**Web Simulator Interface for Autonomous Vehicle Software Stack.**

**This thesis is confidential.**

|  |  |
| --- | --- |
|  | |
| Master Thesis 3263 | |
| Submitted at the University of Stuttgart by  **Sarthak Bapat** | |
| INFOTECH | |
|  | |
| Examiner: | Prof. Dr.-Ing. Michael Weyrich |
| Supervisor: | Msc. Hannes Vietz |
| External Supervisor: | Prof. Dr.-Ing. Christof Ebert |
|  | 19/01/2022 |

# **Table of** **Contents** Style Template: Überschrift 1 / Headline 1 (delete the number)

Table of Contents *Style Template: Überschrift 1 / Headline 1 (delete the number)* ii

Table of Figures iv

Table of Tables v

Table of Abbreviations *Style Template: Überschrift 1 / Headline 1 (delete the number)* vi

Glossary *Style Template: Überschrift 1 / Headline 1 (delete the number)* vii

Abstract viii

1 Introduction 9

1.1 What is Autonomous Driving Software? 9

1.2 Current Setup 10

1.3 Scope of the Thesis 10

2 Analysis and Design of AV Software stack 12

2.1 Existing Interface Architecture 12

2.1.1 Apollo AV software stack 13

2.1.2 Apollo Cyber-RT 13

2.2 Interface options between Simulators and Apollo 13

2.2.1 Cyber RT Interface 13

2.2.2 ROS or ROS2 Interface 15

2.3 Problems with VTD connection to Apollo 16

2.3.1 Camera Video Data Stream 16

2.3.2 VTD LiDAR Sensor Point Cloud 16

2.3.3 HD Map conversion 16

2.4 Decision on flexible Software Interface 17

2.5 Which AV stack with VTD? 18

2.5.1 VTD Internal AV stack 18

2.5.2 Using VTD Internal AV stack 18

3 System Model 21

3.1 Use Case Diagram 21

3.2 Use Case Description 22

3.3 Sequence Diagrams 24

3.4 GUI Concept 24

3.4.1 Principle concepts of operation 24

3.4.2 Elements of the User Interface 24

3.4.3 Structure and Design of Windows 24

3.4.4 Design of Dialogs 25

4 Implementation 26

4.1 System Architecture 26

4.1.1 Development Environment 26

4.1.2 System Architecture Diagram 26

4.1.3 Database Design 28

4.2 VTD Interface with RoboTest Web Application 30

4.3 Software Components 31

4.3.1 Software Component VTD Manual Test Editor 31

4.3.2 Software Component VTD Connector 34

5 Simulation Results 37

6 Traceability and Test Automation 38

6.1 Traceability between Requirements and Test Results 38

6.2 Automated Test Case Generation 40

Bibliography 42

Declaration of Compliance *Style Template: Überschrift 1 / Headline 1 (delete the number)* 45

# Table of Figures

**No table of figures entries found.**

# Table of Tables

**No table of figures entries found.**

# Table of Abbreviations Style Template: Überschrift 1 / Headline 1 (delete the number)

Convention: Mark the letters that make up the abbreviations by making them bold

|  |  |
| --- | --- |
| ASCII  ROS  AV  VTD | **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange  **R**obot **O**perating **S**ystem  **A**utonomous **V**ehicle  **V**irtual **T**est **D**rive |

# Glossary Style Template: Überschrift 1 / Headline 1 (delete the number)

Style Template: IAS\_Glossary (make main term bold)

|  |  |
| --- | --- |
| **Actuator** | Component for the conversion of an information carrying, low-energy control signal to a high energy signal capable of process intervention |

# Abstract

**Key Words:**

# Introduction

Autonomous systems in general are widely used in today’s world. Autonomous driving has been the latest trend in the market with rapid advancement in the AI, sensor, and the software technology. Major players in the automotive market are conducting research in autonomous driving and shifting towards increased levels of autonomy. A completely autonomous car with level 5 autonomy is expected to drive without any human intervention and handle any situation safely. This means that the vehicle should be intelligent enough to be able to take decisions in worst of the situations that are on par with the human decisions [1]. For this, the autonomous vehicle should be tested thoroughly under various scenarios such that they are safe enough to operate on roads. That brings us to the most important and wide research topic in autonomous driving – Validation of Autonomous Driving.

Before an autonomous vehicle is made available in the market, it must be validated and deemed fit enough to operate safely on the roads under any circumstances. The public trust in autonomous vehicles can only be achieved by continuous validation of the autonomous vehicle software [1]. One way of performing the validation is to directly test the vehicle on road. This technique, though possible to use is not convenient in terms of the number of miles that need to be driven to certify safety of autonomous vehicle and poses threats to the human life and property. To demonstrate that the failure rate of autonomous vehicles is statistically significantly lower than the human driver failure rate they need to be driven an approximately 5 billion miles [2]. This is highly inconvenient validation activity and it would take years to cover all weather conditions, road and traffic conditions in different parts of the world.

The other way of validation is to perform the simulation of the scenarios and test the performance of the autonomous vehicle software. A scenario consists of different weather, road, light and traffic conditions in different parts of the world, on different terrains and with contrasting driving conventions. For the autonomous vehicle software to be deemed safe, it must perform optimally under various scenarios as mentioned. The autonomous vehicle under test in the simulation is controlled by the autonomous vehicle software depending on the inputs received from the simulation environment. In this master thesis the validation of autonomous driving system using the simulation technique is considered.

## What is Autonomous Driving Software?

Autonomous driving software is the main component or the control unit of the autonomous vehicle. It basically receives the environment data from various sensors embedded in the vehicle, processes and makes decisions on the received data and sends the control commands back to the autonomous vehicle. The control commands include the throttle, gear, brake and acceleration values that determine the next position and the action of the autonomous vehicle. Any autonomous vehicle software stack consists of Localization, Perception, Prediction, Planning and Control as its core components.

The ‘Localization’ module is responsible for single digit centimeter level accuracy localization of the vehicle in the environment. Generally, the data from GPS, IMU (Inertial Measurement Unit) and LiDAR sensors is input to this component. ‘Perception’ module takes in the data from LiDAR, RADAR, Camera sensors and performs tasks such as classification, detection, segmentation and learning convolutional neural networks to perceive the surroundings of the vehicle. ‘Prediction’ module predicts the movement of other traffic elements like cars and pedestrians with respect to the autonomous vehicle. This information is consumed by the ‘Planning’ module to generate the right trajectory for the vehicle. Finally, the ‘Control’ component ensures that steering, throttles and brake commands are issued to the vehicle to execute the planned trajectory [3].

