Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

A Simulator for Connected Car Scenarios

Bedirhan Keskin

Course of Study:	Information Technology
Examiner:	Prof. Dr. -Ing. habil. Bernhard Mitschang
Supervisor:	Dr. rer. nat. Pascal Hirmer
Commenced:	October 15, 2021
Completed:	April 15, 2022

Abstract

Internet of Things (IoT) is utilized in various industries to deal with problems and provide new features to customers. In addition, there is a trend that most devices are evolving into entities that have communication capacities. Mobility is a prominent aspect of our lives and industry, universities and companies in the automotive field keep in step with the trend. Internet of Vehicles (IoV) can be basically defined as a network of smart cars that are connected to other vehicles and infrastructure [STM18]. In this concept, vehicles gather various data internally and from the environment, refine the data and share it. Besides, data transmission among vehicles is a fundamental component of autonomous driving.

Data flow steps of the Internet of Vehicles should be designed to create this kind of complex system. A digital twin is a substantial segment of the system. It is a digital replica of a thing or system that regularly communicates with the physical world and manipulates the actions of the physical part. The data loop specifies the data flow steps which are obtaining data, processing, and forwarding it to a digital twin and then back to the vehicle. The development of such an immense system, which has cyber and physical parts, on real vehicles is costly and arduous regarding testing.

This master thesis aims to conceptualize and implement a simulator that provides a data generator for connected car scenarios and imitates the data loop. The generation of synthetic vehicle data is another objective of the thesis. A connected car simulator grounds existing vehicle simulators and technologies. It extends a suited simulation tool (MetaDrive) for creating Internet of Vehicle scenarios.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

2D 2 Dimension. 21

3D 3 Dimension. 20

ABS Anti-lock Braking System. 27

ADAS Advanced Driver-Assistance Systems. 28

AI Artificial intelligence. 32

API Application Programming Interface. 21

CSV Comma-separated values. 81

EGR Exhaust Gas Recirculation. 26

FOV Field of View. 24

GPS Global Positioning System. 21

GPU Graphics processing unit. 32

GUI Graphical User Interface. 16

IAT Intake Air Temperature. 25

IoT Internet of Things. 3

IoV Internet of Vehicles. 3

JSON JavaScript Object Notation. 11

KPa kilopascal. 25

Lidar Light Detection and Ranging. 20

m meter. 49

MAP Manifold Absolute Pressure. 25

MQTT MQ Telemetry Transport. 45

N Newton. 49

NHTSA National Highway Traffic Safety Administration. 17

Radar Radio Detection And Ranging. 21

RL Reinforcement Learning. 31

RPM Round per Minute. 25

SUMO	Simulation of Urban Mobility. 21
TCP	Transmission Control Protocol. 69
UDP	User Datagram Protocol. 67
UML	Unified Modelling Language. 39
V2I	Vehicle to Infrastructure. 18
V2P	Vehicle to Personal Devices/Pedestrian. 18
V2R	Vehicle to Roadside Units. 17
V2S	Vehicle to Sensors. 18
V2V	Vehicle to Vehicle. 17
V2X	Vehicle to Everything. 18
VANETs	Vehicular Ad hoc Networks. 17
XML	Extensible Markup Language. 11

1 Introduction

Internet of Things is one of the integral technology in our lives and customers demand more connectivity in miscellaneous areas of their lives. 11.3 billion IoT devices are in commission presently and studies show that the number will escalate up to 27.1 billion in 2025 even though the COVID-19 pandemic lowered the expectations due to chip shortage and transportation issues [IoT21]. On the other hand, demands for car ownership started to increase again in 2021 after they dropped in 2020 because of the pandemic [Sta21]. Overall, there is obvious appeal in both domains. Implementation of IoT in vehicles is specified as the IoV.

Internet of Vehicles ensures extra safety and entertainment features to car owners via obtaining data from a car, a driver, and an environment. The data is utilized for monitoring, car and traffic management, localization, emergency services, diminishing pollution, and other application areas. The key factor of IoV is how to treat data in terms of gathering, processing, and transmitting it. The digital twin contains the same key factor. Digital Twin is a digital mirror of the real-world system that retrieves data from the system, simulates it, and governs the system. The system's current, forthcoming status and behaviors are forecasted to enlarge decision-making. The path which is followed by data is called the "data loop". The path starts from the data source and then goes to the digital twin and processed data returns to the source. For instance the tire slip data is obtained from a sensor and this data is transmitted to distributed or central server to be processed. The server transfer data into a information and send the information to the ego vehicle and other vehicles to inform them about road condition.

Nevertheless, the development of a system that consists of a data loop, multiple vehicles, varying environmental factors, pedestrians is sophisticated. Besides, verification and validation of the system are not simply applicable in the world because of safety and costal reasons. The development steps of such a system shall be tested in an unreal environment first. This engenders a need for a simulator environment for connected cars.

The goal of the thesis is the conceptualization and development of a simulator for connected car scenarios. Generation of data to realize the data loop shall be maintained. Generating of synthetic data as realistic as possible is also one of the targets of the thesis. Thus, existing data generation tools and simulators are researched to be employed in the connected car simulator. The main focuses of the thesis are:

- Requirement analysis for a connected car simulator. Since there is no specific simulator for connected cars regarding our research, the simulator shall be designed and conceptualized.
- Development of a data generator which runs concurrently with the simulator or is embedded into the simulator. Data related to the vehicle, driver, and environment is the foundation of the data loop. Hence, it is imperative to achieve realistic data generation. Besides, generated

data is processed and then used to control vehicles in the simulator for various scenarios. The simulator shall have an interface which provides data to other applications, such as machine learning systems, databases, and dashboards.

- Development of a connected car simulator. The simulator allows to control and acquire data from multiple cars simultaneously. It is required to have an extendable and flexible structure to support adding new features.
- Realization of a scenario creator that users can use to design and configure scenarios. The objective is to create a user-friendly Graphical User Interface (GUI). Therefore, the user can construct specific scenarios regarding test requirements. Users set up maps, vehicles, and environmental variables.
- A demo scenario shall be created to test and evaluate the simulator.

The structure for the thesis and basic definition of chapters is defined below:

- Chapter 2 provides the essential theory to comprehend this thesis concerning digital twin, internet of vehicles, vehicle simulators, and vehicle electronics and sensors.
- Chapter 3 examines the related work for this thesis and existing simulators.
- Chapter 4 describes the system architecture and concept of how thesis requirements are fulfilled such as data storage, message transmission between tools, approach for scenario creation. Besides, it concludes fundamental design points. It is required to have an extendable and flexible structure to support adding new features.
- Chapter 5 discusses which decisions were made regarding realizing the concept. In addition, it shows implementation details and how components of the architecture work.
- Chapter 6 shows a scenario creation and evaluates the output of the thesis.
- Chapter 7 explains the outcome of the thesis and future improvement points.

2 Theory and Fundamentals

In this chapter, the fundamentals to develop a connected car simulator are discussed. This chapter sums up the theory around connected cars and simulators. Based on this, necessary features for the simulator can be specified in Chapter 4 Simulation System and Concept.

2.1 Internet of Vehicles

It is helpful to discuss the IoV and Vehicular Ad hoc Networks (VANETs) before diving into the internet of vehicles.

IoV is a revolutionary innovation that bridges the gap between the digital and physical worlds. By enabling connectivity between items and humans, the IoVs has contributed to the birth of a wiser and more intelligent world. Internet of vehicles refers to the application of IoT in cars. It can be thought of as a platform with data gathering, processing capabilities, as well as a wireless network gateway that transmits and receives data.

It is hard to provide an exact number, however, according to researches, the total number of cars in use globally, whether commercial or passenger, is slightly more than one billion [Sta17]. Besides, it is expected to reach over two billion by 2035 [JOH11]. The number of intelligent connected cars is estimated to hit 381 million by 2025 [HAF21]. The connection between cars and intelligent transportation systems for safety is not a new idea. VANETs were introduced in 2001. It focuses on safety and efficiency and has features to improve these domains using wireless connectivity [BSB18]. VANETs supply essential connections that the traditional VANETs have transformed into the internet of vehicles thanks to the momentum towards the IoV applications. VANETs have hurdles regarding unstable internet access, incompatibility with personal devices, lack of commercialization, restricted processing capabilities, etc. Basically, IoV is a hybrid of VANETs and IoT [SK21] which has advantages with respect to incorporate smartness in vehicles.

Internet of vehicles consists of three fundamental connection components which are intra vehicular communication, inter-vehicular communication, and vehicular mobile internet [GM19]. Communication of IoV can be divided into five parts [SK21]:

- Vehicle to Vehicle (V2V): It refers to the ability of connected cars to interact wirelessly with one another which are in the same vicinity. It decreases the impact of an accident or prevents it completely, therefore, it eliminates risks and increases road traffic efficiency. V2V communication has the ability to address about 80% of multi-vehicle collisions according to the National Highway Traffic Safety Administration (NHTSA) [Nat10].
- Vehicle to Roadside Units (V2R): Roadside unit is a roadside wireless communication device that offers networking and information assistance to passing cars, such as safety alerts and traffic updates

- Vehicle to Personal Devices/Pedestrian (V2P): It refers to data transmission between a car and a person or a group of pedestrians in the near vicinity. Information can also be directed towards other vulnerable receivers including cyclists.
- Vehicle to Sensors (V2S): This inter vehicular communication component defines the connection to a car's sensors that perceive car status and environment. Unlike others, this part can be a wired link.
- Vehicle to Infrastructure (V2I): It describes wireless information transactions between a car and road infrastructure, such as traffic lights. It is crucial in autonomous driving in long term. It is beneficial to diminish the impact of accidents.

