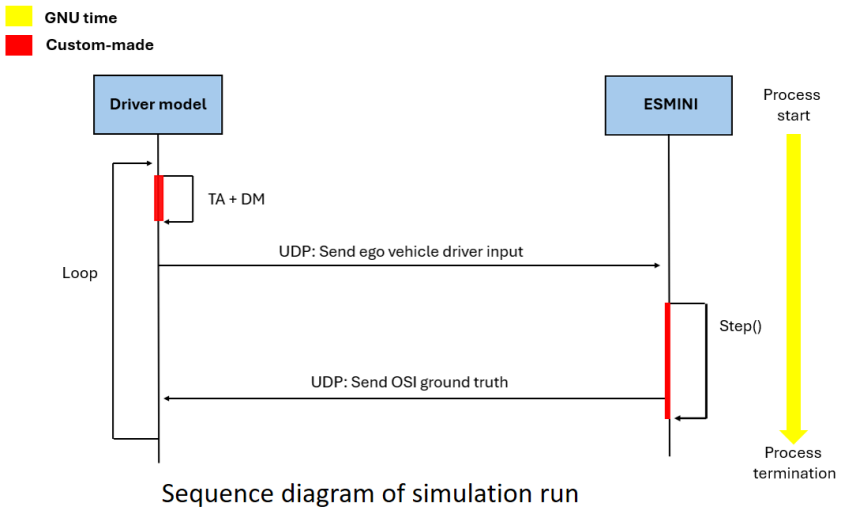


Figure 3.5: *Sequence diagram of the simulation setup. The custom-made benchmark measured the execution times of the TA + DM function as well as the Step function. The GNU time measured the complete execution from start to termination.*

3.3 Evaluation of results

The results of the benchmark were stored as individual files in a common location. For the evaluation of the benchmark, the files have been processed in two steps:

- 1. Filter outlier

- 2. Create histograms and tables

The need to filter was caused by the scheduler of the multitasking operating system. In order to use the resources of a CPU with many processes, a technique called context switching is used. Depending on the schedulers' prioritization of processes and interrupts, the state of a currently running process is saved and another process loaded. The saved process will continue executing when the scheduler eventually loads it again. This sometimes happened to the benchmark, where a context switch occurred before the end of the loop. The CPU would be busy with other processes and more time would pass before the benchmark process was loaded again and got the end time value, resulting in a small percentage of outliers with more than 150 times of the average calculated in the results 4. As a measure of limiting this percentage, the computer was restarted and a fresh system state with minimal running processes of the operating system was created. This led to the CPU load during the measurement of the execution times consisting of operations to handle all operating system functionality, additional scripts loaded during the operating system start-up process and the two software components of the simulation setup. This system state was intended as the starting point of the benchmark process, trying to minimize the frequency and impact of potential context switches leading to outlier.

3.3.1 Filtering

The first script used the interquartile range (IQR) to determine the outlier. It iterated all files stored in the result folder and loaded the data stored in them. Based on the name of the file, files created by the custom-made benchmark were identified and their data sorted. The IQR is defined as:

$$IQR = Q3 - Q1 \tag{3.5}$$

Q1 and Q3 are the lower and upper quartiles which are defined as the median of the first/last half of the sorted data, respectively. With those three parameters, the lower and lower limits for outliers are:

$$lowerlimit = Q1 - 1.5IQR \tag{3.6}$$

$$upperlimit : Q3 + 1.5IQR \tag{3.7}$$

Values not in between those limits were discarded and filtered data was stored in new folder with the same name. Files created for the GNU time command where saved without filtering, as they were not affected the same way based on the scale the values were measured.

3.3.2 Histogram and Tables

For the creation of histograms and tables all data needed to be accessible. The files inside the filtered data folder were looped and cached as individual runs as well as appended to an object collecting all execution times in one array based on simulation variant and bench configuration. After looping all the files, the prepared data was used to calculate the averages for each driver model variant. The sum of all execution times for a given esmini wrapper and bench configuration was divided by the number of execution times and written to a CSV file, formatted as seen as in the tables in 4.

For the creation of the histograms, the used global edges were determined by finding the lowest and highest values for a given esmini wrapper and bench configuration. This was done in multiple steps. First, the lowest and highest times for each configuration per driver model were cached during the data preparation loop. Then, the lowest and highest values between the five edge values of the different driver model variants were found and used.

With the edges defined, the prepared data was looped. First, the histogram for all runs of a driver model variant for given simulation variant and bench configuration was created using the matplotlib.pyplot hist function with 100 as the input for the number of bins. Afterwards, labels and text were added, and the plot saved to disk as a PDF file. This was followed by looping the individual simulation runs of the driver model variant for a given esmini wrapper and bench configuration combination and creating histograms in the same manner as before. As the number of data is drastically lower, the number of bins was reduced to 10. The tables and histograms are presented in section 4 and the script for their creation is found under Scripts/calculateAverages.py in the link provided in C.

4 Results

This section contains the results of the benchmarks performed for each driver model with one of two esmini wrappers in one of four different benchmark configurations, presented as table or histograms. In the case of tables, table 4.1 shows an abstract view of the data presented to help understand how to read them. In the top left corner, the used benchmark tool and the measurement unit are shown. The cells below indicate for which driver model and esmini wrapper combination the results in the two rows to their right are presented. Here, 'sim' is used to indicate the row for esmini wrapper and 'dm' for the driver model results (e.g., only measuring the execution time of the driver model).

The headers divide the row for this combination according to the used configuration and present either the custom-made benchmark or GNU time command results. To make it easier to compare if a driver model was affected by the choice of esmini wrapper, the order of presenting in row two of a table is switched, so that the results of the driver model are presented back to back.

The first subsection provides tables summarizing the average execution times of all individual loops performed during a hundred simulation runs using the custom-made benchmark function. Also using tables, the second subsection offers an overview of the average results measured using the Linux built-in GNU time command during the same runs. For an explanation of the different statistics provided by the GNU time command, refer to 3.3. This is followed by the third subsection, displaying histograms generated based on every driver model function iteration recorded during the data collection.

Table 4.1: abstract view of the data presented in tables

benchmark tool [unit]	**benchmark configuration**
sim **C++ esmini wrapper**	result for C++ esmini wrapper
dm **measured driver model**	result for measured driver model combined with C++ esmini wrapper
dm **measured driver model**	result for measured driver model combined with Python esmini wrapper
sim **Python esmini wrapper**	result for Python esmini wrapper

4.1 Execution times measured with custom-made benchmark

Each cell presents the average execution time for the driver model threat-assessment and decision-making function, as well as the execution time for stepping the simulation into the next time step, calculated from data collected during 100 simulation runs. The unit of the time values are milliseconds [µs].

Table 4.2: The average execution time for a loop of the C++ driver model measured for a combination of simulation versions and benchmark variations

custom-made benchmark [µs]	**headless**	**headless+log**	**visual**	**visual+log**
sim **C++**	80.34	84.002	16634.10	16628.67
dm **C++**	0.17	7.08	2.05	44.26
dm **C++**	0.16	7.13	2.20	47.81
sim **Python**	81.88	83.74	16633.50	16633.54

Table 4.3: The average execution time for a loop of the Python Class driver model measured for a combination of simulation versions and benchmark variations

custom-made benchmark [µs]	**headless**	**headless+log**	**visual**	**visual+log**
sim **C++**	159.46	178.773	16629.31	16629.35
dm **Python Class**	16.39	30.34	73.47	113.10
dm **Python Class**	16.24	31.34	78.65	122.17
sim **Python**	156.20	179.07	16636.07	16636.56

Table 4.4: The average execution time for a loop of the Python data class driver model measured for a combination of simulation versions and benchmark variations

custom-made benchmark [µs]	**headless**	**headless+log**	**visual**	**visual+log**
sim **C++**	155.95	178.325	16629.74	16630.37
dm **Python Data class**	15.91	30.43	74.27	113.73
dm **Python Data class**	16.30	30.20	79.98	123.88
sim **Python**	155.69	176.09	16634.68	16636.63

Table 4.5: The average execution time for a loop of the Python Dictionary driver model measured for a combination of simulation versions and benchmark variations

custom-made benchmark [µs]	**headless**	**headless+log**	**visual**	**visual+log**
sim **C++**	162.60	182.552	16625.62	16632.58
dm **Python Dictionary**	16.67	30.68	74.91	113.53
dm **Python Dictionary**	16.50	31.10	80.51	122.08
sim **Python**	158.53	179.27	16636.14	16635.36

Table 4.6: The average execution time for a loop of the Python List driver model measured for a combination of simulation versions and benchmark variations

custom-made benchmark [µs]	**headless**	**headless+log**	**visual**	**visual+log**
sim **C++**	167.30	186.308	16629.04	16629.69
dm **Python List**	16.96	31.39	75.04	113.89
dm **Python List**	16.38	32.07	78.94	121.58
dm **Python**	159.83	184.75	16636.01	16634.56

4.1.1 Execution time differences of the C++ and Python implementations of the driver model

The tables above help to answer the question of how large the differences between the C++ and Python implementations of a driver model are. To get measurements purely based on implementation programming language performance differences, the reception of the OSI GroundTruth messages were excluded. This was done with the intention of removing as many external factors as possible in an attempt to isolate the programming language performance difference from the state of the system and network.

Measuring this way, the C++ implementation of the driver model achieved approximately 100 times faster execution times in headless benchmark configuration compared to any of the Python implementations. With the simulation setup in headless mode and logging of the driver model enabled, the C++ implementation was approximately 4 times faster than the best performing Python implementation. Running the simulation in visual mode increased the load on the system and resulted in the C++ implementation finishing its function roughly 36 times faster compared to the Python driver models. The heaviest load on the system was put on the CPU in the visual mode with driver model logging enabled, where the C++ implementation executed the driver model's threat assessment and decision-making function around 2,5 times faster.

Based on these results, using C++ performs at least 2.5 times and up to 100 times faster than the best Python implementation, varying depending on the benchmark configurations.

4.1.2 Performance difference between the two esmini wrappers

The two esmini wrappers used the esmini API by directly including the provided esmini.hpp file and linking the shared library in the case of C++ or by using the ctypes library in the case of Python. Accounting for margin of error and comparing the measured execution times of those two shows that both performed on the same level when excluding table 4.2 with the C++ driver model data. Depending on the bench configuration, either the Python esmini wrapper (headless) or the C++ esmini wrapper (visual) had marginally faster execution times.