## Current Setup

The Robo-Test demonstrator has the LGSVL simulator connected to the Apollo autonomous vehicle software stack, a proprietary simulator VTD (Virtual Test Drive) and a web interface designed to create the test scenarios for the LGSVL as well as the VTD simulators. The web application was designed in MT-3136 [4] and VTD was introduced and integrated at a basic level in the web application in MT-3183 [5].

MT-3192 [6] has established an interface between the VTD simulator and the Apollo autonomous software stack using ROS [6]. The interface doesn’t support all the sensor data exchange between VTD and Apollo, also it introduces a few problems that are analyzed further in this thesis.

## Scope of the Thesis

This master thesis explores the possibility of creating a flexible interface between the LGSVL, VTD simulators and the Apollo software stack using ROS or ROS2. A feasibility check regarding a ROS based interface is performed before taking the decision. Depending on the outcome of the feasibility check, the existing Robo-Test web application interface is enhanced with new features and the details are discussed in the sections to follow.

Furthermore, in this thesis a detailed literature survey regarding the traceability techniques between the requirements and the test results is performed and a concept has been worked out for the current setup of the Robo-Test system. The concept tries to link the requirements engine to the test execution and test results. A detailed literature review regarding the techniques for automatic generation of test cases from requirements is also presented.

The rest of the report is structured as follows. Chapter 2 discusses the analysis of the current system and the fundamental design decisions. In the Chapter 3, we discuss the software architecture and corresponding system model. The implementation details are discussed in the Chapter 4, followed by the results in Chapter 5. Chapter 6 presents the concept for traceability for our system along with a literature review of the automated test case generation techniques. User manual is provided in the Chapter 7, followed by the conclusion.

# Analysis and Design of AV Software stack

For the selection and connection of an autonomous vehicle software stack with LGSVL and VTD simulators using a flexible software interface, the analysis of the current system architecture must be performed and the interface options must be explored. In the following sections, we will analyze these aspects and arrive to a decision with the help of certain criteria.

## Existing Interface Architecture

Let’s understand the existing interface architecture for our simulator and autonomous software stack system.

Robo-Test Web Application

VTD Connector

LGSVL Connector

VTD Simulator

LGSVL Simulator

ROS Interface

Apollo Cyber RT

Apollo 5.0

Apollo 6.0

Figure 2.1: Architecture of Simulators and Apollo AV software stack.

The architecture shown above consist of the VTD and LGSVL simulators connected to the Robo-Test web application using the individual connector components. The web application is a tool developed to generate test scenarios easily, without knowing the process to create scenarios on different simulators. The process and scenario creation details for the LGSVL simulator using the web application can be found in MT-3136 [4], whereas the details for VTD are in MT-3183 [5].

Both the simulators are connected to different versions of Apollo software stack using different interfaces. The LGSVL connects to Apollo 5.0 using the Cyber-RT based interface and the VTD connection is using ROS interface to Cyber-RT developed in MT-3192 [6]. The Cyber-RT layer is a common layer that connects both the simulators to Apollo. To better understand this, let’s look at these concepts in more details.

### Apollo AV software stack

Apollo is an open-source, high performance, flexible software architecture developed by Baidu for development, testing and deployment of autonomous vehicles [7]. It provides all the AV software stack components like localization, perception, prediction, planning, control and its own HD map format specifications. The Apollo versions upto 3.0 support the ROS interface to connect any external component. Since Apollo 3.5, Baidu has developed a robust Cyber-RT layer that takes care of Apollo’s interactions with the external components. So, the versions starting from 3.5 don’t support a ROS interface with Apollo directly as there is a Cyber-RT layer in the middle.

### Apollo Cyber-RT

The Apollo Cyber-RT is a robust framework and offers developers with a high-performance computing to support complex tasks of autonomous driving [8]. It enables a high concurrency of execution, low latency and high throughput [8]. The newer versions of Apollo have improved since the introduction of Cyber-RT making it one of the most robust open-source AV software stacks available. Integration of external software components with Cyber-RT framework is easy due to its plug-and-play architecture.

## Interface options between Simulators and Apollo

As per the current architecture of our system, we have Apollo versions 5.0 and 6.0 which come with a Cyber-RT layer on top of it. Thus, any interface for our simulators with Apollo must interact with Cyber-RT layer to establish communication with Apollo. The analysis of different possibilities of interface with Apollo is discussed in this section.

### Cyber RT Interface

One way of connecting both the simulators to Apollo AV software stack is to interface them directly with Apollo’s Cyber-RT layer. The LGSVL simulator offers this functionality and offers support for a connection with Cyber-RT. In the architecture that we showed, the LGSVL is interfaced to Apollo using Cyber-RT without any additional middleware component. The LGSVL simulator offers four types of bridges where each bridge has its own message format – ROS, ROS Apollo, ROS2 and Cyber-RT [9]. Using the Cyber-RT message format, it is convenient to exchange data between both the components. Let us look at a couple of important factors that must be considered for the interface between the simulators and the AV software stack with respect to our system.

#### Addition of a new Simulator

What if a new simulator must be integrated into our system? A standardized middleware interface layer will ensure the flexibility to accommodate a new simulator. The table below shows a few simulators and the interface support that they provide.

Table 2.1: Simulators and Interface Support.

|  |  |  |
| --- | --- | --- |
| Simulator | Type | Interface Support |
| LGSVL | Open Source | ROS, ROS2, ROS Apollo and Cyber-RT. |
| VTD | Proprietary | ROS and ROS2. |
| Carla | Open Source | ROS and ROS2. |

Table 2.1 indicates that the VTD simulator does not explicitly offer any support for Cyber-RT interface data communication. Being a proprietary simulator, there are quite a few limitations on the exchange of data into and out of VTD. Since there is no Cyber-RT message format supported in VTD, there is a need of an additional layer to convert the messages into Cyber format and vice versa. Also, most simulators provide a ROS/ROS2 based interface support for further data exchange.

#### Migration to a new AV software stack

How about the software interface if a migration to a new AV software stack is desired? The following table contains different AV software stacks and the interface support they provide to communicate with the external components.

Table 2.2: AV stacks and Interface Support.

|  |  |  |
| --- | --- | --- |
| AV Stack | Type | Interface Support |
| Baidu Apollo | Open source | Apollo Cyber-RT |
| Autoware | Open source | ROS migrating to ROS2 |
| Nvidia Drive | Proprietary | NVIDIA Driveworks SDK |

In case of our system, the possibility of migration to Autoware AV stack is high as it is an open-source AV stack and supports open-source ROS based interface for data exchange. Based on the inputs from the above tables, a ROS/ROS2 based interface looks a better option. Thus, the idea of having a single Cyber-RT interface that handles data exchange for both the simulators is not possible due to the factors analyzed above. Moreover, the VTD doesn’t support a Cyber-RT type message exchange, so designing a full Cyber-RT interface is not an option.