The above classification is not standard and other academic papers have a different number of subcategories. Vehicle to the internet, vehicle to cloud, Vehicle to Everything (V2X) are some of the subcategories that are mentioned in other classifications. The contrast of VANETs from IoV is that VANETs consist of just V2I and V2V.

IoV's architecture is organized into four levels as shown in Figure 2.1 regarding the paper of Yang et al. [YLLW17]:

1. Environmental sensing and control layer: This layer is the source of data that is perceived from the car itself and the environment using sensors.
2. Network access and transport layer: The key functions of this layer inside IoV are node management, surveillance system, data analysis, and data processing. It provides a connection to vehicles with distinctive techniques.
3. Coordinative computing layer: It is responsible for IoV organization by facilitating the interaction of cognitive computing capability or swarm intelligent coordinative computing capability. This layer also controls data handling and resource management.
4. Application layer: Open and closed services are two types of service. Open services assist overall business strategy. On the other hand, closed services are tailored to certain applications such as control platforms and traffic management.

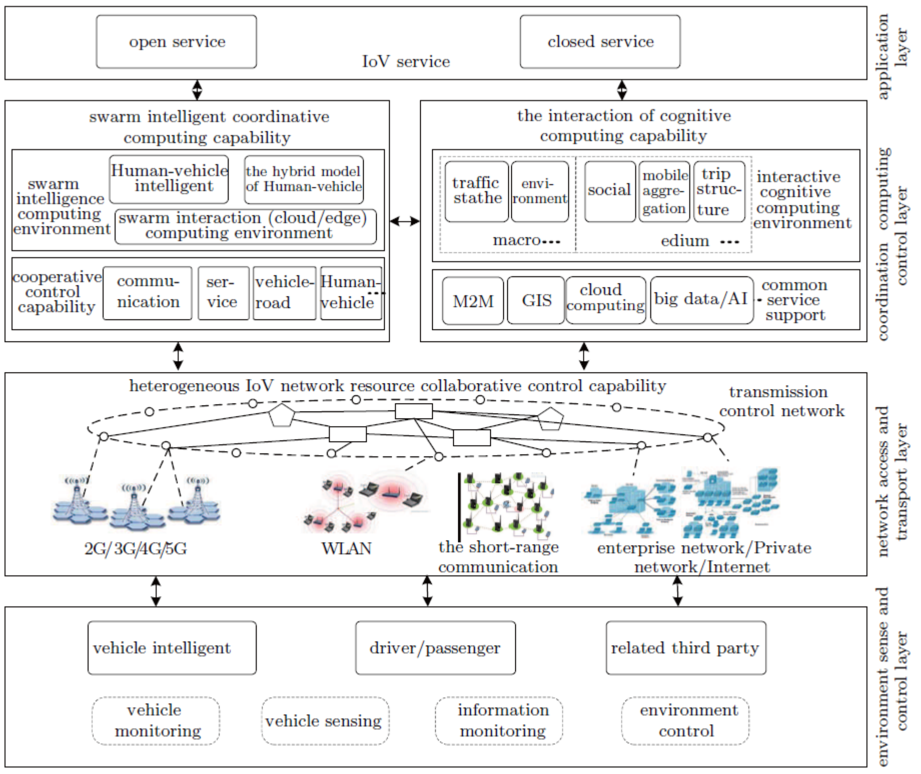

open service
IoV service
closed service
application layer
swarm intelligent coordinative computing capability
swarm intelligence computing environment
Human-vehicle intelligent
the hybrid model of Human-vehicle
swarm interaction (cloud/edge) computing environment
cooperative control capability
commu- nication
ser- vice
vehicle- road
Human- vehicle
the interaction of cognitive computing capability
traffic stathe
envi- ronment
social
mobile aggre- gation
trip struc- ture
macro···
edium ···
interactive cognitive computing environment
M2M
GIS
cloud computing
big data/AI
common service support
coordination computing control layer
heterogeneous IoV network resource collaborative control capability
transmission control network
2G/3G/4G/5G
WLAN
the short-range communication
enterprise network/Private network/Internet
network access and transport layer
vehicle intelligent
driver/passenger
related third party
vehicle monitoring
vehicle sensing
information monitoring
environment control
environment sense and control layer

Figure 2.1: Architecture of IoV [YLLW17]

Some prominent implementations of IoV are emergency vehicle warnings, slow-moving vehicle notice, collision risk notice, roadside assistance, and SOS service. Besides many advantages, IoV has issues and challenges. One of the major challenges is big data because the huge volume of data is created by a great number of vehicles and requires to be processed. Security and privacy are some of the main concerns of the customers and any flaw in it can cause severe damage. Due to the rapid mobility of cars, data transmission in real-time becomes challenging. Lastly having no standard in IoV, communications such as V2I and V2V turn into arduous work.

2.2 Digital Twin

Since the connected car simulator imitates a data loop which also has similar requirements to the digital twin. It is required to understand digital twin basics.

Digital Twin is at the heart of the Industry 4.0 revolution, which is made possible by powerful data analytics and IoT connection. The establishment of a linked physical and digital twin helps to address the difficulty of interconnectivity between IoT and data analytics. Ever since the early 2000s, specific concepts about Digital Twins have existed [Gri14]. The National Aeronautical Space Administration (NASA) defines the digital twin as *"[. . .] an integrated multiphysics, multiscale, probabilistic simulation of an as-built vehicle or system that uses the best available physical models, sensor updates, fleet history, etc., to mirror the life of its corresponding flying twin"* [GS12]. In the digital twin, data is transmitted between an actual physical thing and a digital entity, and also they are tightly integrated. This means any change in either a physical thing or a digital entity affects one another [FFD20].

Like any other technology, it has its own challenges. Security and privacy is the first challenge for such a system that controls devices in real-time. Handling data is another compelling topic. Lastly, trust and IT infrastructure are remaining challenges of the Digital Twin [FFD20].

2.3 Vehicle Simulators

The simulator platform is the most fundamental element of the thesis. Britannica Dictionary[Bri21] defines computer simulators: *"The use of a computer to represent the dynamic responses of one system by the behavior of another system modeled after it. A simulation uses a mathematical description, or model, of a real system in the form of a computer program."* Simulation software is now an integral part of engineering and many other disciplines.

Simulators can be categorized into different types. There are various commercial and open-source simulators available on the market. They can be classified as Robotics Simulators and Autonomous Vehicle Simulators regarding the goal of this thesis.

Autonomous Vehicle Simulators: Autonomous vehicle simulators focus on creating a realistic outdoor environment and providing physical properties of cars, driving models, and sensors, such as cameras, Light Detection and Ranging (Lidar) [Nik21]. They are mostly developed upon a game engine that is utilized for rendering. These kinds of simulators have one certain task, and they are built around it. The task is the training and testing of an autonomous vehicle. They have limited controls over actors which are managing vehicles and pedestrians and mostly weather. This limited flexibility is a disadvantage when other actors are wanted to be put into play. Nevertheless, research and development around driving and vehicle can be done with less effort thanks to already created models and features of the simulator. CARLA, LGSVL, and CarSim are some examples of this type.

Robotic Simulators: There are various sensors and hardware on other robotics applications, such as robotic arms, humanoid robots, and underwater robots which are not covered by Autonomous Vehicle Simulators. On this point, robotic simulators, which have Robot Operating Systems, are widely used. This type of simulator has more flexibility, but a developer must design and create many details like environments, buildings, the shape of actors, and so on. Some of the robotics simulators are highly mature and provide plugins that developers can use modularly to establish environments and actors. One disadvantage of the robotics simulator is providing less realistic 3 Dimension (3D) scenes. It is also possible to connect hardware parts to a simulator to test software. Gazebo is a very popular robotic simulator. Furthermore, because autonomous vehicle research is a part of robotics, common solutions can be considered to be used as a foundation for autonomous vehicle simulation [PR12].