The esmini wrapper results when run together with the C++ driver model in both headless modes form an exception to this statement. Using any of the Python driver models in both headless mode configuration resulted in execution times of the two esmini wrappers being roughly 2 times of the values in table 4.2. Results for both visual modes in table 4.2 on the other hand were in line with the statement. This discrepancy is quite

likely explainable with the UDP socket implementations used for the OSI interface and response functions having a time-out set. If the timeout duration is defined as a positive value, the socket will wait for the values' duration to perform an operation before raising a timeout error when the defined limit is passed. During this waiting duration, the socket is blocking, meaning that the code does not continue to run unless datagrams are received or the timeout duration passed. With the C++ driver model being executed so fast, the UDP datagrams sent by it as a response are ready to be consumed as soon as esmini gets to it. The simulation might have had to wait until it received the datagrams from the Python driver models, adding an overhead to the measured times.

With the execution times in the visual bench configurations of the esmini wrappers being in the smaller ten thousands, this overhead seems to diminish. This might be a factor explaining why the results of the esmini wrappers running with the C++ driver model in the headless bench configurations perform roughly 50% better than when the Python driver models are used instead.

Given that the Python esmini wrapper calls the esmini library, and the library executed being the same as the C++ wrapper calls, it is not surprising that the results of the esmini wrappers do not differ except for the mentioned deviation.

4.1.3 Performance differences between the Python implementations using different data structures

The results show only small differences between the averages of the Python driver models. In an attempt to identify a trend, the number of fastest averages per cell were counted, which resulted in the Class and Data class implementations having the most number of best times. This indicates that those datastrcutures resulted in better performance of the driver model in this specific use case, but it has been deemed that the driver model algorithm does not make enough use of reading and modifying datastrcutures to make a general conclusion if one is better than the other in the context of performance. For this, a more purposeful setup and algorithm would be suggested.

4.1.4 Effect of system load on driver model performance

The effect of the load, which consisted of all processes the CPU had to execute at any given system state, was tried to be quantified by focusing on one table at a time and comparing the execution times of the driver model measured in both headless bench configurations and the execution time of their equivalent in visual mode. In these configurations, the driver model had the exact same operations to run at the same computational cost, but the CPU load was higher when the simulation was visualized on screen. Seeing that the execution times measured with visual presentation were slower, it can be said that increased load caused by other processes negatively affects the execution time of driver model functions.

The effect of the load is also visible when looking at the execution times of the esmini wrappers. Headless runs of the simulation with the added small computational load of the driver model logging resulted in slower execution times of the parallel running simulation. With the execution time of the simulation in visual mode being constantly close to 16 630 microseconds, the impact of the small load caused by the driver model logging was too small to affect the execution time. The list below presents the example values based on the C++ driver model table 4.2. Values for the other driver models are of the same scale.

- Driver model execution time change due to visualization of the simulation
  - without logging: 12.06 times higher with C++ esmini wrapper
  - with logging: 6.25 times higher with C++ esmini wrapper
- Simulation execution time change due to logging with driver model
  - headless: 1.05 times higher with C++ esmini wrapper
  - visual: <1.001 times higher with C++ esmini wrapper

This phenomenon could arguably be caused by a higher frequency and impact of context switches issued by the scheduler, resulting from the increased load on the CPU.

4.2 Execution times measured with GNU time command

The tables for the average output of the GNU time command follows the same row and column representation. What differs is that each cell presents the three values representing the total real time, user time and system time in seconds, collected from the start of a software component to its end. For an explanation of what those times mean, see table 3.3. The presented values are the averages of 100 simulation runs and capture the influences of using OSI as well as all external factor the simulation setup is exposed too.

In the case of the C++ driver model running in headless mode, the presented system time is 0. This is caused by the measurement of the time command in seconds and not having a high enough resolution to measure the very little time spent on system calls. With more load and extra system calls used for logging, this phenomenon was not a problem anymore.

Table 4.7: The average time of the Linux built-in command GNU time for the C++ driver model, measured for a combination of simulation versions and benchmark variations

GNU time (Real[s] User[s] Sys[s])	**headless**	**headless+log**	**visual**	**visual+log**
sim **C++**	0.64 0.11 0.01	0.64 0.11 0.01	11.98 2.57 0.23	11.98 2.56 0.24
dm **C++**	0.64 0.02 0.	0.64 0.02 0.01	11.94 0.05 0.02	11.93 0.06 0.04
dm **C++**	0.72 0.02 0.	0.72 0.02 0.01	12.02 0.06 0.02	12.02 0.06 0.04
dm **Python**	0.73 0.18 0.02	0.73 0.18 0.02	12.07 2.86 0.27	12.07 2.84 0.26

Table 4.8: The average time of the Linux built-in command GNU time for the Python class driver model, measured for a combination of simulation versions and benchmark variations

GNU time (Real[s] User[s] Sys[s])	**headless**	**headless+log**	**visual**	**visual+log**
sim **C++**	0.38 0.14 0.01	0.38 0.13 0.01	12.16 2.58 0.23	12.14 2.49 0.22
dm **Python Class**	0.4 0.53 0.36	0.4 0.5 0.34	12.13 0.7 0.39	12.11 0.71 0.38
dm **Python Class**	0.41 0.51 0.32	0.51 0.49 0.32	12.28 0.69 0.34	12.27 0.71 0.36
sim **Python**	0.4 0.26 0.03	0.49 0.24 0.02	12.32 2.94 0.27	12.3 2.78 0.26

Table 4.9: The average time of the Linux built-in command GNU time for the Python data class driver model, measured for a combination of simulation versions and benchmark variations

GNU time (Real[s] User[s] Sys[s])	**headless**	**headless+log**	**visual**	**visual+log**
sim **C++**	0.38 0.13 0.01	0.39 0.13 0.01	12.17 2.6 0.23	12.15 2.5 0.23
dm **Python Data class**	0.4 0.54 0.36	0.41 0.51 0.33	12.15 0.72 0.4	12.12 0.71 0.38
dm **Python Data class**	0.42 0.51 0.32	0.47 0.49 0.31	12.27 0.7 0.34	12.27 0.71 0.34
dm **Python**	0.4 0.27 0.02	0.45 0.25 0.02	12.31 2.93 0.28	12.3 2.81 0.26

Table 4.10: The average time of the Linux built-in command GNU time for the Python dictionary driver model, measured for a combination of simulation versions and benchmark variations

GNU time (Real[s] User[s] Sys[s])	**headless**	**headless+log**	**visual**	**visual+log**
sim **C++**	0.38 0.13 0.01	0.39 0.13 0.01	12.17 2.61 0.24	12.15 2.49 0.23
dm **Python Dictionary**	0.4 0.52 0.35	0.41 0.49 0.32	12.14 0.7 0.39	12.12 0.7 0.38
dm **Python Dictionary**	0.41 0.5 0.31	0.51 0.49 0.3	12.27 0.68 0.33	12.27 0.71 0.34
sim **Python**	0.4 0.26 0.02	0.5 0.25 0.02	12.31 2.95 0.27	12.31 2.8 0.25

Table 4.11: The average time of the Linux built-in command GNU time for the Python list driver model, measured for a combination of simulation versions and benchmark variations

GNU time (Real[s] User[s] Sys[s])	**headless**	**headless+log**	**visual**	**visual+log**
sim **C++**	0.39 0.14 0.01	0.4 0.13 0.01	12.16 2.61 0.24	12.14 2.5 0.23
dm **Python List**	0.41 0.51 0.33	0.42 0.49 0.32	12.13 0.69 0.37	12.12 0.7 0.37
dm **Python List**	0.44 0.48 0.3	0.52 0.48 0.3	12.28 0.67 0.32	12.27 0.7 0.34
sim **Python**	0.42 0.26 0.03	0.5 0.25 0.02	12.32 2.94 0.27	12.31 2.79 0.26

4.2.1 Execution time differences of the C++ and Python implementations of the driver model

Aligning with the results in section 4.1, the C++ driver model beats all Python driver models comfortably for measurements in user and system time, independent of the esmini wrapper and bench configuration. As there are null values in certain cases, 0.01 has been taken as replacement value to be still able to use a 'X times faster than" measurement. Zero values mean that the used measurement resolution was not high enough to capture the minimal activity that actually happened. The measurement resolution was 0.01 seconds, meaning that the zero values missed capturing a time < 0.0045s as times above this limit would have been rounded up to 0.01. While the true value could be anything below that limit and therefore is undefined, using the measurement resolution as replacement values defines a reference close to the upper limit of possible values and helps to create an indicative comparison where zero divisions are no issue. This way an outcome with worst case results are assumed in cases of zero value, where in reality the time was < 0.0045s instead of 0.01s and therefore slightly better. While the value produced this way does not indicate anything about the best case, it allows using the calculated value as a worse than worst case result. Looking at the calculated value this way, the C++ driver model performed at least 24 (user time) and at least 30 (system time) times faster than the best performing Python driver model in headless mode.

The user and system time results of the headless and logging benchmark configuration were similar to the results of the headless benchmark configuration of the C++ driver model compared to the Python implementations. When the simulation was run in visual mode, the C++ driver model performed 13.4 (user time) and 16 (system time) times faster than its fastest Python equivalent. When the visual mode was combined with the logging of the driver model, the factors decreased by 11.67 (user time) and 8.5 (system time) times.

Surprisingly, real time measurements provided by the time command are substantially higher in both headless benchmark configurations when the C++ driver model was used, resulting in the execution times being between 1.44 and 1.71 times slower. This applies to both software components of the simulation setup compared to their real time results of the Python driver models running with any of the two esmini wrapper. This is not expected, and not explainable by the scope of work done. A potential cause could be a synchronization issue affected by the start-up time of the two software components. This assumption is based on the difference between the real time results and the sum of the user and system being close to the defined timeout value of 500 milliseconds.

Benchmarking in any of the two visual benchmark modes, the real time measurements of the GNU time command for the C++ driver model command were again better than the best performing Python driver model, ranging between 1.016 and 1.022 times faster execution times.

4.2.2 Performance differences between the Python implementations using different data structures

The results of Python driver models are almost always identical for a given combination of esmini wrapper and visual benchmark configuration. If there was a difference, it could be argued that the difference is in the margin of error. In the two headless configurations, the measurements varied by a factor between 1.025 and 1.10 times.

Comparing the execution times of a driver model executed together with the C++ esmini wrapper with the execution times of the same driver model executed together with the Python esmini wrapper, some differences could are observed:

- Performance benefit for Driver Model executed with C++ esmini wrapper
  - in all but the headless bench configuration, the real time measurements are between 1.025 and 1.216 times faster when compared to the values of the driver model when run with the Python wrapper of

the simulation.