### ROS or ROS2 Interface

The analysis so far suggests that it is better to design a ROS/ROS2 based software interface to connect the LGSVL and the VTD simulators to Apollo. There already exists a ROS interface between the VTD and the Apollo but the system is not fully functional with this interface. Also, the LGSVL is connected to Apollo using its Cyber-RT message support. Thus, we explore the possibility of a ROS2 interface between both the simulators and the Apollo AV software stack by migrating and enhancing the existing ROS interface. A comparison between ROS and ROS2 is presented based on certain features.

Table 2.3: Comparison between ROS and ROS2 [10].

|  |  |  |
| --- | --- | --- |
| Feature | ROS | ROS2 |
| Multi-machine application | Set one machine as ‘master’. | No ‘master’ required. Independent working. |
| ROS Master | Required. | No more necessary. |
| Parameter Server | Global parameter server. | Local parameters to nodes. |
| Underlying Middleware | Central discovery mechanism. | Based on DDS standard. |
| Nodes in a process | One node per process. | Multiple nodes per process. |
| Roslaunch | XML | Python (Complex logics) |
| Language Support | C++ 03 and Python 2. | C++ 11,14 and Python 3.5 or greater. |

Table 2.3 compares ROS and ROS2 on a few features and ROS2 prevails better due to its distributed mechanism, support for modern C++ and Python for application development, multi-master capabilities, multiple node support in a process and good support for a next few years depending on the version.

## Problems with VTD connection to Apollo

As discussed above, there is a ROS interface between the VTD and Apollo software stack which was developed in MT-3192 [6]. The work puts forth a few problems that were encountered during the design and development of this ROS interface to connect the VTD and Apollo. This analysis is very important for our decision regarding the software interface between simulators available in our setup and the Apollo stack. The issues that occurred are discussed below in details.

### Camera Video Data Stream

The camera sensor data includes the camerainfo, image or compressed image provided by VTD. Since, sending the video image data over the TCP/UDP network may cause overload issues, the VTD does not support to take image data as a sensor plugin. Only the information regarding the image is available at the port 48196 using the TCP communication. A VTD API allows to save the image of every frame in a video to a local memory and it can be then published to the ROS topic. So basically, this approach introduces a delay in the camera sensor video data exchange with the ROS server [6].

Analysis of possible options to address this issue was done by contacting the VTD support team. We proposed the use of an API that streams the data without saving each frame to the local memory and sought feedback from the VTD support. The VTD support team confirmed that the use of streaming API would result in a poor quality of the camera sensor video data. No other API is available in VTD to address the video data issue in a different way. The approach to save every image frame by frame and publish it to ROS would cause performance issues in the system and is thus discarded.

### VTD LiDAR Sensor Point Cloud

The LiDAR sensor point cloud data collection from the VTD causes troubles in the current ROS interface system. The simulation hangs in between if the LiDAR sensor plug-in is enabled in the Module Manager configuration file. Disabling the LiDAR sensor plug-in makes the system function properly [6].

Disabling the LiDAR sensor data is not a solution as point cloud data from the LiDAR sensor is an important input to Apollo’s localization and perception modules. Incorrect or loss of data as inputs to these modules would lead to an incorrect control of the autonomous vehicle by the Apollo AV software stack.

### HD Map conversion

HD maps are being used as a fast-track solution to boost the self-driving ability of the autonomous vehicles. The self-driving vehicle understands the world with the help of a HD map. HD map basically filters out the 3D graphics and simplifies the road layout, traffic lights and other objects in it [6].

VTD uses the standard OpenDrive format for the road layout and OpenSceneGraph for the 3D graphics. Apollo on the other hand uses its own modified version of OpenDrive format. Since VTD does not have any communication interface with Apollo, there is no implicit map conversion from VTD’s format to Apollo’s format. One way is to convert standard HD map to Apollo HD map using Apollo’s OpenDrive specifications. But Apollo has stopped sharing its OpenDrive specifications to the outside world. The other way is to use some software like Roadrunner and Unity LGSVL Map Annotation to convert standard maps into HD maps. The problem with Unity LGSVL tool is that it supports OpenDrive version 1.4 whereas the VTD maps are in version 1.5. Also, the HD map is generated without road boundary and junctions. In the Roadrunner tool provided by MathWorks, only simple maps are converted correctly. Complex maps generally end up getting converted with loss of features or sometimes the conversion crashes. Moreover, the location of the Ego vehicle is also incorrect on the new generated HD map [6].

The HD map conversion to Apollo’s format is an important step as Apollo’s modules will work based on the vehicle’s location in its HD map. For the simulator and Apollo to function as expected an accurate conversion of HD map is required, which unfortunately lacks at the moment.

## Decision on flexible Software Interface

The above sections present a detailed analysis from the perspectives of the LGSVL and VTD simulators as well as the Apollo AV stack. Analysis of the existing ROS interface between the VTD and Apollo is also presented. Also, possible interface options that suit both the simulators are discussed keeping in mind the possibility of integrating a new simulator or migration to a different AV software stack in the future. A discussion of the important factors that affect the design of the flexible interface is summarized below.

1. The analysis suggests that a ROS based interface is better than the Cyber-RT interface given the current status quo of our system.
2. An interface developed using ROS2 would benefit given that ROS2 is more advanced than ROS and benefits from a better community support.
3. The analysis of existing ROS interface between VTD and Apollo highlights a few problems from the VTD end discussed in section 2.3. The issues discussed are important from the system performance and functionality perspective. Migration of this interface into a new ROS2 interface would add no value to the system unless the VTD can provide API’s that solve these data exchange issues in a better way.
4. From the LGSVL perspective, it is connected to Apollo using the Cyber-RT bridge. A new ROS2 interface in between could mean an additional layer that is not necessary for LGSVL to function. Though a standard interface in between simulators and Cyber-RT + Apollo would benefit the system from a flexibility perspective as discussed earlier, the problems from the VTD end have ensured that a new interface would add a little value and further complicate our existing system.

Considering all the factors discussed, a new interface between both the simulators and the Apollo AV software stack is not a value adding option. Therefore, this idea is not taken into consideration further.

## Which AV stack with VTD?

The decision of not having an interface in between the simulators and Apollo has brought up a question – Which AV stack to use with VTD simulator? Since, the LGSVL is connected to Apollo using a Cyber-RT bridge there is no need to contemplate a new AV stack for LGSVL.

Rather than exchange of VTD data with an external component such as any AV software stack, utilization of VTD’s internal AV software stack could be a better solution. Our system will benefit from this as there is no need of any interface to exchange the data in and out of VTD. In the below sub-section, we will discuss the usage of VTD’s AV stack with our system.