2.3.1 Basic Components of Vehicle Simulators

Simulators are constructed over interconnected parts. Patel [Pat20] divided them into six components as below:

- **Game Engine (Rendering Engine):** It provides 2 Dimension (2D) or 3D graphics which is called rendering. It also has various features. However, they may not be robust enough to represent the real world e.g., physical laws, localization, artificial intelligence. The combination of a game engine with other engines sustains high fidelity simulators. Unreal Engine, Blender, and Unity 3D are popular game engines on the market. Unreal Engine and Unity 3D are free for non-commercial purposes.
- **Physics Engine:** This software yields realistic simulation to delineate physical systems of the environment and items. It focuses on models of rigid, soft, and fluid dynamics.
- **Environment Model:** It manages conditions for weather and lightning because synthetic data are generated based on these conditions.
- **Vehicle Model:** Advanced vehicle model is crucial for autonomous driving simulators because they have to support lifelike actions regarding inputs, such as a change in steering wheel and braking.
- **Sensor Model:** It is a vital component of modern driving simulators that autonomous vehicles can perceive an environment and car status. Most modern driving simulators provide cameras, Radio Detection And Ranging (Radar), Lidar, and Global Positioning System (GPS). Nevertheless, they are not sufficient for a connected car simulator. Thus, the modularity and easy extendibility of this component is required.
- **Application Programming Interface (API) layer:** A user can control the vehicles and environmental conditions using this component. Most of the existing simulators use client-server architecture.

In addition to Patel [Pat20] structure for car simulators, Yang et al. [YXM+21] splits a car simulator into seven parts: static environment simulation, dynamic environment and behavior simulation, traffic flow simulation, sensor simulation, and vehicle dynamics simulation.

The car simulators are classified into two groups by Yang et al. [YXM+21] which are simulation platforms based on point maps and simulation platforms based on 3D engine:

Simulator Based on Point Maps: This type of simulator mostly utilizes real data to create abstract scenes in 2D or 3D e.g., Carcraft (Waymo) Autoware, and Simulation of Urban Mobility (SUMO).

Simulator Based on 3D Engine: This type of simulator utilizes 3D engines to build scenes. Unity Engine and Unreal Engine are popular 3D engines that both are used in games, movies, and simulators. CARLA, Airsim, and LGSVL are examples of this type of simulator.

2.3.2 Features of vehicle simulators

Simulators for autonomous driving mostly focus on similar features for analogous aims of the thesis. They are good at generating synthetic sensor data, which is required for perception and creating 3D scenes. However, a simulator for connected cars needs further sensor data. The reality gap shall be as low as possible to test developed algorithms for connected cars. Thus, features of autonomous driving simulators and extra features for a connected car simulator are combined and listed below:

- **Realistic Graphics:** It is a critical feature of simulators for autonomous vehicles because most of the data for perception depends on realistic graphics. In addition, realistic graphics and visuals of vehicles are required for sensors, such as Lidar, camera, and Radar.
- **Controlling Environment:** The environment has a huge impact on vehicle dynamics as well as generated data. It comprises natural conditions, such as weather, road friction (wet, dry), sun position.
- **Controlling Actors:** An actor indicates other vehicles, such as cars, bicycles, and pedestrians. Scenario creation is only feasible with controlling them.
- **Controlling Multiple Car:** It is an ability that simulators let users control multiple vehicles at a time. This feature is remarkably practical when cars are connected, and various scenarios can be tested.
- **Map Creation:** This capability allows users to generate diverse test scenarios and data depending on the environment. Some simulation platforms have handy approaches to editing existing maps and creating new ones.
- **Reproducibility [AKGT20]:** Repeating the same condition and scenario is not provided by every simulator. Still, the diagnostic of a tested algorithm in a simulator is possible and uncomplicated when conditions are repeated.
- **Scenario Creation [KTTS21]:** The capacity to create diverse traffic scenarios influences whether or not a simulator is useful for the thesis. Researchers can test their software at the limits with this feature by creating scenarios that are dangerous and costly to realize in the real world. In addition, this allows researchers to analyze different types of algorithm behaviors in the same situation. A simulator shall include a versatile API that allows users to handle several parts of a simulator. For instance, pedestrian behavior, car collisions, distinct sensor models, stop signs, and so on.
- **Vehicle Dynamics [YXM+21]:** Simulating the dynamics of a vehicle is essential to replicating real-world specifications and body characteristics, such as the center of mass, inertial characteristics, motion control, suspension, acceleration, braking, and steering. Simulators in the commercial market are well developed to parameterize car components, such as tire model, steering system, braking system, aerodynamic model, and transmission system. When a simulator generates realistic data in these components, a researcher can enhance the number of sensors by using these data.
- **Traffic Flow [YXM+21]:** Autonomous cars interact with other vehicles in the simulated test environment. As a result, modeling the movement of nearby cars is prominent for generating a realistic traffic situation. The most challenging and volatile aspect of the dynamic environment is traffic flow. The traffic flow feature of the simulator manages the behaviors of cars, pedestrians, traffic lights, and other components to provide a continuous testing environment for autonomous cars.
- **Sensors:** Sensors are devices that connect the autonomous vehicle to the outside world, allowing it to understand its surroundings and detect and categorize barriers, as well as forecast the vehicle's speed. Some simulators have built-in sensors that simplify researchers' tasks.

- **Sensor Creation Support:** Since there is a considerable focus on autonomous driving, most of the simulators provide sensors regarding it. However, this thesis focuses on a simulator for connected cars which means creating synthetic data for various sensors is required. Some of the simulators have support and tutorials to add new sensors to existing simulations flawlessly.
- **Data Recorder:** Users want to evaluate the test scenarios by examining data multiple times. Furthermore, recording test data is beneficial. This feature can be provided by a simulator itself.
- **Interfaces to other software:** Some software provide effective tools about car simulation and car dynamics or make model creation easy for the simulation such as MATLAB/Simulink. Combining a simulator with this kind of software is useful and time-saving.
- **Level of Maturity:** There is an increasing trend in research about simulators in the last decades. For this reason, there are many simulators on the market and some of them are new. It is logical to consider that chosen simulator was validated and has sufficient documentation and tutorial for researchers.
- **License:** Simulator must have an open license due to the limitation of this thesis. Commercial simulators cannot be utilized in this thesis, but they are studied to understand their strengths and distinct features.

2.4 Vehicle Electronics and Sensors

A sensor is a piece of equipment that transforms energy from a measurement variable's form to an electrical output which can be analog or digital [Rib17]. Vehicle electronic systems can be defined regarding three main tasks: Control application, measurement application, and communication application [Rib17]. Block diagrams in Figure 2.2 shows the main steps of these tasks: A sensor model can be defined as a simplified representation of the sensing process. It explains what information a sensor can offer, how it is affected, and the restrictions of a sensor. It gives researchers the ability to evaluate a sensor in order to create approaches to the main system thanks to the information that comes from the model [Dur90].

There are two main approaches to building a sensor model regarding Tom Harbid [Tom20] paper. The first of them is Explicit Modelling which means sensor attributes are clearly understood to represent a sensor via source code. The second of them is the Data-Driven Approach, a sensor model is derived with the help of stored measurement data of real sensors. In this type of model, machine learning and deep learning algorithms can be utilized to generate synthetic data. However, it requires a huge dataset, and it is a black box. Hence, it is impossible to interpret the formulation of the model.

Sensor models can be categorized regarding their accuracy and explicit access to the formulation of model [SMS+20]:

- Ideal Model: It provides seamless synthetic data.
- Stochastic Model: It takes distortions and ambiguity into consideration to create data.
- Physical Model: Physical and mathematical formulations describe the sensor model. It is the most complicated model.

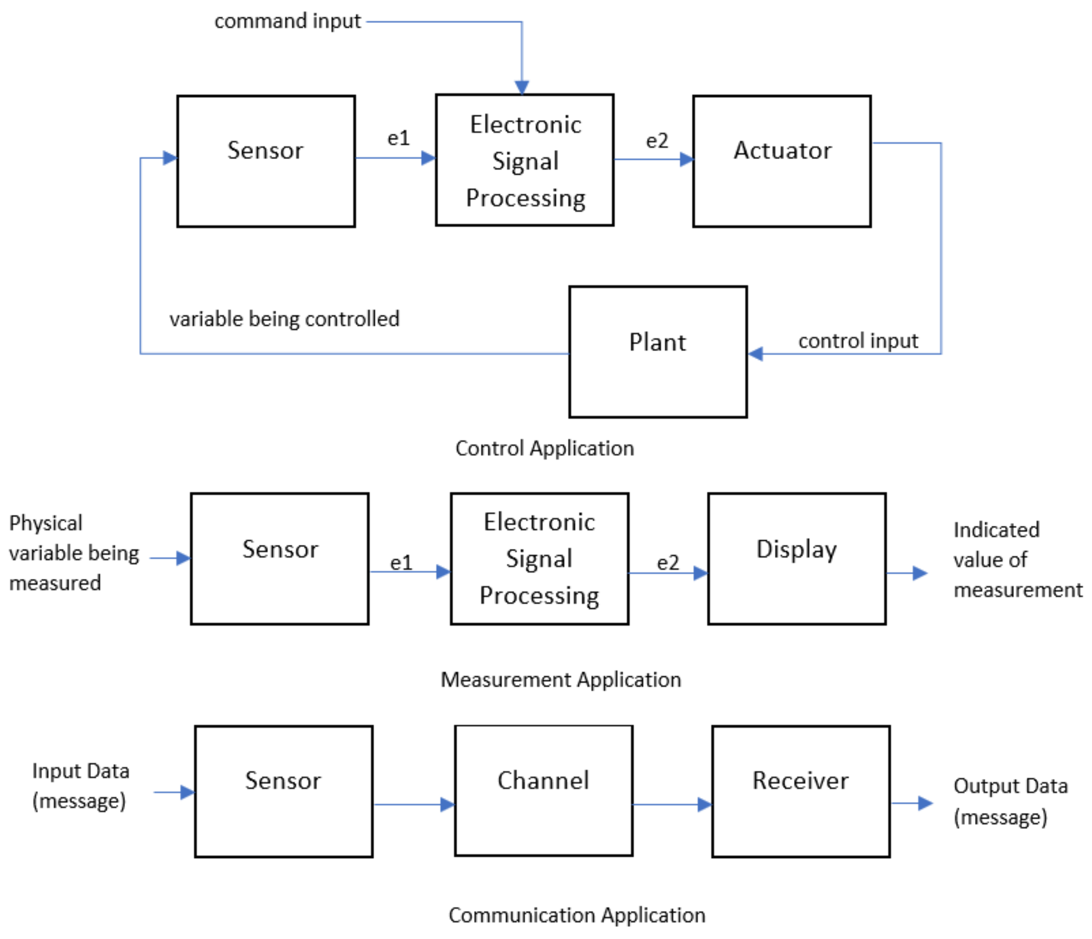


Figure 2.2: Tasks of Vehicle Electronic System [Rib17]