- Performance benefit for Driver Model executed with Python esmini wrapper
  - all system time measurements are between 1.053 and 1.155 times faster when compared to the values of the driver model when run with the C++ esmini wrapper
  - slight differences depending on bench configuration for user time measurements
    - visual + logging: same as with C++ esmini wrapper or 1.014 times slower in the case of the Python dictionary driver model
    - all other configurations show using the Python esmini wrapper influenced the driver model positively, resulting in user times being between 1.014 and 1.059 times faster when compared to the values generated when the Python driver models were executed together with the C++ esmini wrapper

Using the C++ esmini wrapper instead of the Python esmini wrapper resulted in better real time measurements of the GNU time command for the Python driver model. The system time results of the GNU time command for the Python driver models benefited of running parallel with the Python esmini wrapper instead of the C++ esmini wrapper. In all but the visual and logging bench configuration, the Python esmini wrapper benefiting the driver model performance is also true for the user time results of the GNU time command.

4.2.3 Performance difference between the two esmini wrappers

Using the C++ esmini wrapper resulted in the fastest results in every possible combination of driver model variant, esmini wrapper and bench configuration. Excluding the headless bench configuration results collected when using the C++ driver model together with the C++ esmini wrapper, the measured benefits ordered in real, user and system time were:

- headless: 1.05-1.11x, 1.462-51.9x, 1.50-1.667x
- headless + logging: 1.134-1.225x, 1.559-1.58%, 1.50x
- visual: 1.008-1.013x, 1.13-1.127x, 11.2-17.9x
- visual + logging: 1.01-1.014x, 1.11-1.16x, 1.08-1.154x

The user and system time measurements capture the overhead caused by the Python Interpreter, showing a big impact in headless modes which decreases with increased load on the CPU to a still noteworthy amount. As a result, the real time measurements of the Python esmini wrapper are slower than the C++ esmini wrapper real time values.

4.2.4 Effect of system load on driver model performance

As the driver model has to wait for updates from esmini, its real-time execution time is dependent on the simulation execution duration. Therefore, it is not sensible to compare the real time measurements of the headless and visual or headless + logging and visual + logging to conclude the effects of increased load on the driver model performance. Instead, the user and system time values can be used, as they were not influenced by the described phenomenon. Table 4.7 shows that for the C++ driver model, the user and system time measurements were between 2x to 4x slower when the load was increased. In the case of the Python driver models, tables 4.8 to 4.11 show that the user time measurements were approximately 1.4x and the system time measurements 1.086x and 1.13x slower.

4.3 Histogram of all measured times via the internal function

For this section, all measured execution times of a driver models' threat assessment and decision-making function for a bench configuration executed with one of the esmini wrappers are shown as histograms. The focus of this section was only on the performance difference between the C++ implementation compared to the Python implementation. Instead of adding all histograms here, the Python driver model with the best mean time is presented. The histograms of the other Python driver models can be found in the appendix. All

histograms include the percentage of removed outlier, potentially caused by the scheduler of the operating system used, and the mean value of the data set. The multitasking nature of the operating system and its effect on the measurements are like noise on an electrical signal. Every sample is affected by it, but some can be so noisy that they do not represent the real voltage. In this context, using the filter was aiming to remove measurement which were too distorted to be representative.

4.3.1 Headless bench configuration

In both headless modes, the C++ was magnitudes faster when executing the threat-assessment and decision-making function.

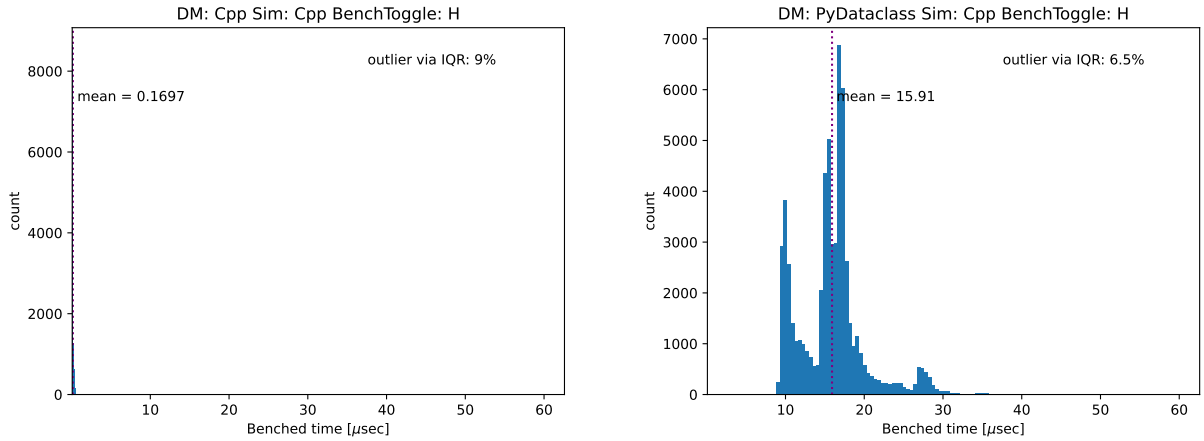


Figure 4.1: *Histogram for the C++ and best performing Python driver model implementations results in the headless benchmark configurations using the C++ esmini wrapper*

Figure 4.1 visualizes the difference nicely, as the all the bars are located on the left edge and really look like only one bar. That is caused by choosing edges based on the minimum and maximum values of all measurements for a given esmini wrapper and bench configuration combination. Figure 4.2 has been added to get a proper look at the C++ histogram, as it does not use the global edges, but edges based on the C++ driver models' own data only.

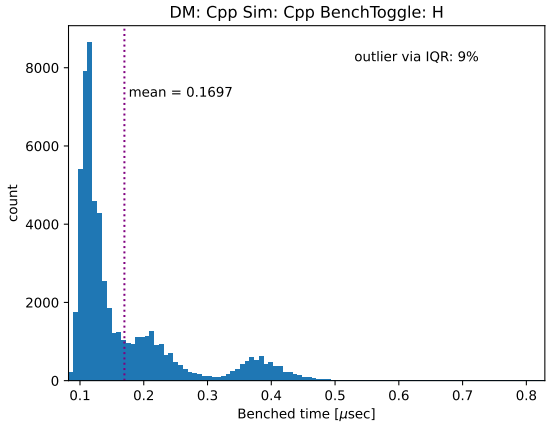


Figure 4.2: *Zoomed view of histogram for the C++ driver model implementation results in the headless benchmark configuration using the C++ esmini wrapper*

The first takeaway is the higher percentage of removed outlier in the results of the C++ implementation compared to Python driver models. While the percentage for the Python driver models was between 5.3%

and 7%, the C++ driver model had 9%. This can be explained by scheduler processing other processes before finishing the process of the driver model. As the execution time without the scheduler interfering is so low for the C++ driver model, even context switches that would add a small amount of execution time which would not lead to outlier in the Python driver models, lead to outlier in the C++ driver model.

4.3.2 Headless logging bench configuration

With the driver model also logging, the execution time of the C++ implementation was more than four times faster.

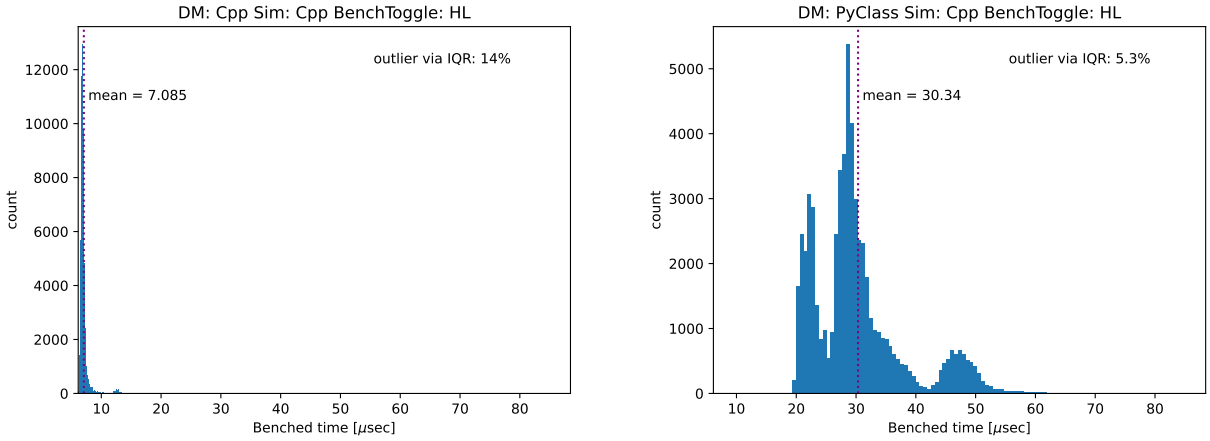


Figure 4.3: *Histogram for the C++ and best performing Python driver model implementations results in the headless + logging benchmark configuration using the C++ esmini wrapper*

The percentage of outlier increased to 14%, while the Python driver models had a percentage between 5.3% and 6.2%. Zooming into the C++ histogram results in figure 4.4.

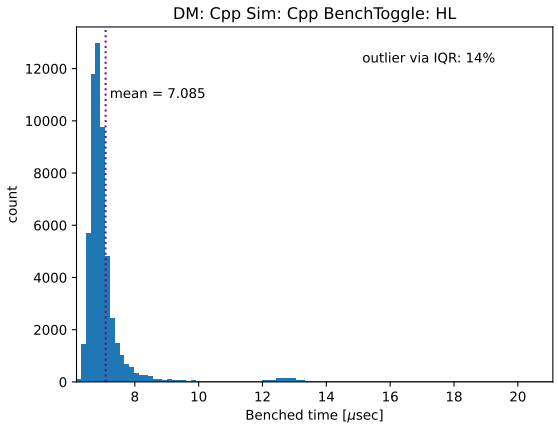


Figure 4.4: *Zoomed view of histogram for the C++ driver model implementation results in the headless + logging benchmark configuration using the C++ esmini wrapper*

A more compact and even close to normal distribution was observable, even when compared to figure 4.2. The higher percentage of outlier can be explained by this fact. The more compact band of the bins results in narrower interquartile ranges used to determine the outlier. Given those narrow interquartile ranges and a baseline execution time of the function under examination falling in one of those bins, a context switch adding some extra time on top of the baseline is more likely to be classified as an outlier.

4.3.3 visual bench configuration

In visual bench configuration, the C++ implementation performed 35 times faster than the best Python implementation.