### VTD Internal AV stack

Vehicle dynamics and Driver are two important components inside VTD that are responsible for the control of the Ego and other traffic elements in VTD. They can be internal components from the VTD or external user defined components like Apollo AV software stack to control the vehicles [11]. The functions of these two components are as described.

1. **Driver**: It sends the control commands like steering, speed, brake and acceleration to the traffic elements. Driver belongs to VTD’s task control module whose output is available at port 48190. Either an internal or external VTD driver can be selected at one time, else there is a conflict [11].
2. **Vehicle Dynamics**: Vehicle dynamics utilize the driver control commands and different vehicle parameters to calculate the updated position of the vehicle. VTD offers a configurable simple and a complex vehicle dynamic [11]. In this thesis, a simple vehicle dynamic has been utilized.

A quite sophisticated model for driver is available in VTD’s internal implementation. It is referred as ‘ghostdriver’ or VIRES driver. This driver model is responsible for the control of all traffic elements in the VTD [12]. In this thesis, the internal VTD driver has been utilized as an AV software stack with the VTD simulator.

### Using VTD Internal AV stack

The VTD’s internal AV stack can be used and tested with more complex scenarios where the tester has a choice to apply certain actions to the scenarios and configures different driver types for different scenarios. This allows to test the behavior of internal AV stack under different scenarios. The Player Action Control Concept in VTD is discussed below to understand this better.

#### Action Control Concept

In VTD, a deterministic behavior of a player is achieved by assigning it with some specific actions. By default, the traffic module controls the vehicles autonomously and ensure that they follow the road, keep distance between each other and obey the recommended speed. The action control concept allows the tester to test the Ego vehicle with some action like Lane change, Speed change or Autonomous control. Player actions are linked with trigger points which ensure certain conditions are fulfilled before an action set by the tester becomes active. In general, the actions can be added for any player in the simulation, but this master thesis focuses on the actions for Ego vehicle [13].

Let’s discuss about the types of triggers that could be created to associate an action with the Ego vehicle.

1. **Absolute Trigger:** This trigger is in absolute (inertial) co-ordinates and the actions associated with it would get activated inside a circular range around the Ego vehicle called activation radius [13].
2. **Relative Trigger:** This trigger is activated when the distance between the selected player i.e., the Ego vehicle and the pivot player is less than the activation radius. A pivot player can be the Ego vehicle, or any other vehicle present in the simulation scenario. But for relative trigger it is necessary that the pivot is anything other than the Ego vehicle [13].

For the players in VTD, three actions are available which could be active concurrently. We will discuss all with respect to the Ego vehicle as the focus is the same.

1. **Action Autonomous:** This action controls the Ego vehicle autonomously using the VTD simulators internal AV software stack. If the force option is enabled, this action interrupts other running actions if any [13].
2. **Action Lane Change:** If this action is enabled, the Ego vehicle will change the lane. The direction of lane change could be left or right and is selected using the Directions field in the test scenario editor. The duration of the lane change is set using the fields Type and Time [13].
3. **Action Speed Change:** In this action, the Ego vehicle will change its speed from the current speed to the target speed that is set, with the rate of speed change set by the tester [13].

There are a few attributes that are common to all the actions that we just discussed.

1. **Delay Time:** It specifies the time in seconds by which the activation of given action is delayed after the trigger condition is met [13].
2. **Activation Event:** It specifies whether the action is active on enter (when the conditions are met) or on exit (when the conditions are false again) [13].
3. **Number of Executions:** It specifies the number of times an action is executed in one run of the simulation. It could be any positive number, infinite or never [13].

In our system, a web application Robo-Test allows the tester to create different scenarios and run the simulation using LGSVL and the VTD simulators. In this thesis, the VTD scenario creation using the web application is enhanced with the action control concept discussed above to utilize and test the VTD AV software stack under different driver types and the scenarios. Furthermore, the test results of the simulation are also captured and displayed on the web application.

# System Model

In this chapter, let us discuss the model for our system. We will look at the use-case diagram for our system and discuss the important use-cases. The existing use cases of the Robo-Test web application for the LGSVL simulator remain the same whereas for VTD two new use cases are added. The use case ‘Manual Test Editor’ is modified and improved to include a new use case – ‘Select Driver’ which further extends the ‘Add Player Actions’ use case. The database use case stays the same, but the functionality is enriched to accommodate the data generated from the new use case.

## Use Case Diagram

Diagram

Description automatically generated

Figure 3.1: Use case diagram

## Use Case Description

Table 3.1: Use case ‘Select Driver Type’ description.

|  |  |  |
| --- | --- | --- |
| **USE CASE** | *Select Driver Type* | |
| **Objective** | To allow the tester to select the VTD driver type for the EGO vehicle. | |
| **Category** | Mandatory | |
| **External Actors** | Tester | |
| **Preconditions** | Tester is successfully logged into the application and is creating a test scenario. | |
| **Success End Condition** | Driver type is added and saved to the database. | |
| **Failed End Condition** | Tester is not able to select the desired driver type and data is not saved to the database successfully. | |
| **Trigger** | Tester starts creating a test scenario. | |
| **DESCRIPTION** | **Step** | **Action** |
|  | *1* | The application renders the page with Player Action options when requested by the tester. |
|  | *2* | Selected option by the tester for driver type is stored in the database. |
|  | *3* | Further actions are handled by the use case ‘Database’. |
| **EXTENSIONS** | **Step** | **Branching Action** |
|  | *1a* | None. |
| **SUB-VARIATIONS** |  | **Branching Action** |
|  | *1a* | None. |

Table 3.2: Use case ‘Add Player Actions’ description.

|  |  |  |
| --- | --- | --- |
| **USE CASE** | *Add Player Actions* | |
| **Objective** | To allow the tester to set various actions on EGO vehicle and enable autonomous control. | |
| **Category** | Optional | |
| **External Actors** | Tester | |
| **Preconditions** | Tester is successfully logged into the application and is creating a test scenario. | |
| **Success End Condition** | Player Actions are added and saved to the database. | |
| **Failed End Condition** | Tester is not able to create player actions and data is not saved to the database successfully. | |
| **Trigger** | Tester starts creating a test scenario. | |
| **DESCRIPTION** | **Step** | **Action** |
|  | *1* | The application renders the page with Player Action options when requested by the tester. |
|  | *2* | Entered values by the player are stored in the respective database. |
|  | *3* | Further actions are handled by the use case ‘Database’. |
| **EXTENSIONS** | **Step** | **Branching Action** |
|  | *1a* | None. |
| **SUB-VARIATIONS** |  | **Branching Action** |
|  | *1a* | None. |

## Sequence Diagrams

The sequence diagrams for these use cases are similar to that of the ‘Manual Test Editor’ use-case and can be referred in MT-3136 [4]. Therefore, the sequence diagrams are not discussed in this thesis.