- Phenomeno-logical Model: It is a consolidation of Physical and Stochastic models.
- Data-based Model: It is established on data that is obtained from real sensors.
- White box Model: Formulation of the model is precisely known.
- Black box Model: The opposite of the white box model.
- Gray box Model: It is a mix of Black and White models to define an entire sensor model.

As there are various autonomous driving simulators and they are the closest state-of-the-art simulators for connected car simulator, similar structures and definitions for a sensor model can be adopted. In addition to earlier categorization, sensor models are classified into three types in most of the autonomous vehicle simulators [SMS+20]:

- **Low Fidelity Sensor models:** They are built on geometrical characteristics, such as a sensor's Field of View (FOV) and item location in the virtual world. Its operation basis depends on geometric aspects which is a pattern of objects in surroundings. Low-fidelity sensor models ignore the impact of various environmental factors and material properties. As a result of this, the difference between a real sensor's output and a sensor model's output is greater than for medium- and high-fidelity sensor models. However, it demands less computation power than other types.

- **Medium Fidelity Sensor models:** This model analyzes detection probability as well as physical and geometrical features of the sensor. Physical aspects and detection probabilities are the foundation of this model operation. Contrary to Low Fidelity Sensor models, this type pays regard to environmental factors and material properties. It consists of more detailed information than low-fidelity models.
- **High Fidelity Sensor models:** Rendering techniques are used to create a high-fidelity sensor model as an operation principle. Environment simulation's virtual three-dimensional world is required as an input of the model. Sensor models receive weather and reflection of light information from the 3D world. Some sensor models also take into account factors like diffraction and interference. Thus, it requires high computation power that may create complications for the real-time ability of a model. Sensor models with high fidelity are usually sensor-specific which makes it hard to design generic sensor model.

2.4.1 Sensors in Modern Car

Modern cars comprise numerous sensors for various purposes. They are an essential part of vehicles and perform a huge task on the smooth and safe operation of vehicles. Ordinary cars consist of around 60-100 sensors, and it is expected to reach 200 in the long view [Tom20]. Sensors can be divided into groups regarding their tasks [SMS+20] [Rib17]:

- **Engine Control Sensors:**
  - Mass airflow sensor: It senses the airflow rate into the engine to provide a precise amount of fuel. It can be estimated with the following formula [Bru13] which has dependencies on engine speed in Round per Minute (RPM), Manifold Absolute Pressure (MAP), and Intake Air Temperature (IAT). MAP feeds the engine's electronic control unit with real-time manifold pressure data in kilopascal (KPa) unit. The temperature of the suction pipe is determined by IAT in Kelvin.
    IMAP(MAS) = RPM * MAP/ IAT / 2
  - Engine temperature (coolant temperature) Sensor: It is between -40 degrees to +130 degrees. A mathematical model of the engine coolant temperature was developed in [YSBM00] and it was tested in Simulink. Numerous inputs from the vehicle and environment are required to calculate this value as shown in Figure 2.3.
  - Engine speed /Crankshaft sensor: It detects the rotational speed of a crankshaft in an RPM unit. A reasonable RPM value can be obtained using the speed, axle ratio, and gear status :

    vehicle speed = (engine RPM * wheel tyre perimeter)/(gear ratio * axle ratio)[Shi15]
  - Engine crankshaft angular position sensor: It defines the position of a crankshaft.

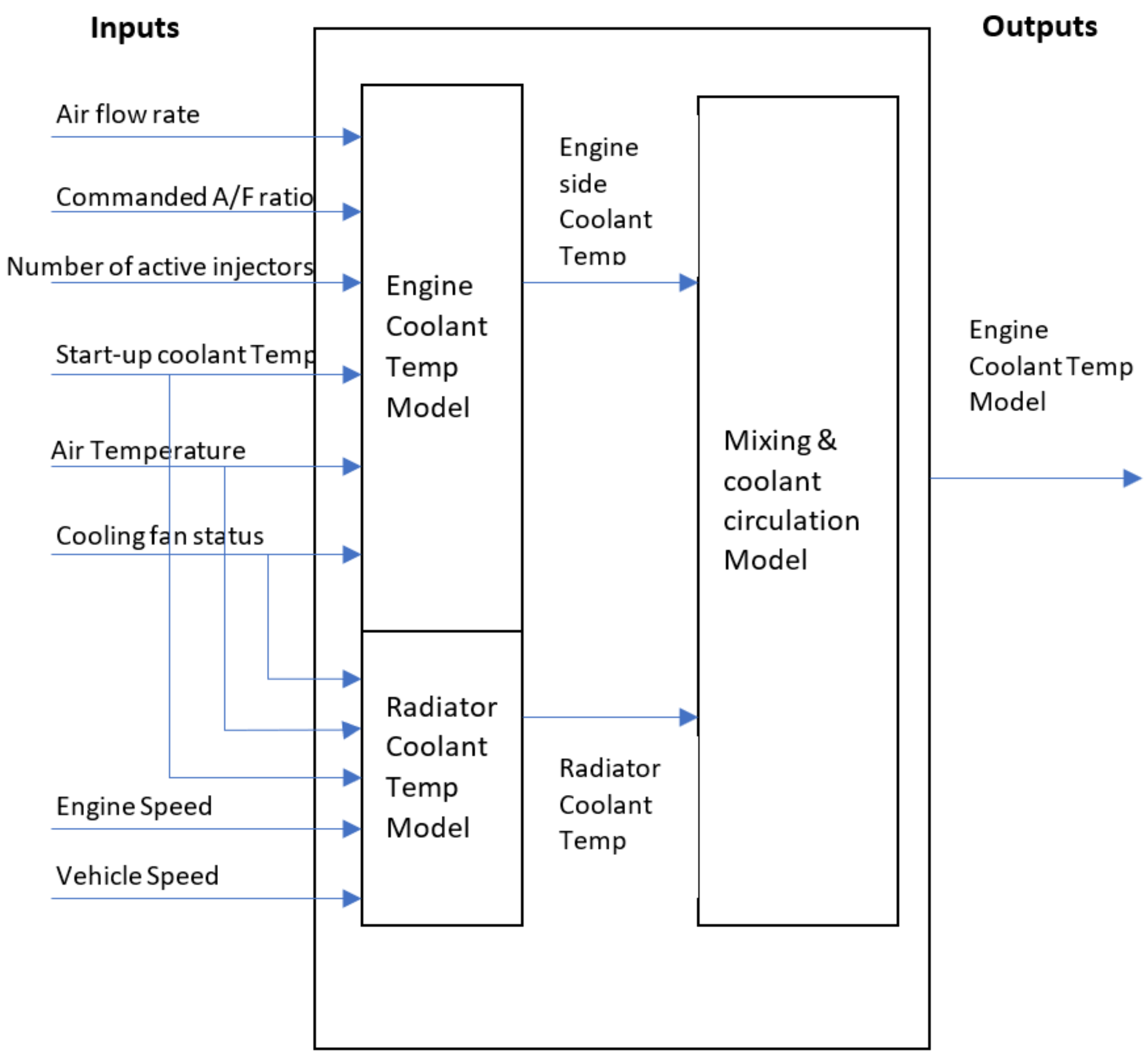


Figure 2.3: Inputs and output of the cooling system model [YSBM00]

- Exhaust Gas Recirculation (EGR) valve position sensor: The EGR valve permits an accurate amount of exhaust gas to be reintroduced into the intake system. Thus, it has to be sensed.
- Intake Air Temperature Sensor: This value depends on the environment and affects engine temperature.
- Intake pipe pressure sensor: It detects the negative air pressure within the intake pipe, after passing through the throttle valve.
- Oxygen Sensors: It determines the amount of remaining oxygen in exhaust gas to provide optimum combustion.
- Fuel flow rate: It is measured to provide a precise amount of fuel. It depends on the vehicle structure, engine capabilities, and instant engine speed of the vehicle.