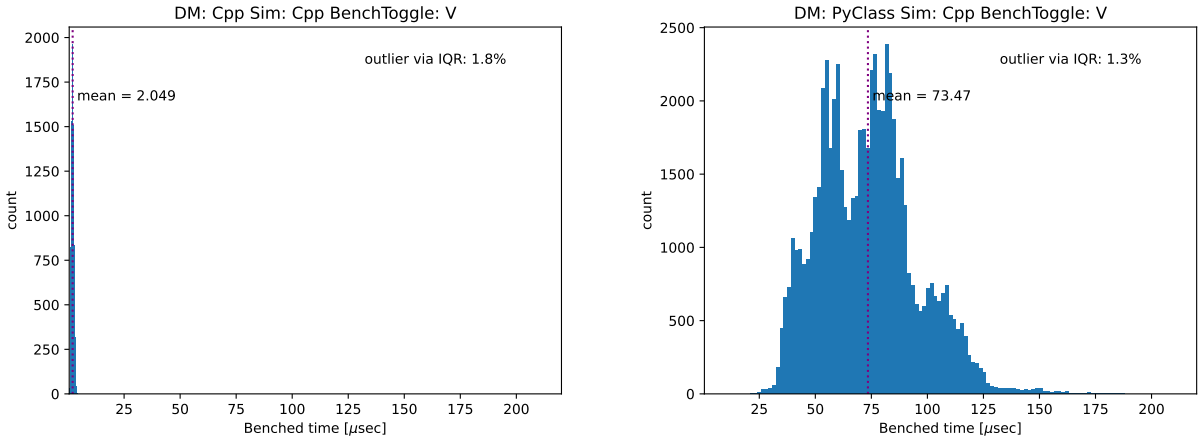


Figure 4.5: *Histogram for the C++ and best performing Python driver model implementation results in the visual benchmark configurations using the C++ esmini wrapper*

Figure 4.6 presents a zoomed in view again and a normal distribution is observable. With the increased load, the Python driver model got closer to a normal distribution as well.

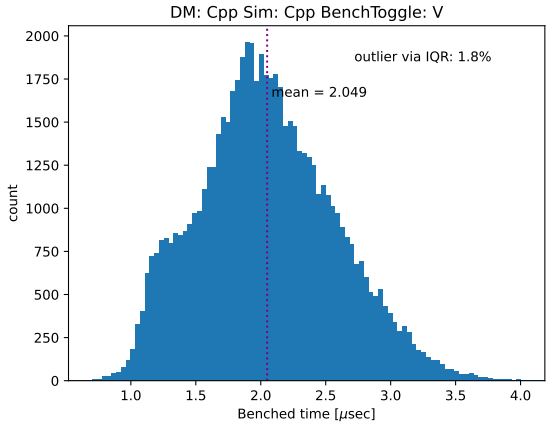


Figure 4.6: *Zoomed view of histogram for the C++ driver model implementation results in the visual benchmark configurations using the C++ esmini wrapper*

The percent of outlier decreased to 1.8%, while the Python driver models had a percentage between 1.3% and 1.5%.

4.3.4 visual logging bench configuration

With logging of the driver model enabled, the C++ implementation performed around 2.5% faster.

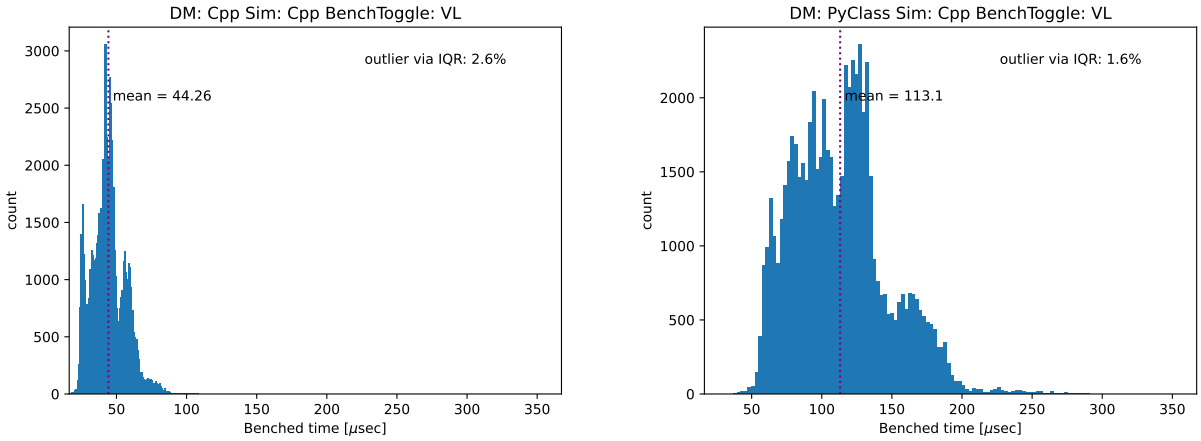


Figure 4.7: *Histogram for the C++ and best performing Python driver model implementations results in the visual + logging benchmark configurations using the C++ esmini wrapper*

Its outlier percentage was 2.6% while the Python driver models had a percentage between 1.6% and 1.7%.

5 Discussion

A simulation setup consisting of an esmini wrapper component, which stepped the simulation scenario, as well as a driver model component, which controlled the ego vehicle in the simulation, was used for benchmarking 100 times under different benchmark configurations. During these benchmark runs, data was collected about:

(I) the execution time of an iteration of the simulation setup component functions

  (a) threat-assessment and decision-making (driver model)

  (b) scenario stepping (esmini wrapper)

(II) the statistic of the GNU time command on both simulation setup components used in the simulation setup

The generated execution times of the function iterations were filtered, and all data was evaluated. This work set out to answer the research question:

How much does the choice of programming language (between Python and C++) affect the execution time of virtual simulations, when the simulations include a driver model interfaced via the open simulation interface (OSI)? On the side, the influence of data structures and load on the CPU were tried to be captured.

These questions were attempted to be answered by evaluating

(I) the execution time of the main functionality provided by each component of the simulation setup

(II) the execution time for the full simulation runs

While the subsections provide more details, the following list presents concise answers to these questions:

- based on the custom-made benchmark, the C++ driver model implementation execution times were between approximately 2.5 and 100 times faster than the Python driver model implementation, depending on the benchmark configuration. The lower value was obtained in the benchmark configuration with the highest CPU load by visualizing the simulation and external influences, such as context switches. The higher value was measured in a configuration with minimal external factors and CPU load, as the simulation ran headless..

- The C++ esmini wrapper performed around 2 times faster in simulation runs without visualization, while there was no observable difference between a wrapper implemented in C++ and Python in visualized simulation runs

- based on the GNU time command data captured, using C++ for the simulation setup instead of Python is performing faster (between 11x and 24x for user and system time measurements), but the data also showed unexpected results.

- based on the custom-made benchmark and GNU time measurements, computer load affected the performance of all components in the simulation setup negatively

- based on the custom-made benchmark and GNU time measurements, the choice of data structure used in the Python driver model only resulted in small differences in their measured times, negligible when compared to the performance difference when using C++

5.1 Performance differences of the driver model between C++ and Python

In this subsection, the observed performance differences for the driver model are discussed in detail. The collected data for the main functionality iteration and using the GNU time are discussed separately.

Table 5.1: The performance factors of the C++ driver model over the best Python driver model in main functionality iteration execution times

benchmark configuration	headless	headless + logging	visual	visual + logging
Performance factor driver model	~ 100x	~4x	~35x	~2.5

5.1.1 Driver model main functionality iteration execution time

Table 5.1 summarizes the performance factor the C++ driver model has over the Python implementations based on the measured execution times of the main functionality execution. As variations of the Python driver model and simulation variants delivered different times, the presented performance factor is a representative value established based on the ranges calculated.

The data shows that using C++ instead of Python can result in drastically faster performance, depending on the benchmark configuration, up to a factor of 100 times. This was the case in the headless benchmark configuration, which is a configuration with low load on the CPU and driver model functionality that doesn't contain any high-level system calls.
As the functionality of the driver model is the same between the headless and visual benchmark configuration, the decrease of the performance factor to 35x is attributed to the effects of increased load. The specific cause is assumed to be increased occurrences of context switches based on the multitasking nature of the used operating system, but it was not confirmed in the scope of this research.The same decrease with a lower effect is observed when comparing the performance factor of the C++ driver model in headless and logging to visual and logging. In these benchmark configurations, the driver model functionality is the same, but the load on the CPU is increased, resulting in the performance factor dropping from 4x to 2.5x.

The effect of system calls are deducted by comparing the results of the headless and visual benchmark with the benchmark configuration results of the same benchmark with driver model logging enabled. In these comparisons, the loads are almost identical and only the number of higher-level system calls per iteration differ, being either constantly 0 or 1 during a simulation run. The comparison for the headless benchmark configurations shows the performance factor decreasing from 100x to 4x, while it decreased from 35x to 2.5x in the comparison of the virtual benchmark configurations.

Histograms based on these measurements show visually how drastic the performance difference between the driver model function iteration is when C++ was used instead of Python. Those histograms also presented the percentage of values filtered, showing a slightly higher rate for the C++ driver model, with the difference being less the more load was on the CPU.

5.1.2 Driver model GNU time results

While the custom-made benchmark measured only an isolated part of the execution chain, comparable to lab conditions, the GNU time measured the whole execution chain, being more relevant in real applications. Both provided a wall-clock time, but the GNU time command additionally provided times of the process using the CPU actively in user and kernel space.

The results of the GNU time command for the driver models show the C++ driver model spending less user time and system time, with the factor ranging between 11.17 and 24 confirming the findings based on the custom-made benchmark measurements. The C++ driver model implementation seems to perform the same functions in substantially less time based on time spent actively on the CPU in user space and kernel space, but the real time measurements of the time command in both headless benchmark configurations show the C++ driver model require between 1.6 and 1.8 times longer compared to the driver model real time measurements of the Python driver models. The cause for this has been not uncovered in this research, but a potential cause is provided later in this section. For the visual benchmarks, the real time measurements are again faster when using C++. Based on the nature of compiled and interpreted programming languages, the C++ driver model was expected to have faster real time measurements in all benchmark configurations.

As the sum of the user and system time measurements of the C++ driver model are smaller than the real time measurement in headless benchmark configurations, a strong indicator of the driver model waiting is present. A potential cause could be synchronization problems based on the faster start of esmini in headless modes. The difference of the real time measurement with the sum of user and system time are close to the set timeout of 500ms, a timeout was suspected. Going through the logs of all simulation runs shows no indications

of timeouts, though.