## GUI Concept

Here, the main factors pertaining to the design of the UI in the web application are discussed in detail.

### Principle concepts of operation

The new web pages designed for the tester to add more details in the scenario creation should follow the existing template of the web application. The integration of new web pages should not break the existing functionality of the web application. The GUI should support different resolutions for the display. Also, the web GUI should contain enough information text to guide the tester through the scenario creation process.

### Elements of the User Interface

In synchronization with the existing design, the GUI must contain options to navigate to the previous and the next page. Consistency of design with the current GUI must be maintained. The other important details such as the login, homepage are not affected by integration of the new web GUI pages.

### Structure and Design of Windows

The newly added actions page allows the tester to add three actions, the autonomous action, lane change action and the speed change action. All the three actions have similar parameters that must be set while creating the action. To distinguish between the actions and their parameters, every action is surrounded by a thick border on the GUI. The parameters that must be set irrespective of the action are not included inside the border. Before navigating to the next page, the data entered by the tester is saved inside the respective database at the back end. Moreover, every action has an enable button associated with it, so the tester can enable the action of his/her choice.

### Design of Dialogs

The dialogs are disabled by default for the actions page as the actions are not a mandatory requirement. On enabling the action using the ‘Enable’ button, the dialogs are enabled for the tester to input the details for the selected action. Necessary information is associated with every dialog box to ensure ease to the tester while creating the test scenarios.

# Implementation

In this chapter, the architecture for our system is presented with the details of implementation. The development details like the hardware and the software requirements for the utilization of VTD AV stack are discussed. The modified database design is also presented with the help of database Entity Relationship Diagrams.

## System Architecture

### Development Environment

#### Software Requirements

In this thesis, the below mentioned software tools are used to integrate the required functionality with the web application.

1. Django V3.0 framework for web-based applications.
2. Python 3.x
3. HTML5, CSS and JavaScript.
4. Bootstrap V4.0 CSS library.
5. Vires VTD simulator installed on a Linux PC [5].

#### Hardware Requirements

There are no special hardware requirements for this development. A Linux machine with the generally used web browsers and the VTD simulator installation has been used. The VTD simulator was being installed on the Linux system during the MT-3183 [5].

### System Architecture Diagram

The architecture diagram for the system is presented in this section and the important components are described. The important components of our architecture are the Robo-Test web app, the VTD and LGSVL simulators that are connected to the web application. The Robo-Test web application is built using the Django web development framework. The web front-end is managed using HTML, CSS, Bootstrap and JavaScript whereas the back end is in Python. In the Robo-Test web application project, all the components are designed as separate Django applications. More about the Django and Django applications can be found out on the official documentation on the website and in MT-3136 [4]. The VTD connector, the VTD test editor and the database view applications are utilized and modified during this development. The components that concern this thesis are discussed below.

Graphical user interface, diagram

Description automatically generated

Figure 4.1: System Architecture

1. **VTD Test Editor**: VTD test editor application contains html forms that are rendered to the user while creating the test scenario for VTD simulator. User input is accepted, validated and stored in database successfully by the corresponding function assigned for each form. There are multiple forms that are supplied to the tester to create scenarios making this as one of the most important apps in the Robo-Test web application. In this thesis, additional form for the driver selection and setting actions on the Ego vehicle. Corresponding function to accept, validate and save the inputs to the database tables is also added in this application.
2. **VTD Connector**: VTD connector application is responsible for the communication between VTD simulator and the Robo-Test web application. It acts as an interface to the VTD simulator. It is responsible to run the simulation and allow the tester to fetch the results after the simulation. In this thesis, the functionality to generate runtime VTD data and store it into the database and display to the tester has been added.
3. **Database**: The database for the VTD simulator contains the test data entered by the tester. The database has been further extended in this thesis to record the test results of the VTD simulation run.

### Database Design

The database design is an important aspect in any software development project as it is not always flexible to change the design mid-way. In this thesis, the existing Python based SQLite database has been used and the current design for the VTD simulator database tables has been extended. In the VTD test editor application, the new database tables are added as Django models in the views.py file. The additional database tables and their integration with the existing database is discussed below with the help of Entity Relationship Diagram (ERD). An ERD contains of all the tables with attributes and relation between tables with the help of Primary and Foreign key constraints. First, the ERD for Ego vehicle Action Control is discussed followed by the ERD for VTD simulation results.

#### Entity Relationship Diagram for VTD Test Editor

A screenshot of a computer

Description automatically generated with medium confidence

Figure 4.2: ERD for VTD Test Editor

The highlighted tables are the new additions in the existing database design for the Ego vehicle Actions. The details are discussed below.

1. **AddPlayerActionsDB**: The player actions database table contains the VTD driver type details along with the activation radius and the pivot. It is linked with AddEgoDB on a one-to-one relationship basis.
2. **AddLaneChangeDB**: This table contains the parameters that are necessary to set the lane change action on the Ego vehicle. It contains data like the lane change direction, activation event, number of executions and delay time for the action after the trigger condition is met. It is connected to the AddEgoDB on a one-to-one relationship basis.
3. **AddSpeedChangeDB**: This table contains the parameters that are necessary to set the speed change action on the Ego vehicle. It contains the data like the rate of speed change, the target speed, the activation event and the number of executions. This table is connected to the AddEgoDB on a one-to-one relation.
4. **AddAutonomousDB**: The autonomous action parameters are stored in the AddAutonomousDB table. The parameters like the delay time for the action, number of executions and activation event are stored in this table. It is connected to the AddEgoDB on a one-to-one relationship basis.

#### Entity Test Relationship for VTD Connector

Graphical user interface, application

Description automatically generated with medium confidence

Figure 4.3: ERD for VTD Connector

1. **TestConfigDB**: The TestConfigDB contains the test case name, test description and the details about the creation of the test case. This database table acts like a parent entity for the other database tables.
2. **VTDSimulationResultsDB**: The VTDSimulationResultsDB is linked to the TestConfigDB on a one-to-many relationship. For a given test configuration there are more than one simulation result database objects in VTDSimulationResultsDB table. The VTD provides a large amount of runtime data relevant to the Ego vehicle, other traffic elements in the simulation and static information about the road and lane types. In this table, the most relevant data pertaining to Ego vehicle like the simulation time and frame, steering, gear, target acceleration and steering target has been extracted and stored.

## VTD Interface with RoboTest Web Application

Before discussing the software components that are used in this thesis, it is important to understand the interface of the RoboTest web application with the VTD simulator. The following figure depicts the interfaces that VTD offers to communicate with the outside components.