- **Position sensors**
  - Steering angle sensor: Cars have different steering angle limits to control the direction and the steering angle is utilized also in other applications of vehicles.
  - Fuel tank level sensor: Fuel consumption depends on numerous factors. On the other hand, car brands publish the average fuel consumption of cars for urban and extra-urban areas. This value can be considered for long-distance calculations. Besides, fuel consumption can be calculated depending on Mass airflow [Bru13] which is stated above.

    (gallons of fuel) = (grams of air) / (air/fuel ratio) / 6.17 / 454
  - Throttle valve sensor: The valve controls the exact amount of air supply and fuel that are supplied into engine cylinders.
  - Accelerator and Brake pedal sensors: This value represents pressure on pedals via sensing the pedal status. It is one of the main control inputs of vehicles.
  - Water tank level sensor: The water in the vehicle engine flows around it to cool it down. Automobiles can deal with engine overheating if the engine is in a water shortage.
  - Battery level sensor: The main task of the battery is to help the engine to be started. It Is also used for entertainment, safety, and security applications. A battery of a car does not abruptly charge down. Therefore, synthetic data for it can be defined before the simulation.
  - Gyroscope (Yaw-Rate) sensor [HGG03]: Yaw-rate sensors, which are independent of the reference system, monitor the rotation of a body along a specified axis or the variation in angle per time unit. It is used in ESP, roll-over protection, navigation systems, and distance control.
- **Speed Sensors:**
  - Camshafts sensor: The intake and exhaust valves are controlled by the camshaft. It runs at accurately half the speed of a crankshaft in most applications
  - Wheel speed/Anti-lock Braking System (ABS): It is necessary to measure each wheels' speed to prevent sliding and adjust car balance.
- **Temperature Sensors**
  - Vehicle's interior temperature sensor: The interior temperature is sensed to provide a comfortable environment for the driver and passengers. There is a mathematical model for calculating maximum vehicle cabin temperature [GMD09] which depends on solar radiation, ambition temperature, and cloud cover. CARLA has no support for temperature for the environment or actors.
  - Engine oil temperature sensor: The temperature of oil affects the viscosity of oil which should be thinner or thicker depending on the situation. Hence, it is monitored.

  - Tire air temperature sensor: It has an influence on performance and grip which is prominent in motorsports. There are papers [FRST19] [SJP18] on modeling the temperature of the tire. They provide complex models to determine the temperature which depends on numerous variables.

- **Pressure Sensors**
  - Brake pressure sensor: It is sensed from all four wheels to detect response after braking request. Brake pressure can be applied to all wheels independently. It has a role in ABS and cruise control. There is a simplified equation [Eng22] to determine braking torque which requires information of inertia of wheel, angular deceleration of wheels, vehicle mass, friction between the road and a tire.
  - Suction pressure: A negative difference in pressure between two places is known as suction pressure. It is used in air conditioner systems of cars.
  - Hydraulic reservoir pressure sensor: Power steering and ABS has this pressure information to be operated.
  - Refrigerant pressure sensor: It is required for the air conditioner system.
  - Seat pressure sensor: It detects the number of persons in the vehicle. This information plays a role when a smart car contacts emergency services. It is also a useful sensor for comfort features and seat belt warnings.
  - Tire pressure sensor: Tire pressure directly affects vehicle performance ad passenger safety. However, the sensor value depends on numerous parameters that cannot be modeled easily.

- **Others**
  - Knock sensors: It detects sudden rises in cylinder pressure.
  - Fuel quality: It influences engine performance in the short term and engine maintenance in the long term.
  - Crash sensor: Crash sensors trigger airbags in the car and the sensor data is utilized in safety-related applications. Figure 2.4 shows the crash/impact sensors in a vehicle.
  - Airbag sensor: It depends on the intensity of collision which is measured by a crash sensor. Seat belt locks and door locks also have a dependency on airbag sensors because seat belt locks should be activated to keep the passenger safe and doors are unlocked to provide an escape from a vehicle after a crash. It can be simulated by setting a crash level in the simulator and seat belts' status can be randomly decided.
  - Rain sensor: Automatic windscreen wipers are triggered by this sensor.
  - Door/boot state detection sensor: It is a safety feature which exists in almost all modern cars.
  - Day/night detection sensor: It helps to put the vehicle lights on automatically during the night.

- **Advanced Driver-Assistance Systems (ADAS)/Autonomous Driving Sensors:**

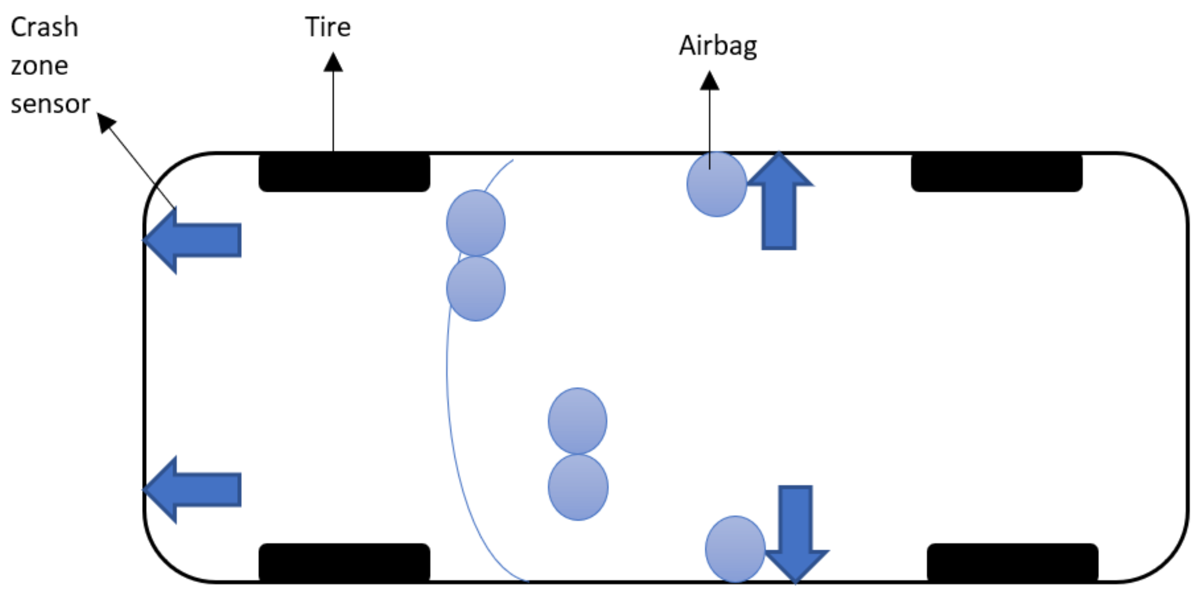


Figure 2.4: Crash sensors and Airbags in a car [HGG03]

- Radar sensor: There are diverse Radar sensors for specific tasks that a car can have multiple of them. It mainly measures the speed and distance of objects in the car's close area. It uses radio signals for measurement.
- Lidar sensor: It helps a car to perceive the environment around it especially for autonomous driving scenarios. It uses infrared lights for realizing a 3D environment.
- Distance/Ultrasonic sensors: It makes use of the echolocation principle to observe the distance of objects around the vehicle. Radar and Lidar sensors work with ultrasonic sensors to provide more accurate data. There are different approaches to provide blind spot detection functionality such as optical blindspot sensors. Ultrasonic sensors also serve this purpose. The data is utilized to warn the driver when any other vehicle approach from the rear backside.
- Camera: It gained importance with the trend on autonomous driving functionalities. Modern vehicles mostly have multiple cameras that also support drivers for other scenarios, such as car parking.
- GPS sensor: It is fundamental for navigation and autonomous driving, it uses satellite signals to calculate the vehicle's position.

3 Related Work

There is a wide range of vehicle simulators that serve a variety of purposes. This chapter analyzes the open-source and commercial vehicle simulators in order to comprehend the pros and cons of the existing simulator. It is planned to construct the connected car simulator on an existing simulator. For that reason, a deep examination of related work regarding existing simulators plays a significant role in further decisions about system architecture and implementation.

3.1 AirSim

AirSim [Air22; SDLK18] is high fidelity simulation platform developed by Microsoft for deep learning, computer vision, and Reinforcement Learning (RL). It is an Unreal Engine-based simulator for drones, automobiles, and other vehicles. It provides a highly realistic rendering that consists of reflections of lights, lifelike shadows, truthful weather effects. Even though Airsim was originally designed to simulate drone flights, now it also has support for vehicles, most notably automobiles. Its strengths are a high-fidelity realistic driving environment and pre-made, advanced car models [ND18]. However, new map and environment creation is a tough process. Desired scenarios can be created by using Unreal Engine's platform which is a gradual process and requires experienced developers in Unreal Engine.