The reason why the GNU time command real time measurements show this issue and the custom-made benchmark doesn't, could lie in the way they measured. The GNU time command measures all operations invoked by the program measured from its start to end, while the custom-made benchmark measurement captured only a specific sequence of operations. This sequence of operations did not include UDP datagram packet exchange functions, and it is assumed to observe a similar result if the datagram exchange operations were included.

As the deviation of the real time measurements in the headless measurements can not be explained, the validity of those measurements are not given, and the real time results are not considered in the discussion.

5.1.3 Effects of esmini wrapper choice on driver model performance

The choice of programming language stepping the simulation also seems to affect the performance of the driver models tested, with the C++ powered esmini wrapper having almost negligible to noteworthy better performance in most cases across all metrics. The main functionality iterations time of all driver model implementations are in all but two cases faster when the C++ esmini wrapper was used. The GNU time command results for user and system of the C++ driver model show no differences, while the Python driver model implementations show a slightly better user and system times when combined with the Python esmini wrapper. Real time measurements of the GNU time command on the other hand are slightly faster for all driver model implementations when the C++ esmini wrapper was used. The faster real time measurements are potentially caused by the C++ esmini wrapper being faster than its Python equivalent, stepping faster through the complete scenario and processing the driver model responses faster, and therefore, exiting the simulation run before than when the same driver model is combined with the Python esmini wrapper.

5.1.4 Effect of data structure on Python driver model

The data is also evaluated if there are noteworthy differences depending on the data structure used in the Python driver model. While both measurements showed slight tendencies favouring the class and dataclass data structure, the differences are negligibly small generally, and especially irrelevant when compared to the differences when switching to using C++. The driver model algorithm doesn't seem to perform enough operations with the data structure to create data that properly captures the influences based on the choice of used data structure. Therefore, a tendency of performance benefits is observed for using classes and data class over the other two used in this research, but the benchmark is deemed as insufficient to formulate a valid, general statement.

5.1.5 Effect of increased CPU load on the driver model performance

The effect of increased load were researched on the driver model, as there are different benchmark configurations where it has the same functionality but is measured with different load. Technically, the esmini wrapper performance could be compared similarly, but the added load off driver model logging was deemed too small to arrive at meaningful results. With the collected data, it is observed that the load of the CPU negatively affects the performance of any driver model function iteration, as well as their user and system time measurements of the GNU time command.

5.2 esmini wrapper performance

In this subsection, the observed performance differences for the esmini wrapper are discussed in detail. The collected data for the main functionality iteration and using the GNU time are discussed separately. As the Python esmini wrapper calls the esmini shared library directly using ctypes, no differences in the main functionality iteration execution time were expected. After all, the same library and binary is used. Since the GNU time command measures from start to finish, instead of measuring a specific part of software, differences were expected caused by the additional overhead interpreted languages like Python have to deal with.

5.2.1 Esmini main functionality iteration execution time

The results mostly confirmed the expectations. Independent of the language used for the esmini wrapper, the driver model implementation it ran with and the benchmark configurations, almost all results are on the same level with minimal differences. One exception was found when any of the esmini wrappers was run with the C++ driver model.

Table 5.2: The performance factors of both esmini wrapper running with the C++ driver model over both esmini wrapper running with any Python driver model in main functionality iteration execution times

benchmark configuration	headless | headless + logging	visual | visual + logging
Performance factor esmini wrapper	∼2x	∼1x

As shown in table 5.2, the C++ and the Python esmini wrapper showed 2 times better performances when combined with the C++ driver model instead of the Python driver models in headless benchmark configurations, while no real difference could be captured in the visual benchmark configurations. As presented in the results section, the results of the headless benchmark configurations of the esmini wrappers combined with the C++ driver model are assumed to originate from faster driver model responses, allowing esmini to continue with the next simulation step instead of having to wait. As it was not in the scope of this thesis to figure out what is behind this behaviour, other factors are not ruled out.

5.2.2 Esmini wrapper GNU time results

The same issue as described in subsection 5.1.2 was observed for the esmini wrappers running together with the C++ driver model in any of the headless benchmark configurations and is not reiterated here. Besides the issue with the real time measurements, the C++ esmini wrapper showed improved user and system time measurements compared to the Python esmini wrapper. The degree of performance gained depended on the benchmark configurations, with the headless benchmark configuration showing the biggest performance differences and the visual + logging configuration showing the least.

As there are concerns regarding the validity of the real time measurements of the GNU time command, the next observation comes with reservations: The real time measurements yielded by the GNU time command for measurements involving the Python driver models show faster results for the C++ esmini wrapper. While it is negligible in both visual benchmark configurations (∼1.013x), it ranged between 1.05x to 1.23x in the headless benchmark configurations. Overall, the GNU time results matched the expectations and captured the overhead introduced of the interpreted implementation, showing in the form of slower measurements of the Python esmini wrapper.

5.3 In which cases is the performance gain of using C++ over Python relevant?

The observations indicate a serious performance gain using C++ for the complete simulation setup. The overhead of launching and terminating the Python interpreter, as well as having the interpreter as a layer between the source and byte code, has a noticeable impact even in this minimal working example. In cases of complex simulations with a serious amount of logical and arithmetic operations, such as in sensor fusion functions, a performance limited simulation setup in Python could potentially run when implemented in C++. Depending on the algorithms used, even bigger performance differences could be observed in more complex cases.

Nonetheless, those situations would have to be quite complex and the performance of Python on a modern day computer system should be enough to cover most cases of simulations for the purpose of ADAS & ADS research as well as driver model and behaviour research, specially in the early stage. Here, the usage of Python comes with benefits as stated in the introduction: Its more natural syntax like language allows for shorter and more readable code, while its interpreted nature allows for fast prototyping and iterations. Arguably, it offers easier debugging.

In the opinion of the author of this thesis, in the stages where only a few simulations runs are performed or the functionality of the software changes fast, the advantages of using Python are more relevant than the performance gains of using C++. The slower performance of the Python implementations in this research

did not affect the simulation itself in any way, as the logs of the simulation runs are identical for all logged parameters. A prerequisite is that the performance is not limiting the applications' functionality. If that prerequisite is not given, switching to C++ can help solve the problem. It should be also mentioned that there are options to optimize the Python performance, but it depends on the individual and their comfort in using C++ or advanced techniques with Python.

Where the author sees C++ always ahead is in stages of research or development, where large amounts of simulations are performed, i.e. parameter optimization. With a complex enough simulation, a use case where the C++ implementation is 3 seconds faster per simulation run than the Python implementation seems to be possible. In such a case, 10 000 simulation runs would result in saving more than 8 hours in execution time. At those stages, the software functionality doesn't change much, if at all, and should be tested thoroughly. This leads to the advantages of Python loosing importance, while the efficiency of C++ gains importance. To give recommendations based on applications, the author would recommend using Python for driver modelling and understanding driver behaviour, while recommending the usage of C++ for ADAS & ADS during all stages of development and research.

5.4 Why do the real-time measurements of the GNU time command in visual benchmarks show less performance differences?

The duration of the simulation varies between headless and visual simulation runs, as the visualization is computationally demanding. Therefore, there is a minimum time passing required for the stepping of the simulation, where the visual simulations have a higher minimum time. This minimum time causes the driver model to wait for OSI GroundTruth updates, and therefore, is coupled to the simulation execution. The GNU time command's real time measurements show the coupling to the simulation execution duration, while the custom-made benchmark excluded the data exchange and therefore, could capture the wall-clock time performance differences between the programming languages.

5.5 Discussion about the methodology

This section discusses aspects of the chosen methodology, with a subsection talking about the driver model specifically. The methods in this research try to provide a good initial view of performance differences to be expected when using C++ or Python in virtual driver model simulation by looking at a minimal working example of their implementation. Two measurements were performed in different benchmark configurations, trying to capture the performance of the main functionality and the complete simulation run isolating different influencing factors. The data captured was attempted to be analysed by putting the system state (operating system function and the current load on the CPU) in context.
With the intention stated, the methodology mainly succeeded, but certain points should be discussed:

- Trying to capture the effect of CPU load, the load was increased by switching to the visual benchmark configuration. While the load caused by the simulation setup did increase noticeably, in the context of a modern CPU it is still small. Therefore, running the same benchmark configuration on a freshly restarted system and on a system, which has already a high load might be a better approach.

- It can be argued that the computational cost of the driver model algorithm is so small that the results are not representative for more advanced cases.

- The GNU time command produced some unexplainable results, and the author didn't investigate its cause and no alternative to GNU time was considered.

- The filtering of the main functionality iteration execution times with the IQR method were done to remove extreme outlier caused by the operating system's multitasking nature. It is debatable if they should have been included, as those outliers do not purely represent the programming language performance, but also contain the ever present varying influence of the operating system. All measurements have that influence, but in the case of outlier it was disproportionally high.

- In this research, the simulation setup was terminated and restarted after each run. Towards the end of the research, it was pointed out that it is not uncommon to start up the simulation setup only once

and run repeated simulation runs that way. In that case, the Real time measurements of the GNU time command for the Python implementations are expected to be closer to the one using C++. The impact of starting up the Python Interpreter would affect the measurements only once for all simulation runs, instead of once for every simulation run.

5.5.1 Driver model

The chosen driver model, while being conceptually advanced, was a quite short and computationally lightweight algorithm. It only required four inputs from the OSI GroundTruth, namely the length, width and position of the target vehicle as well as the position of the ego vehicle. A driver model requiring more inputs and performing more arithmetic would potentially show bigger performance benefits overall when using C++, based on the custom-made benchmark measurements in headless modes. In the headless modes, the measurements were based on almost purely user space operations, in which the arithmetic operations fall. The increased number of inputs would also mean that the performance of the used data structure would have a bigger impact, as more information would have to be copied/accessed.

Using a more complex driver model, which aims to replicate multiple systems used in a vehicle, would have to also replicate the data exchange of the electronic network. In that case, the data exchange impact on the performance is more important than in this research, which excluded it in the custom-made benchmark. The author has no predictions on how the increase of data exchange in a network would affect the performance differences between C++ and Python.

5.6 Relation to other work

While [61] focused on the performance, safety, and security of the Rust programming language, it was compared to C++ and Python amongst others. The CPU time measurements show that for the three algorithms used, C++ performed 30.82, 60.02x and 160.71x faster than Python. A hybrid approach has been demonstrated in [62], where Python is combined with C/C++ and Fortran to achieve performance close to native C/C++ implementations.