Diagram

Description automatically generated

Figure 4.4: VTD Interfaces [14]

The VTD offers two public interfaces for communication with external components. The interfaces are the SCP (Simulation Control Protocol) and RDB (Runtime Data Bus). The SCP generally offers event-based data transfer and is used to send control/action commands and receive the status information. With the help of SCP commands, the tester can have control over the simulation. It has XML like syntax with tags and attributes making the SCP commands human readable, customizable, and extensible [14].

RDB protocol on offers a periodic data per simulation frame. It is possible to receive object information such as vehicle positions and states, environment information, sensor information as configured in the Module Manager file, road information like lane, road marks, image data, light sources etc. The RDB data is available for exact timestamps and frames of the simulation. It is possible to trigger the VTD via RDB as well as to send the custom data using RDB interface. It is possible to record the RDB data using three options – recording of the RDB stream into the task control, recording an arbitrary data stream using RDB sniffer tool, using RDBRecorder Plugin [14]. The utilization of one of these methods to record the simulation data is discussed in the later sections in this thesis.

## Software Components

The Django apps that are built in MT-3183 [5] are enhanced in this thesis to integrate and test the VTD AV software stack using the RoboTest web application. These apps are discussed here with the details of the new functionality and the additional URL’s that are added as a part of enhancement.

### Software Component VTD Manual Test Editor

The VTD Manual Test Editor application contains the functionality to collect the scenario creation parameters from the tester, validate the entered data and store it in the respective database. The application is named as ‘vtdtesteditor’ in the Django project and contains the necessary static and template files along with the urls.py, models.py and views.py files. The urls.py file contains the list of URLs that are associated with this application. The models.py file contains the database tables described as Django model classes and the views.py file contains the functionality like validation of inputs and storing them into a database table for every html form rendered to the tester. Let’s check the new URL that has been added in this application. The existing URL details can be referred in MT-3183 [5] and hence are not discussed in this thesis.

Table 4.1: Additional URLs for ‘vtdtesteditor’ application.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| URL Suffix | Mapped Function | HTTP Method | Parameters | HTML Template name | Purpose |
| …/vtdaddEgoActions | vtdaddEgoActions | GET-POST |  | vtdaddEgoActions.html | To render the VTD Ego actions form for setting actions on Ego vehicle. |

The new database tables added as Django models in models.py file are listed below.

Table 4.2: Additional database tables for ‘vtdtesteditor’

|  |  |  |
| --- | --- | --- |
| Django model name | Purpose | Belongs to |
| VTDAddPlayerActionsDB | To store VTD driver details and necessary parameters for actions on Ego vehicle. | VTDAddEgoDB |
| VTDAddLaneChangeDB | To store the Lane Change parameters. | VTDAddEgoDB |
| VTDAddAutonomousDB | To store the Autonomous action parameters. | VTDAddEgoDB |
| VTDAddSpeedChangeDB | To store the Speed Change action parameters. | VTDAddEgoDB |

The GUI design of the form input for the Ego vehicle actions is as follows.

Graphical user interface, text, application, email, Teams

Description automatically generated

Figure 4.5: Ego Actions Common Parameters GUI

Figure 4.5 shows the Ego Actions common parameters like the selection of the driver type, the trigger type, pivot and activation radius. All the input fields are provided with the necessary information for the tester that would be useful while creating the scenarios.

Application

Description automatically generated with medium confidence

Figure 4.6 Action Autonomous GUI

Figure 4.6 shows the GUI for setting Autonomous action on the Ego vehicle. There is an ‘Enable’ checkbox that must be set by the tester for the form fields to become active. By default, the form fields are inactive and the tester cannot enter any information. This is achieved with the help of the JavaScript file that has been added to deactivate all the form inputs if the tester does not enable a particular action.

Graphical user interface, text, application

Description automatically generated

Figure 4.6: Action Lane Change GUI

Figure 4.6 shows the Lane Change action GUI for the Ego vehicle. Here, the necessary information is provided for a few inputs. By default, all the form fields are inactive and the tester has to select the ‘Enable’ checkbox to be able to activate this action and enter the inputs.

In the Figure 4.7 the Speed Change action GUI for the Ego vehicle is shown. The required form input fields are marked with an Asterix. Like the Autonomous and the Lane Change actions, by default the form input fields are disabled and the tester must enable the action before entering any information.

All the actions are available on a single form input page. The form template for this page is inherited from the existing VTD test editor template to maintain the consistency throughout the application.

Application

Description automatically generated with medium confidence

Figure 4.7: Action Speed Change GUI.

### Software Component VTD Connector

The VTD Connector component is responsible to interact with the VTD simulator using the SCP interface provided by VTD. The Django application in the RoboTest web application project is named as ‘VTDConnector’. This component is enhanced with the collection and the processing of the test results obtained from VTD during the simulation in this thesis. To collect the runtime data, VTD’s RDB interface is used in this software component. More details of the data provided by RDB interface are discussed in section 4.2.

VTD offers two tools called SCPGenerator and RDBSniffer that are used in this thesis. The SCPGenerator tool issues the SCP commands generated by this component to the VTD simulator for the simulation to execute. The RDBSniffer tool allows to extract the required runtime data during the simulation with the help of commands. There is a set of RDB commands with configuration options offered by the VTD to collect the data as per the user requirement as shown in the following figure.

Text, letter

Description automatically generated

Figure 4.5: RDBSniffer Usage

The configurable options for the usage of the RDBSniffer tool are shown in the above figure. One of the options provided by this tool to store the runtime data is to gather it in a CSV file and then process it later as per the requirements. In this thesis, this approach to gather the runtime data during the simulation is used.

The data must be recorded and written to the CSV file while the simulation is running. For this it is required that the simulation and the data collection run parallel. In this software component, an additional thread is integrated that operates concurrently with the thread that is responsible to run the simulation and collects the data. Later, this data is read from the CSV file using the python’s Pandas and the required data like the steering, acceleration, gear is extracted and stored in the VTDSimulationResultsDB database model associated with this component.

No new URLs are added in this software component. All the enhancement is done in the views.py and the models.py files. The table below shows the database additions for this software component.

Table 4.3: Additional tables for ‘VTDConnector’

|  |  |  |
| --- | --- | --- |
| Django model name | Purpose | Belongs to |
| VTDSimulationResultsDB | To store the VTD simulation runtime data collected using the RDBSniffer tool. | VTDTestConfigDB |

The following table summarizes the new functions added in the views.py file.