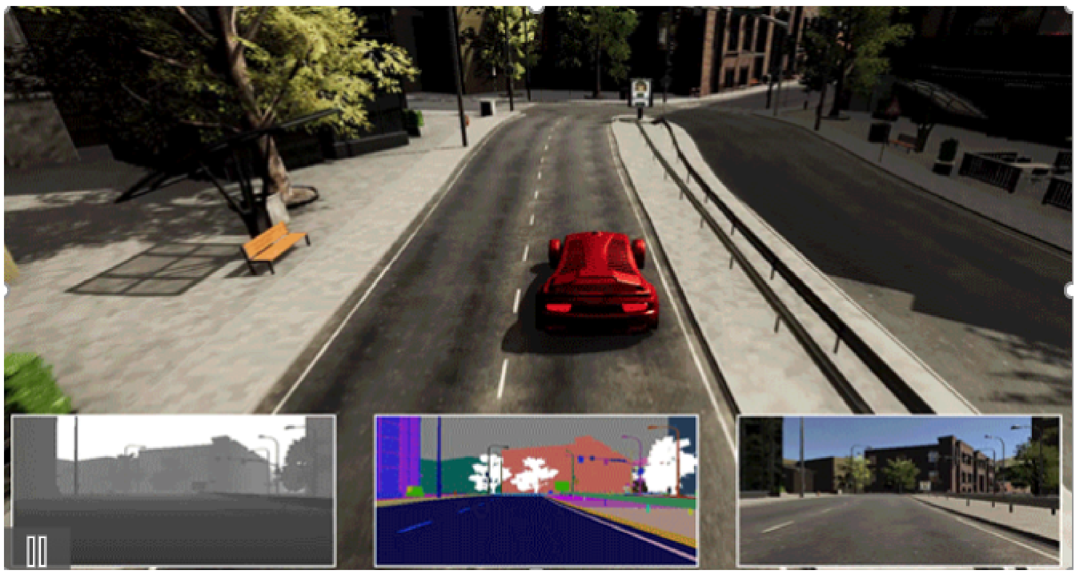


Figure 3.1: Screenshot of AirSim [Kyl18]

3.2 Gazebo

Gazebo [AKC+15] [KTTS21] is an open-source simulator that is defined as a robotic simulator. It is currently under the supervision of Open Source Robotics Foundation. It is enormously popular in a variety of applications, not just in car simulation. It has multiple physics engines that provide modularity. Despite its resemblance to game engines, Gazebo provides quite higher fidelity in physics modeling. Due to the fact that it is open-source, and it is inside the Robotic Operating Systems bundle, it has a vast community in that researchers can benefit from diverse resources and plugins. It was developed on Ogre3D graphic engine. Researchers can create any object from scratch by deciding its graphics and physical properties. Its strengths are the opportunity to customize and control the model and environment in a custom way [ND18]. Nevertheless, it is not a specific car simulator. This means that users may have to build car models and environments from the very beginning unless there can not find plugins that already created models for the same purpose.

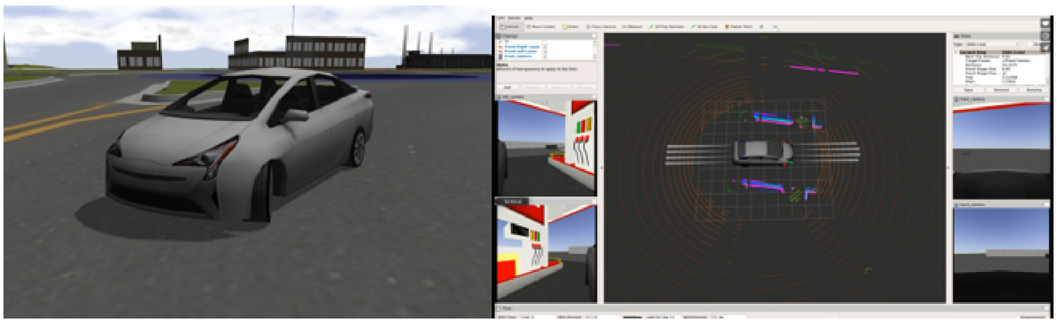


Figure 3.2: Gazebo Simulator [Tul17]

3.3 CARLA

CARLA [DRC+17] (Car Learning to Act) is an open-source simulator built on Unreal Engine 4 for the development, training, and test of autonomous driving systems. It was released in 2017 by Intel labs and Toyota Research Institute. It extends Unreal Engine 4 technically by supplying sensor data, such as Lidar, and ground truth depth maps, as well as ground truth semantic segmentation [Nik21]. Users can configure variable sensor suites and these sensors deliver signals for training Artificial intelligence (AI) and during driving. It consists of fundamental sensors for autonomous driving, such as Lidar, GPS, depth camera, collision detector, IMU, Radar. The CARLA Simulator and the CARLA Python API module are the two primary components of CARLA. The simulator is in charge of the majority of the hard work that is controlling the logic, physics, and rendering of all the actors and sensors in the scene [18]. Thus, it needs a dedicated Graphics processing unit (GPU). Users can control the simulator and obtain data from it via Python API. It also consists of extensions like Scenario Runner [CAR22b] and tutorials for creating new sensors that are useful for this thesis's goals.

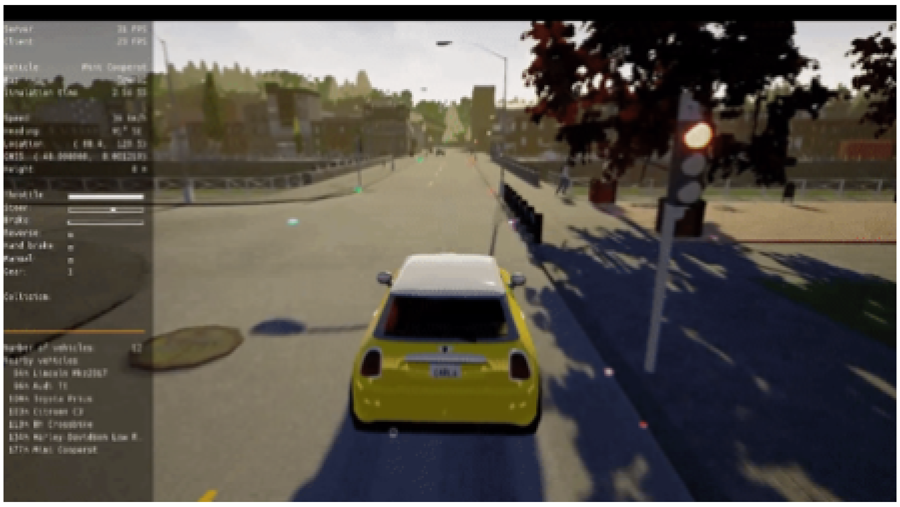


Figure 3.3: Carla Simulator [CAR22a]

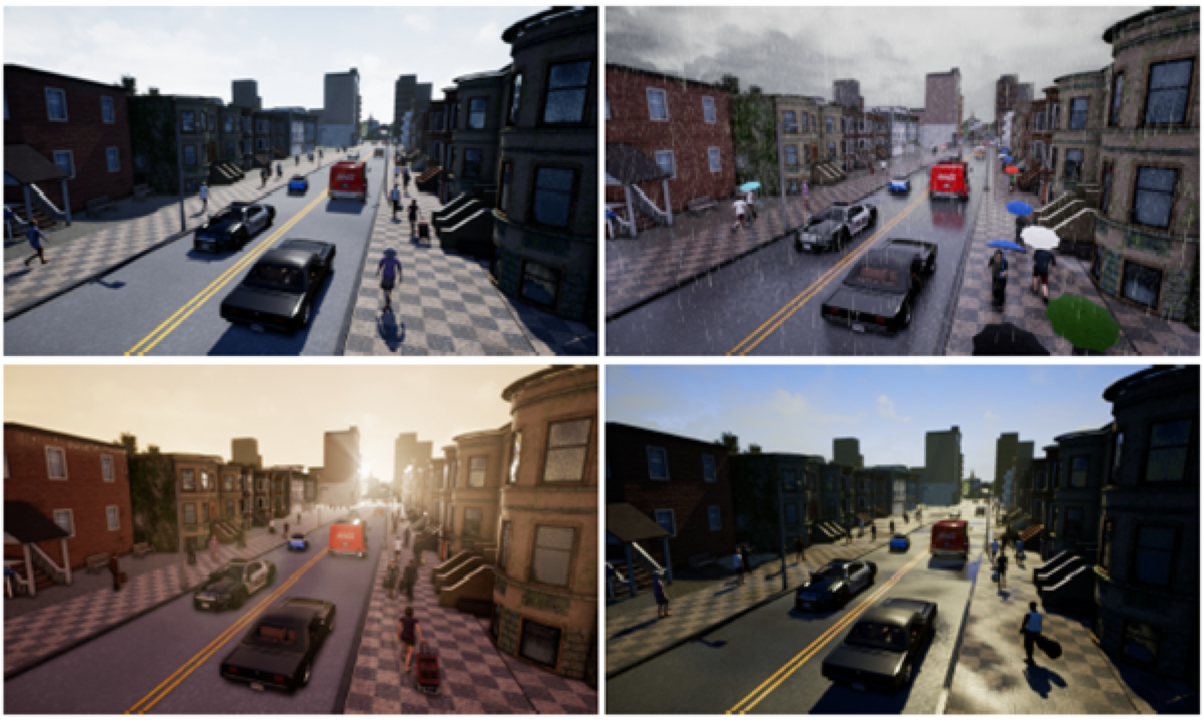


Figure 3.4: Different Weather Conditions in CARLA [DRC+17]

3.4 LGSVL

LGSVL [RST+20] [KTTS21] is a multi-robot autonomous vehicle simulator by LG Electronics America. It is built upon a Unity game engine that offers high-quality graphics. Even though its main goal is being an end-to-end autonomous driving simulation, there are examples of simulations for autonomous robotic applications at indoor places. The paper published by LGSVL developers

[RST+20] asserts that other platforms like CARLA and Airsim were primarily built as research platforms to assist RL or synthetic data creation for machine learning. This means integrating them with a user's autonomous driving stack and communication bridge usually takes a substantial amount of time and work. However, LGSVL has a solution at that point. Multiple autonomous driving systems can be linked to the LGSVL Simulator at the same time. Through a specialized bridge, any autonomous driving system may connect with the simulator, allowing interaction between multiple autonomous systems in a single simulation environment. As in CARLA, users can create a custom sensor and add it as a sensor plugin. Map creation is also possible in the simulator.