The pre-print article [63] compared C++ and Python the execution time and memory used executing different algorithms. It concluded that C++ outperforms Python significantly, but also recommends Python for beginners based on its easier syntax, readability and portability.

The work conducted in [64] compares one C++ implementation of an experiment with 4 different Python implementations to determine their performance in the context of animations using Autodesk Maya. Similarly to this work, the implementations have been written the same way in all implementations, using language specific data types. What differed in that work is that the 4 Python implementations differed from each other drastically, while the Python implementations in this work differed only in the used data structures. The result was that the C++ implementation performed between 1.38x to 31.65x times faster than the Python implementations, depending on the Python implementation and experiment variant.

In [65] seven different programming languages have been compared for productivity and execution performance by implementing a phone book application by a group of developers. Here, the performance ratio was roughly 10x, but Python has been attested a better productivity based on needing half as much time to write.

The creators of CarMaker state explicitly that it is using C++ (and C) for performance reasons [16]. Given that according to [17] openPASS specifically aims to offer fast performance and it being written in C++, it can be concluded that C++ was chosen for its performance. While the author couldn't find any direct statements about performance claims, the author observed that the source code of esmini is written in C++ and assumes the reason to be the same. The choice of programming language of the simulation platforms is seen as another indicator attesting C++ the best performance.

A thorough comparison of six programming languages, including C++ and Python, was conducted in [66]. The experiments used are categorized in execution time, memory consumption, GUI development, database connectivity and web programming. C++ has been determined to be the performing best in regard to execution time and database connectivity, while Python was recommended for Rapid Prototyping, as it performed best in regard to code to task ratio.

While [67] focus was on answering the energy consumption of twenty-seven programming languages, the conducted experiments were also measured for execution time. The C++ programming language had the second best in two and the third-best execution time in the three experiments conducted, while Python was the

2nd worst performing in two and the 5th worst performing programming language in one of the experiments. Expressed in ratios, the C++ language performed between approximately 40-64x faster in terms of execution time across all experiments.

The Computer Language 24.06 Benchmarks Game is a free project that provides an overview of statistics about the performance of programming languages for a given subset of algorithms [68]. Looking at a subset of the tested algorithms and their wall-clock times, the best C++ implementation for the fannkuch-redux, n-body and mandelbrot algorithms performed 89,31x, 163,86x and 1784.5x faster than the best Python implementation, respectively. The CPU time of those algorithms were 69.4x, 165.85x and 182.75x faster than the lowest CPU time of the Python implementations. It also points out that with optimizations and skill of the programmer, the results vary drastically. Looking at the regex-redux algorithm, one can see that the application also plays a relevant factor. The C++ performance factors went down to 1.53x for both, the wall-clock time and CPU time.

While no other related work looked at the specific application of driver models, their findings align with the results of this thesis: C++ does offer the better performance over Python, while Python is more beginner-friendly and useful for Rapid Prototyping applications.

5.7 Limitations and Future Work

This research was focused on a very specific application using a minimal working example. While there are plenty of performance comparisons of C++ and Python, they are mainly based on synthetic work load. Based on the real work load, the results can be quite different. Therefore, the research was limited to a specific simulation setup using two software components, with one being based on the open-source software esmini and the other one being a driver model implementation. Both have been tested using one defined, minimalistic scenario. These limitations allowed to investigate performance differences originated by the choice of programming language in the context of virtual simulation and gives an indicator, but not a complete picture.
Different software components could result in different results. It is expected that the C++ implementation would continue performing better than a Python implementation, but at least the factors are expected to vary. In order to push Python to the best possible performance, alternative Python language implementations, like PyPy and Pyston using Just-in-Time compilers, could have been used. An alternative to that is to compile the Python source code to an executable binary using Cython, Numba or Pythran [69]. Since standard Python is more widely used, this research only used the standard Python language implementation, called CPython.
This is also a good time to highlight that with programming, the experience and skill of the developer can have drastic impact on the results. As an example, the author learned at the end of the research that using `file << "...\n";` instead of `file << "..." << std::endl;` would have resulted in the performance of the C++ driver model in benchmark configurations using logging being 1.6 (headless) and 2.76 (visual) times faster, based on a small test using the custom-made benchmark. This performance improvement is based on `file << "..." << std::endl;` flushing the output stream after adding a new line, which does not happen when `file << "...\n";` is used instead. The author of this report has many years of semi-professional experience using Python, while the C++ programming was learned to conduct this research. To limit the effect of not being that experienced in C++, the implemented algorithms were written to be the same functionality line by line and no language specific performance optimization has been done on the algorithm itself. With this measure and trying to be aware of any biases, the research conducted was done trying to be as objective as possible. It is believed to be a representative case, as it is quite common that researches and engineers are familiar in using Python but would like to know if using C++ would benefit them in conducting meaningful research.

To verify the results and clarify questions left open, more complex simulations, possibly with language specific performance optimizations, could be used in future work. This could capture results based on more intensive simulation runs and potentially give a more complete picture of the performance differences by looking at more advanced usage of the languages, for example by using alternative Python language implementations. With a complex enough simulation, scenario and driver model, the effects of increased load on the performance of the simulation setup could be better captured. Another limitation was found in the modular approach of the software components.

Based on the need to combine C++ and Python on either side of the simulation setup, UDP sockets were used to send the simulation ground truth to the driver model, where the OSI standard was really practical. Instead, when using a monolithic design, the OSI ground truth can be accessed internally. This would simplify the control over the execution order and allow for different performance measurements by removing the influences of package reception time variation.

6 Conclusion

This section starts with a box containing a short overview of the main results before providing more details and information about what the overview is based on.

A complete Python based simulation setup would be sufficient on a modern day computer to perform not too complex ADAS & ADS safety benefit assessment, perform driver behaviour studies or develop driver models. Many programmers are likely to benefit from the easier syntax and improved readability of Python more than from the performance gains of C++. C++ is, however, recommended for complex and computationally advanced simulations, as it occurs in ADAS & ADS development. Given that ADAS & ADS system often are highly computationally costly, its safety benefit assessment should also use C++. In those cases, as well as in product development, where the final product will run using C/C++ on the hardware, using C++ for the complete software product life seems sensible. In the situation where the performance becomes an issue when using standard Python, the choice between switching to C++ or optimizing Python, e.g. by using a JIT Python Interpreter or compiled Python code, depends on the comfort of the developer of using either of those two choices to achieve the desired performance. Alternatively, future work could explore the usage of AI tools to convert Python code to C++ to improve performance - possibly using template code. As the research was conducted on a minimal working example generating low computational cost, the findings could differ when a more complex and computationally costlier application arises. For this, future work could look into scaled and more complex simulations.

The main goal of the current study was to determine the performance difference between C++ and Python in the context of a driver model used in a virtual simulation powered by the open-source software esmini and using its OSI functionality to exchange data between the simulation and the driver model. For this, the esmini API was used by writing two wrappers, one in C++ and Python. The Python wrapper was written to see if it is feasible to only use Python for the whole simulation setup. The same driver model algorithm was implemented once in C++ and four times in Python. While the Python driver models are identical, they differ in the used data structures to see if data structures impacted the performance notably. In order to compare the performance of the programming languages in a virtual simulation context, the wall-clock execution times were measured for an isolated part of the execution chain (using a custom-made benchmark), as well as the complete execution chain (using GNU time). The measurements of the whole execution chain also provided data about the CPU time in user and kernel space.

Based on the custom-made benchmark results, the C++ driver model implementation performed between 2.5x to 100x faster than the Python implementations, depending on the benchmark configuration. Both esmini wrappers showed similar performances measured with the custom-made benchmark, only differentiating in both headless benchmark configurations. In those, both esmini wrapper variants performed roughly twice as fast when running parallel with the C++ driver model. The simulation software needs a fixed time to execute all its operations, and the rest of the time is spent waiting for a message. Based on the more granular control over the driver model, the waiting time was excluded in the custom-made benchmark measurements of the driver model, but captured in measurements on the esmini wrappers. Therefore, the measured performance benefit is attributed to the C++ driver model executing so fast that it cuts down the waiting time of the esmini wrapper implementations.

GNU time dm Measurements using the GNU time command showed user time results (CPU time in user space) being between 12x and 24x faster, and the system (CPU time in kernel space) being between 8.5x to 30x faster when C++ was used for the driver model instead of Python, again depending on the benchmark configuration.

Real time measurements of the GNU time command showed unexplainable results in headless simulations with the C++ driver model, showing performance benefit factor of roughly 0.5x. In visual simulations, the same measurements resulted in minimally faster times for the C++ driver model implementation (i.e, between 1.016 and 1.022 times faster). The GNU time real time measurements captured the data exchange between the driver model and esmini, where each wait for a message from the other at different time points of the execution chain. This led to the GNU time real time measurements of the esmini simulation and the driver model to be close to each other.

Except the unexplainable results in headless modes in combination with the C++ driver model, the real time results of the C++ esmini wrapper were a bit better compared to the Python esmini wrapper, potentially showing the overhead expected of running an interpreter. User and system time GNU time results show

performance benefits varying between 1.08-1.67x when using C++ over Python for the esmini wrapper. This indicates that C++ was more efficient, or at least caused less load on the CPU, as it executed the wanted functionality using the same algorithm by using the CPU actively for less time.

The attempt to identify performance impact based on the used data structure utilized in the driver model was stopped, as the used code turned out to not be representative enough to answer that question properly. Nonetheless, a trend that the Class and Data class were performing better than dictionaries and lists was identified. Finally, the author evaluated the data to see if the load on the system affected the performance notably, which was the case. The same functionality was performed, requiring more wall-clock time when the load on the system was higher. This was presumably attributed to context switches based on the multitasking operating system used, but was not confirmed within the scope of this thesis.

References

[1] M. Armstrong, *How the world commutes*, Statista Gloval Consumer Survey, 2022.

[2] WHO. "The top 10 causes of death 2019". (2020), [Online]. Available: https://www.who.int/news-room/fact-sheets/detail/the-top-10-causes-of-death (visited on 08/08/2022).

[3] WHO *et al.*, Global status report on road safety 2018 2018, 2018, ISSN: 9789241565684.

[4] R. Lozano, M. Naghavi, K. Foreman, *et al.*, Global and regional mortality from 235 causes of death for 20 age groups in 1990 and 2010: A systematic analysis for the global burden of disease study 2010, *The Lancet* **380**, no. 9859 Dec. 2012, 2095–2128, Dec. 2012. DOI: 10.1016/s0140-6736(12)61728-0.

[5] Y. Page, F. Fahrenkrog, A. Fiorentino, *et al.*, "A comprehensive and harmonized method for assessing the effectiveness of advanced driver assistance systems by virtual simulation: The p.e.a.r.s. initiative", Jun. 2015.