Table 4.4: Additional functions for ‘VTDConnector’

|  |  |  |  |
| --- | --- | --- | --- |
| Function Name | Purpose | Parameters | Returns |
| generate\_rdbData() | Runs the command to use the RDBSniffer tool with the required packages and extract the runtime data for the simulation. This function is called inside a thread in vtdrun\_test function. | None. | No return value. |
| logic\_skiprows(index) | This is a helper function to skip a few rows while reading the CSV file that contains the runtime data. It is called inside process\_rdbData function. | Index (Integer) | Boolean. |
| process\_rdbData(request) | This function is responsible for reading, processing and saving the simulation results from a CSV file into the Django model in RoboTest web application. It is called when the ‘Save Results’ button is clicked on the UI. | HTTP Request |  |
| vtdShowSimResults(request, id) | Responsible for displaying the results of the simulation for the selected test case to the tester. It fetches multiple database objects and shows the results. It is called when the ‘Show Results’ button is clicked on the GUI. | HTTP Request, id (Integer) | Success: vtdSimResults.html. |

# Simulation Results

# Traceability and Test Automation

The traceability between the various artifacts right from the requirements to the test results in a software life cycle is important as it reduces the time and cost in case of bug fixes. This section discusses the aspect of traceability in the RoboTest system at a high level followed by a literature review on the methods of automated test case generation from the requirements. The architecture presented below for high-level cognitive testing of Autonomous Systems was developed in MT-3253 [21] and is discussed briefly to highlight the area of focus with regards to traceability in this thesis.

A picture containing diagram

Description automatically generated

Figure 6.1: Robo-Test Cognitive Testing Architecture [21].

Figure 6.1 shows the high-level Cognitive testing architecture for the Robo-Test system. The different artifacts of the system such as the Requirements engine, Data Generator, Test Environment and management of test cases and training data are depicted with their relationships amongst each other [21]. The Requirements Engine is responsible for the management of requirements and traceability. IBM DOORS Rational software has been used as a Requirements Engine in our system. The test environment for our system is the RoboTest web application which is connected to the LGSVL and the VTD simulators with their corresponding AV software stacks. The RoboTest web application also serves as a tool for creation of test scenarios and management of test results. The scenario indexing tool interfaced with the RoboTest web application acts as a data generator as it selects the corner test cases using reinforcement learning methods. The traceability between these system components is important and, in our system, there are two artifacts to link together – IBM DOORS Rational Software and RoboTest web application. As highlighted, this thesis focuses on the traceability options between the requirements from IBM DOORS and the test results in the RoboTest web application.

## Traceability between Requirements and Test Results

The focus of this part is to create traceability between the requirements, test cases and the test results in our system. IBM DOORS software is the requirements engine, and the RoboTest Web application is the test execution/simulation engine in our system. The idea is to have a traceability between the requirements engine and the test engine by being able to link the test cases/test scenarios with the test results. We will discuss the possible approaches at a high level to achieve the linkage between the requirements and the test results generated by the web application. The table below summarizes the literature survey and its relevance to our system with regards to traceability.

Table 6.1: Literature Review for traceability methods

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Article | Traceability between Requirements and Test Cases | Method/Tool | Traceability between Requirements and Test Results | Method/Tool | Relevance to our system |
| Linking functional requirements and software verification.[15] | Not with test cases, but traceability between requirements and implementation. | CBMC, a software bounded model checker for C. | NO. | NA. | LOW. |
| Test driven requirements engineering.[16] | YES. | NA. | NO. | NA. | MODERATE. |
| OSLC Tools Integration and Systems Engineering – The relationship between the two worlds.[17] | YES. | NA. | YES. | NA. | HIGH. |
| Integration of Requirements Engineering and Test Case generation via OSLC.[18] | YES. | Model based tools connected using OSLC. | YES. | Model based tools connected using OSLC. | HIGH. |
| An integrated framework for traceability and impact analysis in requirements verification of cyber-physical systems.[19] | YES. | A separate server hosted using Node js, python. | YES. | A separate server hosted using Node js, python. | HIGH. |
| Verification challenges for Autonomous Systems.[20] | NO. | NA. | NO. | NA. | LOW. |

H. Post, C. Sinz, F. Merz, T. Gorges and T. Kropf [15] base their research on linking the functional requirements and the formal verification methods. Requirements formalization together with formal verification leads to discovery of implementation problems missed in traditional testing [15]. C. Ebert and R. Ray [16] discuss test driven requirements engineering and the fact that requirements can be directly applied as test cases. This leads to reduction in lead time and the costs in testing are reduced by about 30% [16].

M. Saadatmand and A. Bucaioni [17] discuss the Open Services Lifecycle Collaboration based integration of tools used in different phases of software lifecycle. Relationships between different artifacts are established from the beginning of the software engineering starting from the requirements [17]. B. K. Aichernig et al. [18] addresses the traceability between requirements, test cases and test results using a requirement centered analysis and testing framework that uses model-based tools connected using the OSLC standard [18]. A. Mengist, L. Buffoni, and A. Pop [19] present an approach and a prototype for automatically generating and maintaining the traceability between requirements model, design model, down to simulation and verification results throughout the product lifecycle in model-based design of CPS. The approach consists of hosting a server using technologies like Python or Node.js and connect to an RDF based database that contains all the artifacts linked using a URI [19]. Redfield S.A. and Seto M.L [20] discuss the verification challenges for autonomous systems and identify existing tools and the gaps between them. These challenges are presented at an abstract level in this article [20].

The method and approach discussed in the above literature must be evaluated with respect to our existing system that uses the IBM Rational DOORS software as a requirements engine and RoboTest Web application with the VTD and LGSVL simulators as a simulation/test engine.

The approaches that could be possible for our system are discussed and the following table summarizes and evaluates them based on their complexity with regards to our system.

Table 6.2: Possible Traceability approaches

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Approach | Literature | Software Requirements | Implementation Effort | Complexity | Challenges |
| Use the DOORS database and traceability for requirements and test cases. Store the results of test execution in the RoboTestWebApp database. Create external links for every test case in DOORS database and link it with the test result in the RoboTestWebApp database. The link in DOORS should directly open the test result in the browser on clicking the URL. | Uses only the URL concept from [19]. | IBM DOORS, RoboTestWeb Application. | MODERATE. Linking effort required in DOORS. | Moderate. | Creation of links in DOORS for every test case. Fetching the test results in a browser page on clicking the URL in DOORS. |
| Host a separate server using Node.js or Python Django that connects to an RDF based database backend. Export the requirements, test cases data using a URI to the server. At the RoboTestWebApp end, generate the same URI for the test execution corresponding to the test case. Export the results using the same URI to the server. On fetching the specific URI, all the linked requirements, test cases and test results data should be available. | Utilizes the OSLC, server and URI concepts from [17], [18] and [19]. | IBM DOORS, RoboTestWeb Application, New server (Node.js / Python Django). | Very High. | Highly Complex. | Creation of same URI in RoboTestWeb App for every test execution corresponding to a test case. Exporting results from DOORS and RoboTestWebApp to the new server. |

Table 6.2 summarizes the possible approaches that could be implemented going further in our system to ensure the traceability. The first approach in the table does not need a complex architecture and is moderately easy to understand and implement. The following architecture is a high-level overview of the second approach discussed in the above table. The second approach in figure 6.2 consists of a server hosted in Python/Node.js which interacts with an RDF (Resource Description Framework) based database engine. The server provides its own user interface to fetch the traceability results for any desired URI. The requirements from DOORS are posted into the hosted server using HTTP request. Similarly, the results are posted using HTTP into the server. The requirements and the test results are linked using the same URI and stored into an RDF based database managed by the server. In this way, every individual requirement and its corresponding test results are linked using one URI and are available to the tester using the web interface of the hosted server.