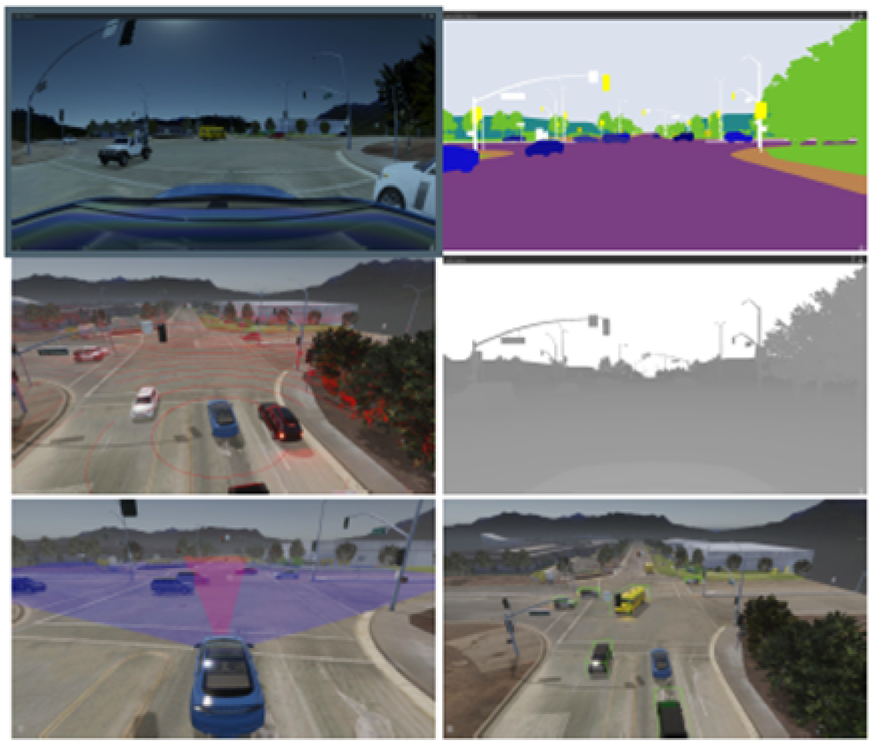


Figure 3.5: Sensors in LGSVL [RST+20]

3.5 MATLAB Automated Driving Toolbox

Automated Driving Toolbox [MAT22a] [KTTS21] is a simulation extension of MATLAB for ADAS and automated driving systems. It consists of example applications, such as adaptive cruise, parking valet, emergency braking. It is possible to import maps from other applications, e.g., HERE HD Live Map. Besides, it has a visualizer integrated inside it that enables to see real-time sensor detection and tracks. A realistic simulation environment can be reached by a co-simulation system with Unreal Engine. Users cannot control the weather conditions in this simulation platform. One disadvantage of this platform is lacking a mature community because it is a commercial product.

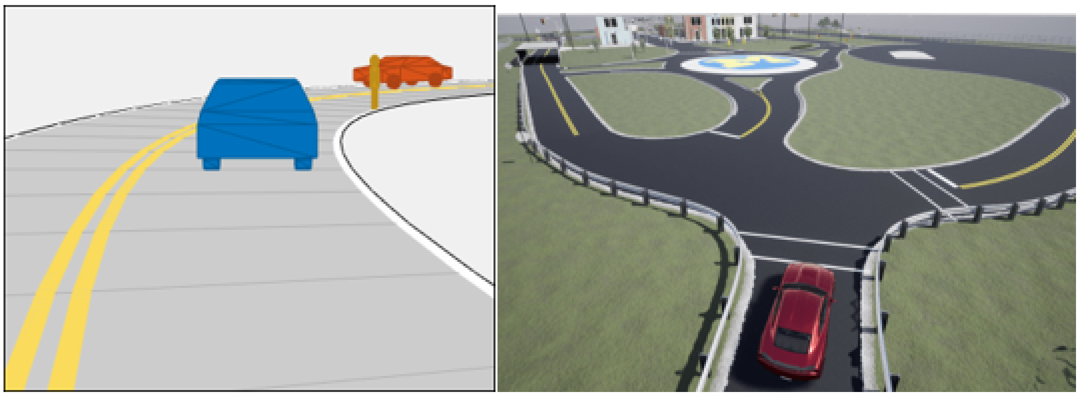


Figure 3.6: Automated Driving Toolbox: left image is scenario creator and right one is co-simulation with Unreal Engine [MAT22a]

3.6 Torcs

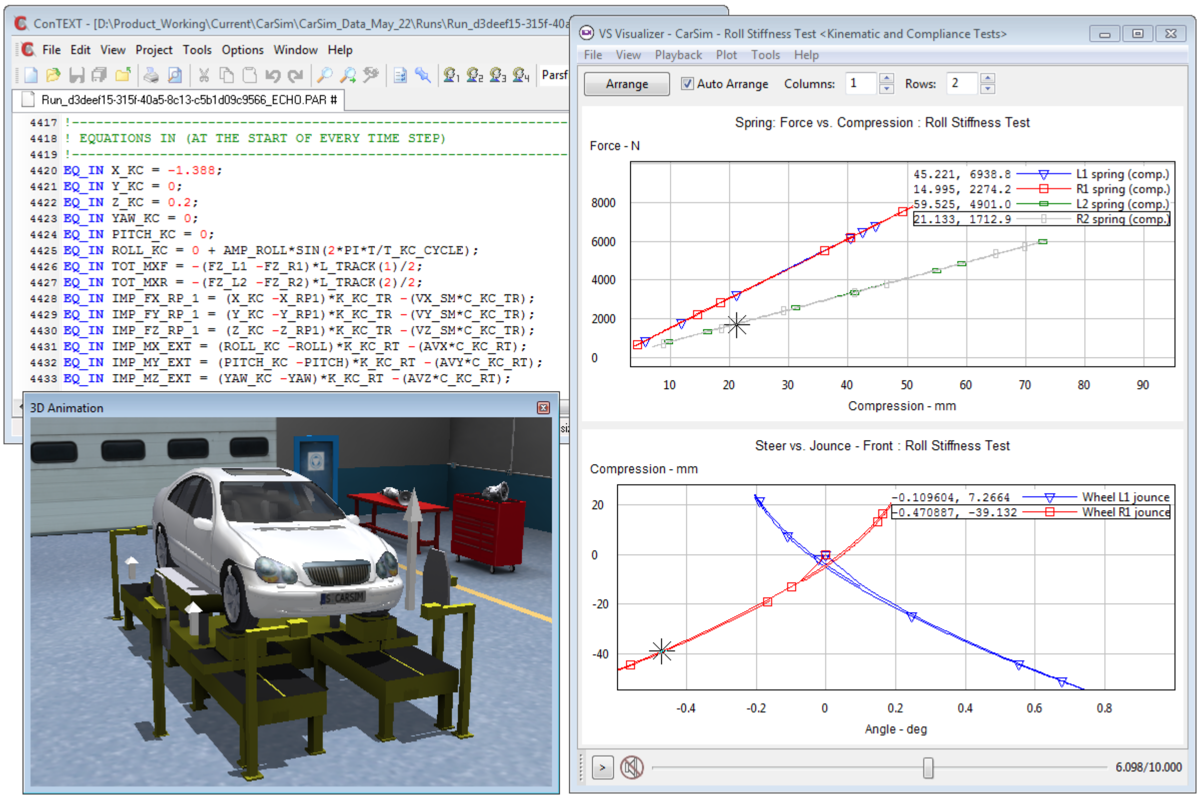
Torcs [Mer18; Wym20] is one of the oldest platforms on the list which was released in 1999. It was originally created as a racing car game rather than an autonomous vehicle simulator. However, it was utilized in various research to test AI algorithms because of its portability and modifiability. Screenshots from the game are used for training and an autonomous system is tested in a race at The Princeton Vision Group research [CSKX15; Wym20]. However, it is not an eligible simulator for this thesis since its low graphical reality and lack of sensor support.