[6] H. Schittenhelm, The vision of accident free driving – how efficient are we actually in avoiding or mitigating longitudinal real world accidents,

[7] M. Ellims, Brake systems: A mind of their own, *Digital Evidence and Electronic Signature Law Review* Mar. 2021, 27–34, Mar. 2021. DOI: 10.14296/deeslr.v18i0.5278.

[8] D. Leblanc, J. R. Sayer, C. B. Winkler, *et al.*, "Road departure crash warning system field operational test: Methodology and results. volume 1: Technical report", 2006. [Online]. Available: https://api.semanticscholar.org/CorpusID:110094475.

[9] J. Fleming, C. Allison, X. Yan, N. Stanton, and R. Lot, Adaptive driver modelling in adas to improve user acceptance: A study using naturalistic data, *Safety Science* **119** Aug. 2018, Aug. 2018. DOI: 10.1016/j.ssci.2018.08.023.

[10] S. Riedmaier, F. Diermeyer, D. Schneider, B. Schick, and D. Watzenig, Model validation and scenario selection for virtual-based homologation of automated vehicles, *Applied Sciences (Switzerland)* **11**, no. 1 2021, 1-24 –24, 2021, ISSN: 20763417. [Online]. Available: https://search.ebscohost.com/login.aspx?direct=true&db=edselc&AN=edselc.2-52.0-85098655288&site=eds-live&scope=site&authtype=guest&custid=s3911979&groupid=main&profile=eds.

[11] J. Wu, C. Flannagan, U. Sander, and J. Bärgman, Modeling lead-vehicle kinematics for rear-end crash scenario generation. 2023, 2023. [Online]. Available: https://search.ebscohost.com/login.aspx?direct=true&db=edsarx&AN=edsarx.2310.08453&site=eds-live&scope=site&authtype=guest&custid=s3911979&groupid=main&profile=eds.

[12] S. Swarup, "Adequacy: What makes a simulation good enough?", *2019 Spring Simulation Conference (SpringSim)*, 2019, pp. 1–12. DOI: 10.23919/SpringSim.2019.8732895.

[13] U. N. E. C. for Europe, *Un regulation no 157 – uniform provisions concerning the approval of vehicles with regards to automated lane keeping systems [2021/389]*, Mar. 2021.

[14] K. Mattas, G. Albano, R. Donà, *et al.*, Driver models for the definition of safety requirements of automated vehicles in international regulations. application to motorway driving conditions, *Accident Analysis & Prevention* **174** 2022, 106743, 2022, ISSN: 0001-4575. DOI: https://doi.org/10.1016/j.aap.2022.106743. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0001457522001798.

[15] M. Balogun, Comparative analysis of complexity of c++ and python programming languages, *Asian J. Soc. Sci. Manag. Technol* **4** 2022, 1–12, 2022.

[16] *Carmaker — ipg automotive*, Online, IPG Automotive GmbH, 2024. [Online]. Available: https://www.ipg-automotive.com/en/products-solutions/software/carmaker/.

[17] J. Dobberstein, J. Bakker, L. Wang, *et al.*, "The eclipse working group openpass–an open source approach to safety impact assessment via simulation", *25th International Technical Conference on the Enhanced Safety of Vehicles (ESV) National Highway Traffic Safety Administration*, 2017.

[18] *Opendrive 1.7*, comp. software, ASAM e.V. [Online]. Available: https://www.asam.net/index.php?eID=dumpFile&t=f&f=4422&token=e590561f3c39aa2260e5442e29e93f6693d1cccd.

[19] *Asam openscenario® xml*, comp. software, ASAM e.V., 2024. [Online]. Available: https://www.asam.net/standards/detail/openscenario-xml/.

[20] B. Krag, "Modeling and simulation—basics and benefits", *In-Flight Simulators and Fly-by-Wire/Light Demonstrators: A Historical Account of International Aeronautical Research*, P. G. Hamel, Ed. Cham: Springer International Publishing, 2017, pp. 19–25, ISBN: 978-3-319-53997-3. DOI: 10.1007/978-3-319-53997-3_3. [Online]. Available: https://doi.org/10.1007/978-3-319-53997-3_3.

[21] D. Watzenig and M. Horn, *Automated driving*. Jan. 2017. DOI: 10.1007/978-3-319-31895-0. [Online]. Available: https://doi.org/10.1007/978-3-319-31895-0.

[22] S. H. Thomke, Simulation, learning and r&d performance: Evidence from automotive development, *Research Policy* **27**, no. 1 1998, 55–74, 1998, ISSN: 0048-7333. DOI: https://doi.org/10.1016/S0048-7333(98)00024-9. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0048733398000249.

[23] E. J. Williams and O. M. Ülgen, "Simulation applications in the automotive industry", *Use Cases of Discrete Event Simulation: Appliance and Research*, S. Bangsow, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 45–58, ISBN: 978-3-642-28777-0. DOI: 10.1007/978-3-642-28777-0_3. [Online]. Available: https://doi.org/10.1007/978-3-642-28777-0_3.

[24] F. Li, "Research on virtual simulation technology in automotive professional culture", *2021 International Conference on E-Commerce and E-Management (ICECEM)*, 2021, pp. 244–247. DOI: 10.1109/ICECEM54757.2021.00055.

[25] I. Krasikov and A. N. Kulemin, Analysis of digital twin definition and its difference from simulation modelling in practical application, *KnE Engineering* **5**, no. 3 Apr. 2020, 105–109, Apr. 2020. DOI: 10.18502/keg.v5i3.6766. [Online]. Available: https://knepublishing.com/index.php/KnE-Engineering/article/view/6766.

[26] J. Schauffele and T. Zurawka, *Automotive Software Engineering*, de, 6th ed., ser. Atz/Mtz-Fachbuch. Springer Vieweg, Sep. 2016.

[27] J. Kovaceva, A. Bálint, R. Schindler, and A. Schneider, Safety benefit assessment of autonomous emergency braking and steering systems for the protection of cyclists and pedestrians based on a combination of computer simulation and real-world test results. *Accident Analysis and Prevention* **136** 2020, 2020, ISSN: 0001-4575. [Online]. Available: https://search.ebscohost.com/login.aspx?direct=true&db=edselp&AN=S0001457519306736&site=eds-live&scope=site&authtype=guest&custid=s3911979&groupid=main&profile=eds.

[28] J. Bärgman, C.-N. Boda, and M. Dozza, Counterfactual simulations applied to shrp2 crashes: The effect of driver behavior models on safety benefit estimations of intelligent safety systems. *Accident Analysis and Prevention* **102** 2017, 165–180, 2017, ISSN: 0001-4575. [Online]. Available: https://search.ebscohost.com/login.aspx?direct=true&db=edselp&AN=S000145751730101X&site=eds-live&scope=site&authtype=guest&custid=s3911979&groupid=main&profile=eds.

[29] Q. Chao, H. Bi, W. Li, *et al.*, A survey on visual traffic simulation: Models, evaluations, and applications in autonomous driving. *Computer Graphics Forum* **39**, no. 1 2020, 287–308, 2020, ISSN: 01677055. [Online]. Available: https://search.ebscohost.com/login.aspx?direct=true&db=bsu&AN=141957445&site=eds-live&scope=site&authtype=guest&custid=s3911979&groupid=main&profile=eds.

[30] *Github - esmini/esmini: A basic openscenario player*, [Online; accessed 10-May-2024], 2024. [Online]. Available: https://github.com/esmini/esmini.

[31] *Esmini/docs/innerworkings.md at master · esmini/esmini · github*, [Online; accessed 10-May-2024], 2024. [Online]. Available: https://github.com/esmini/esmini/blob/master/docs/InnerWorkings.md.

[32] *Powerpoint-präsentation - about-openpass.pdf*, [Online; accessed 10-May-2024], 2024. [Online]. Available: https://openpass.eclipse.org/about/documents/about-openpass.pdf.

[33] *Architecture — openpass — the eclipse foundation*, [Online; accessed 10-May-2024], 2024. [Online]. Available: https://openpass.eclipse.org/architecture/.

[34] *Simcenter simulation software — siemens software*, [Online; accessed 10-May-2024], 2024. [Online]. Available: https://plm.sw.siemens.com/en-US/simcenter/.

[35] *Simcenter prescan - providing a physics-based simulation platform for adas and automated driving*, [Online; accessed 10-May-2024], 2024. [Online]. Available: https://www.plm.automation.siemens.com/media/global/en/Siemens-SW-Simcenter-PreScan-Fact-Sheet_tcm27-76416.pdf.

[36] *Virtual test drive (vtd)*, [Online; accessed 10-May-2024], 2024. [Online]. Available: https://www.asam.net/members/product-directory/detail/virtual-test-drive-vtd/.

[37] *Virtual test drive — hexagon*, [Online; accessed 10-May-2024], 2024. [Online]. Available: https://hexagon.com/products/virtual-test-drive.

[38] *Carmaker / truckmaker / motorcyclemaker for simulink*, Online, The MathWorks, Inc., 2024. [Online]. Available: https://se.mathworks.com/products/connections/product_detail/carmaker.html.

[39] Y. Li, W. Yuan, S. Zhang, *et al.*, Choose your simulator wisely: A review on open-source simulators for autonomous driving, *IEEE Transactions on Intelligent Vehicles* 2024, 1–19, 2024. DOI: 10.1109/TIV.2024.3374044.

[40] *Introduction :: Opendrive®*, Online, ASAM e.V., 2023. [Online]. Available: https://publications.pages.asam.net/standards/ASAM_OpenDRIVE/ASAM_OpenDRIVE_Specification/latest/specification/00_preface/00_introduction.html.

[41] A. e.V., *Introduction :: Openscenario*, Online, 2024. [Online]. Available: https://publications.pages.asam.net/standards/ASAM_OpenSCENARIO/ASAM_OpenSCENARIO_XML/latest/00_preface/01_introduction.html#_what_is_asam_openscenario.

[42] *7.2 storyboard and entities :: Openscenario*, Online, ASAM e.V., 2024. [Online]. Available: https://publications.pages.asam.net/standards/ASAM_OpenSCENARIO/ASAM_OpenSCENARIO_XML/latest/07_components_scenario/07_02_storyboard_entities.html.

[43] *2.2.1 overview of osi architecture :: Asam osi (open simulation interface)*, Online, ASAM e.V., 2023. [Online]. Available: https://opensimulationinterface.github.io/osi-antora-generator/asamosi/latest/interface/architecture/architecture_overview.html.