Diagram

Description automatically generated

Figure 6.2: Server and RDF concept for traceability.

The approach presented above is very complex and its implementation requires sound knowledge of RDF based databases and client server architectures. Moreover, the challenge lies in the URI generation methods and linking the URI generated for test results with the corresponding requirement in the DOORS database.

## Automated Test Case Generation

Automatic generation of test cases for the system from requirements is one of the research areas in software life-cycle automation. There is a lot of research being done already in automated test case generation. The research is based on different techniques like the model-based methods, semantic mining of text and NLP based approaches. Going forward, the objective of the RoboTest system is to have automated test case generation from the text-based requirements. In this thesis, a literature review is done on the existing methods for the automatic generation of test cases from requirements using different methods. The following table summarizes a few articles that discuss these techniques in detail.

# Bibliography

|  |  |
| --- | --- |
| [1]  [2]  [3]  [4]  [5]  [6]  [7]  [8]  [9]  [10]  [11]  [12]  [13]  [14]  [15]  [16]  [17]  [18]  [19]  [20]  [21] | C. Ebert and M. Weyrich, "Validation of Autonomous Systems," in IEEE Software, vol. 36, no. 5, pp. 15-23, Sept.-Oct. 2019, doi: 10.1109/MS.2019.2921037.  N. Kalra and S. M. Paddock, “Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability?,” *Transportation Research Part A: Policy and Practice*, vol. 94, pp. 182–193, 2016, doi: <https://doi.org/10.1016/j.tra.2016.09.010>.  “Apollo.” <https://apollo.auto/devcenter/devcenter.html> (accessed Nov. 27, 2021).  J. John, “Development of a tool for the collection, prioritization and application of test cases,” Institute of Industrial Automation and Software Engineering, University of Stuttgart, Stuttgart, 2020.  Lili Ji, “Modernization of a Simulator for the Improvement of Autonomous Driving Simulation Environment”, Institute of Industrial Automation and Software Engineering, University of Stuttgart, 2021.  Trieu Tach Vu, “Modernizing the autonomous vehicle software stack”, Institute of Industrial Automation and Software Engineering, University of Stuttgart, 2021.  *ApolloAuto/apollo*. Apollo Auto, 2021. Accessed: Nov. 30, 2021. [Online]. Available: <https://github.com/ApolloAuto/apollo>  “Apollo.” <https://apollo.auto/cyber.html> (accessed Nov. 30, 2021).  “Supported message types - LGSVL Simulator.” <https://www.svlsimulator.com/docs/archive/2020.06/simulator-messages/> (accessed Dec. 01, 2021).  V. Mazzari, “ROS vs ROS2,” *Génération Robots - Blog*, Dec. 17, 2019. <https://www.generationrobots.com/blog/en/ros-vs-ros2/> (accessed Dec. 02, 2021).  “Traffic and Scenario - Virtual Test Drive - VIRES Simulationstechnologie GmbH.” <https://redmine.vires.com/projects/vtd/wiki/Traffic_and_Scenario#Driver-Behavior> (accessed Dec. 04, 2021).  “Traffic and Scenario - Virtual Test Drive - VIRES Simulationstechnologie GmbH.” <https://redmine.vires.com/projects/vtd/wiki/Traffic_and_Scenario> (accessed Dec. 04, 2021).  M. Dupuis, “OpenDRIVE Scenario Editor - User Manual,” p. 40.  “Daten aus VTD - Virtual Test Drive - VIRES Simulationstechnologie GmbH.” <https://redmine.vires.com/projects/vtd/wiki/Daten_aus_VTD#RDB-Data-Recording> (accessed Dec. 09, 2021).  H. Post, C. Sinz, F. Merz, T. Gorges and T. Kropf, "Linking Functional Requirements and Software Verification," 2009 17th IEEE International Requirements Engineering Conference, 2009, pp. 295-302, doi: 10.1109/RE.2009.43.  C. Ebert and R. Ray, "Test-Driven Requirements Engineering," in IEEE Software, vol. 38, no. 1, pp. 16-24, Jan.-Feb. 2021, doi: 10.1109/MS.2020.3029811.  M. Saadatmand and A. Bucaioni, "OSLC Tool Integration and Systems Engineering -- The Relationship between the Two Worlds," 2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications, 2014, pp. 93-101, doi: 10.1109/SEAA.2014.64.  B. K. Aichernig et al., "Integration of Requirements Engineering and Test-Case Generation via OSLC," 2014 14th International Conference on Quality Software, 2014, pp. 117-126, doi: 10.1109/QSIC.2014.13.  A. Mengist, L. Buffoni, and A. Pop, “An Integrated Framework for Traceability and Impact Analysis in Requirements Verification of Cyber–Physical Systems,” *Electronics*, vol. 10, no. 8, p. 983, Apr. 2021.  Redfield S.A., Seto M.L. (2017) Verification Challenges for Autonomous Systems. In: Lawless W., Mittu R., Sofge D., Russell S. (eds) Autonomy and Artificial Intelligence: A Threat or Savior? Springer, Cham. <https://doi.org/10.1007/978-3-319-59719-5_5>.  Divith Bajaj, “Development of Flexible Software Architecture for Cognitive Testing of Autonomous Systems”, Institute of Industrial Automation and Software Engineering, University of Stuttgart, 2021. |

# Declaration of Compliance Style Template: Überschrift 1 / Headline 1 (delete the number)

I hereby declare to have written this work independently and to have respected in its preparation the relevant provisions, in particular those corresponding to the copyright protection of external materials. Whenever external materials (such as images, drawings, text passages) are used in this work, I declare that these materials are referenced accordingly (e.g. quote, source) and, whenever necessary, consent from the author to use such materials in my work has been obtained.

Signature:

Stuttgart, on the <DD.MM.YYYY>