[44] *Asam osi®*, comp. software, ASAM e.V., 2024. [Online]. Available: https://www.asam.net/standards/detail/osi/.

[45] P. Rechenberg and G. Pomberger, Eds., *Informatik-Handbuch*, de, 4th ed. Munich, Germany: Hanser, Mar. 2006.

[46] M. Shaw, Abstraction techniques in modern programming languages, *IEEE Software* **1**, no. 4 Oct. 1984, 10–26, Oct. 1984. DOI: 10.1109/MS.1984.229453.

[47] LearnComputerScience. "Machine instructions". (), [Online]. Available: https://www.learncomputerscienceonline.com/machine-instruction/ (visited on 08/14/2022).

[48] Brooks, No silver bullet essence and accidents of software engineering, *Computer* **20**, no. 4 1987, 10–19, 1987. DOI: 10.1109/MC.1987.1663532.

[49] H. Austerlitz, "Chapter 13 - computer programming languages", *Data Acquisition Techniques Using PCs (Second Edition)*, H. Austerlitz, Ed., Second Edition, San Diego: Academic Press, 2003, pp. 326–360, ISBN: 978-0-12-068377-2. DOI: https://doi.org/10.1016/B978-012068377-2/50013-9. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780120683772500139.

[50] B. Stroustrup, *Programming: Principles and Practice Using C++ (2nd Edition)*, 2nd. Addison-Wesley Professional, 2014, ISBN: 0321992784.

[51] ——, "C++ applications". (2021), [Online]. Available: https://www.stroustrup.com/applications.html (visited on 08/08/2022).

[52] D. Kuhlman, *A Python Book: Beginning Python, AdvancedPython, and Python Exercise*. 2013.

[53] J. Guttag, *Introduction to Computation and Programming Using Python, second edition: With Application to Understanding Data*, ser. The MIT Press. MIT Press, 2016, ISBN: 9780262529624. [Online]. Available: https://books.google.se/books?id=KabKDAAAQBAJ.

[54] TIOBE. "The python programming language". (2022), [Online]. Available: https://www.tiobe.com/tiobe-index/python/ (visited on 08/08/2022).

[55] S. Cass, Top programming languages: Our eighth annual probe into what is hot and not, *IEEE Spectrum* **58**, no. 10 2021, 17–17, 2021. DOI: 10.1109/MSPEC.2021.9563957.

[56] M. Treiber, A. Hennecke, and D. Helbing, Congested traffic states in empirical observations and microscopic simulations, *Physical Review E* **62**, no. 2 Aug. 2000, 1805–1824, Aug. 2000, ISSN: 1095-3787. DOI: 10.1103/physreve.62.1805. [Online]. Available: http://dx.doi.org/10.1103/PhysRevE.62.1805.

[57] K. D. Kusano and H. C. Gabler, Safety benefits of forward collision warning, brake assist, and autonomous braking systems in rear-end collisions, *IEEE Transactions on Intelligent Transportation Systems* **13**, no. 4 2012, 1546–1555, 2012. DOI: 10.1109/TITS.2012.2191542.

[58] G. Markkula, "Driver behavior models for evaluating automotive active safety : From neural dynamics to vehicle dynamics.", Ph.D. dissertation, 2015, ISBN: 9789175971537. [Online]. Available: https://search.ebscohost.com/login.aspx?direct=true&db=cat07470a&AN=clc.0055cace.673d.4f0b.804b.dfd5e7ef283d&site=eds-live&scope=site&authtype=guest&custid=s3911979&groupid=main&profile=eds.

[59] M. Svärd, G. Markkula, J. Engström, F. Granum, and J. Bärgman, A quantitative driver model of pre-crash brake onset and control, *Proceedings of the Human Factors and Ergonomics Society Annual Meeting* **61**, no. 1 Sep. 2017, 339–343, Sep. 2017. DOI: 10.1177/1541931213601565.

[60] M. Svärd, G. Markkula, J. Bärgman, and T. Victor, Computational modeling of driver pre-crash brake response, with and without off-road glances: Parameterization using real-world crashes and near-crashes, *Accident Analysis Prevention* **163** Dec. 2021, 106433, Dec. 2021. DOI: 10.1016/j.aap.2021.106433.

[61] W. Bugden and A. Alahmar, The safety and performance of prominent programming languages. *International Journal of Software Engineering amp; Knowledge Engineering* **32**, no. 5 2022, 713–744, 2022, ISSN: 02181940. [Online]. Available: https://search.ebscohost.com/login.aspx?direct=true&AuthType=sso&db=bsu&AN=157613445&site=ehost-live&scope=site&authtype=sso&custid=s3911979.

[62] O. Bröker, O. Chinellato, and R. Geus, Using python for large scale linear algebra applications, *Future Generation Computer Systems* **21**, no. 6 2005, 969–979, 2005, ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2005.02.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X05000178.

[63] F. Zehra, M. Javed, D. Khan, and M. Pasha, Comparative analysis of c++ and python in terms of memory and time, *Preprints* Dec. 2020, Dec. 2020. DOI: 10.20944/preprints202012.0516.v1. [Online]. Available: https://doi.org/10.20944/preprints202012.0516.v1.

[64] P. Svensson and F. Galfi, "Performance evaluation of numpy, scipy, pymel and openmaya compared to the c++ api in autodesk maya", Ph.D. dissertation, 2021. [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:bth-21664.

[65] L. Prechelt, An empirical comparison of seven programming languages, *Computer* **33**, no. 10 2000, 23–29, 2000. DOI: 10.1109/2.876288.

[66] Z. Alomari, O. E. Halimi, K. Sivaprasad, and C. Pandit, *Comparative studies of six programming languages*, 2015. arXiv: 1504.00693 [cs.PL]. [Online]. Available: https://arxiv.org/abs/1504.00693.

[67] R. Pereira, M. Couto, F. Ribeiro, *et al.*, "Energy efficiency across programming languages: How do energy, time, and memory relate?", *Proceedings of the 10th ACM SIGPLAN international conference on software language engineering*, 2017, pp. 256–267.

[68] *Python 3 versus C++ g++ fastest performance (Benchmarks Game)*, https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/python3-gpp.html, [Accessed 04-07-2024].

[69] C. R. Harris, K. J. Millman, S. J. van der Walt, *et al.*, Array programming with NumPy, *Nature* **585**, no. 7825 Sep. 2020, 357–362, Sep. 2020. DOI: 10.1038/s41586-020-2649-2.

Appendices

A. Used System, Standards and Toolchain

System

Name of System Part	Used
Operating System	PopOS_!20.04
CPU	Intel® Core i7-4800MQ
GPU	Intel® HD Graphics 4600
RAM	16 GB

Standards

Standard	Version
ASAM OpenDRIVE	1.0
ASAM OpenSCENARIO	1.0
ASAM Open Simulation Interface	1.0
C++ Standard	17

Toolchain

Name	Version
g++	9.4.0
Python	3.10.6
esmini	2.37.0
protobuf	3.15.2
cmake	3.5
open-simulation-interface	3.4.0

B. Source Code, Measurements, Tables and Histograms

The written code (driver model, UDP class, custom-made Benchmark, bash and evaluation scripts), the measured execution times, created tables and histograms are available on `https://github.com/BurakCSoydas/MasterThesis`. Please note that the parts of the accumulator driver model based on the code provided by Volvo have been replaced with pseudocode to honor Volvos wishes and protect the intellectual property.

C. Bash Scripts

The bash scripts were used to loop the benchmarks a wanted number of times. To achieve this, multiple bash scripts were combined. Every driver model was stored in its own folder. In each of these folders, 8 bash scripts were created to run the local driver model in a specific benchmark configuration once. An extra bash script called LoopAll.sh combined the other 8 scripts into one, allowing to run every benchmark configuration by calling one script. The LoopAll.sh script also took an argument to determine how often it should run the benchmark configurations. A bash script called LoopBenchmarks.sh and located in the parent folder to each of the driver model folders, called all LoopAll.sh scripts and provided them with the same input for the number of repetitions.

RunHCpp.sh - Example script to run a driver model in a specific benchmark configuration once. The scripts for other benchmark configurations were named accordingly.

```
#!/bin/bash
{ \time -f "%e, %U, %S " ../Simulation/scenarioStart -b 2 -m H-Cpp 1>/dev/null ; }  2>&1 |
    cut -d"y" -f2 >> ../output/timeEMCppH-Cpp.csv &
{ \time -f "%e, %U, %S " ./DM -b 2 -n 648 -m H-Cpp ; } 2>> ../output/timeCppH-Cpp.csv
```

Listing 1: Example bash script for the C++ driver model in headless mode and no logging. The different benchmark configurations were achieved by modifuying the input arguments of both simulation components.

LoopAll.sh - Script combining the others to run all benchmark configurations

```
#!/bin/bash
echo ""
echo --------------------------------------------------------------------------------
echo ""
declare -i loop=1
if (( $# == 1 )); then
  loop=$1
  echo input nr loops: $loop
fi

echo Benchmarking Cpp

for ((i = 1 ; i < $loop+1 ; i++ )); do
  echo "---Loop: $i--- ---runVLPy ---"
  bash runVLPy.sh
  echo "---Loop: $i--- ---runVPy ---"
  bash runVPy.sh
  echo "---Loop: $i--- ---runHPy ---"
  bash runHPy.sh
  echo "---Loop: $i--- ---runHLPy ---"
  bash runHLPy.sh
  echo "---Loop: $i--- ---runVLCpp ---"
  bash runVLCpp.sh
  echo "---Loop: $i--- ---runVCpp ---"
  bash runVCpp.sh
  echo "---Loop: $i--- ---runHCpp ---"
  bash runHCpp.sh
  echo "---Loop: $i--- ---runHLCpp ---"
  bash runHLCpp.sh
done
```

LoopBenchmarks.sh - calls the LoopAll bash scripts of all driver model implementations. It also activated a virtual environment used for the Python development, using pyenv, if the virutal environment was not already activated.

```
 #!/bin/bash
declare -i loop=1
if (( $# == 1 )); then
    loop=$1
fi

if [[ -z "$VIRTUAL_ENV" ]]; then
pyenv activate thesis;
fi

maindir="/home/t440pop/Desktop/finalCode/"
```

```
subdir=("Cpp" "PyClass" "PyDataclass" "PyDict" "PyList")

for dir in ${subdir[@]}; do
  temp="$maindir$dir"
  cd $temp
  echo ""
  echo "switched to $temp"
  source LoopAll.sh $loop
done

cd ..
```