Figure 3.7: Torcs [Wym20]

3.7 Carsim

CarSim [Car; KTTS21] is a system dynamics simulation platform developed by Mechanical Simulation cooperation. It supplies remarkably mature dynamic models of cars. CarSim is utilized by seven of the top ten automobile manufacturers [Car]. Users can create complicated test scenarios.

Various synthetic sensor data for ADAS and autonomous driving can be retrieved from the simulator. The strength of this simulator advanced mathematical model for car dynamics and sensors. It can be also connected with other simulator platforms, such as CARLA and MATLAB to create co-simulations. However, it is a commercial product that makes it not suitable for this thesis. Another disadvantage of the CarSim is that it has a restricted ability to efficiently create customized upper-level algorithms.


ConTEXT - [D:\Product_Working\Current\CarSim\CarSim_Data_May_22\Runs\Run_d3deef15-315f-40a
File Edit View Project Tools Options Window Help
Run_d3deef15-315f-40a5-8c13-c5b1d09c9566_ECHO.PAR #
3D Animation
VS Visualizer - CarSim - Roll Stiffness Test <Kinematic and Compliance Tests>
File View Playback Plot Tools Help
Arrange
Auto Arrange
Columns: 1
Rows: 2
Spring: Force vs. Compression : Roll Stiffness Test
Force - N
Compression - mm
45.221, 6938.8 L1 spring (comp.)
14.995, 2274.2 R1 spring (comp.)
59.525, 4901.0 L2 spring (comp.)
21.133, 1712.9 R2 spring (comp.)
Steer vs. Jounce - Front : Roll Stiffness Test
Compression - mm
Angle - deg
-0.109604, 7.2664 Wheel L1 jounce
-0.470887, -39.132 Wheel R1 jounce
6.098/10.000

```
!---------------------------------------------------------------------
! EQUATIONS IN (AT THE START OF EVERY TIME STEP)
!---------------------------------------------------------------------
EQ_IN X_KC = -1.388;
EQ_IN Y_KC = 0;
EQ_IN Z_KC = 0.2;
EQ_IN YAW_KC = 0;
EQ_IN PITCH_KC = 0;
EQ_IN ROLL_KC = 0 + AMP_ROLL*SIN(2*PI*T/T_KC_CYCLE);
EQ_IN TOT_MXF = -(FZ_L1 -FZ_R1)*L_TRACK(1)/2;
EQ_IN TOT_MXR = -(FZ_L2 -FZ_R2)*L_TRACK(2)/2;
EQ_IN IMP_FX_RP_1 = (X_KC -X_RP1)*K_KC_TR -(VX_SM*C_KC_TR);
EQ_IN IMP_FY_RP_1 = (Y_KC -Y_RP1)*K_KC_TR -(VY_SM*C_KC_TR);
EQ_IN IMP_FZ_RP_1 = (Z_KC -Z_RP1)*K_KC_TR -(VZ_SM*C_KC_TR);
EQ_IN IMP_MX_EXT = (ROLL_KC -ROLL)*K_KC_RT -(AVX*C_KC_RT);
EQ_IN IMP_MY_EXT = (PITCH_KC -PITCH)*K_KC_RT -(AVY*C_KC_RT);
EQ_IN IMP_MZ_EXT = (YAW_KC -YAW)*K_KC_RT -(AVZ*C_KC_RT);
```

Figure 3.8: CarSim Simulator [CLLH20]

3.8 MetaDrive

MetaDrive [LPX+21] was developed to deliver solutions for RL and generalization problems of the RL. It is an open-source driving simulation platform. The main advantage of MetaDrive is that users can create numerous maps and traffic scenarios easily to support RL tasks. It provides map blocks and algorithms to generate maps randomly. It is a useful feature for producing unseen environments for RL methods. It is possible to control multiple cars. The weaknesses of the simulator are yielding low-quality rendering and being lack of weather, advanced lighting, and environmental objects such as pedestrians, traffic lights, buildings. Basic sensor models for RL tasks are available in the simulator. Lidar-like cloud points, cameras, and sensory data for the vehicle location and surrounding objects are some of the sensor models. The feature that makes difference from other simulators is being lightweight and requiring very limited resources.

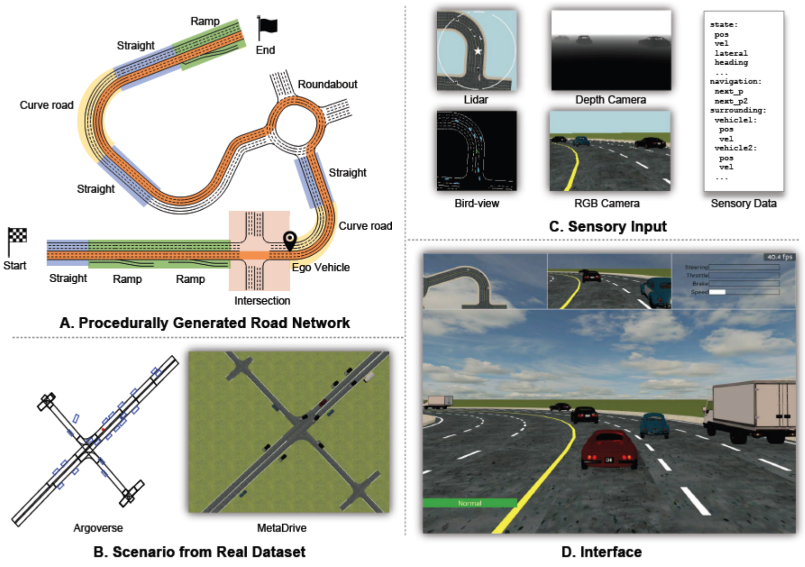


Figure 3.9: MetaDrive [LPX+21]

3.9 SUMO

SUMO [LBB+18; PR12] is an open-source simulation platform that aims to create large traffic networks which is developed by DLR in 2000. SUMO can perform the major responsibilities of traffic simulation construction, execution, and assessment [YXM+21]. It provides 2D microscopic simulations that consist of vehicles, public transport, and pedestrians. For this reason, it is not a sufficient platform for autonomous driving scenarios. Thus, it is co-simulated with diverse simulators, such as CARLA thanks to being cross-platform. SUMO's main goal is to generate and move through complex traffic systems. It is more concerned with traffic flows than with individual automobiles' behavior [ND18]. Different types of vehicles are supported by SUMO. However, it does not generate any synthetic sensor data.

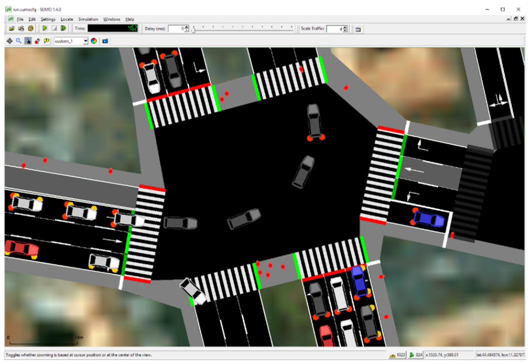


Figure 3.10: Scene from SUMO [co422]

3.10 SUMMIT

SUMMIT [CLLH20] [SUM22] (Simulator for Urban Driving in Massive Mixed Traffic) is a simulator that aims to create high-fidelity data for unregulated, intense traffic on real-world maps. It

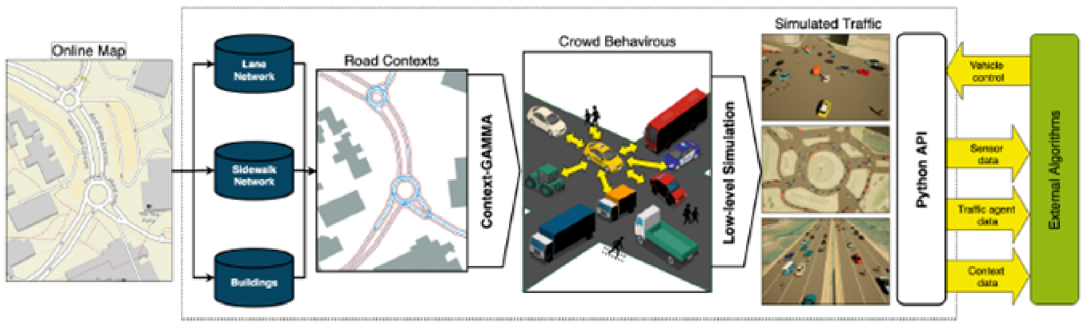


Figure 3.11: Overview of SUMMIT [CLLH20]

is developed based on the CARLA simulator. It can be considered an extension of the CARLA simulator. Any place in the world which is supported by OpenStreetMap can be simulated by SUMMIT with a dense realistic traffic environment. It simulates swarms of heterogeneous traffic agents based on arbitrary locations. Since it is built on CARLA, it contains low-level and high-level features of CARLA, such as sensors and weather control. Its strength is supporting unregulated complex behaviors in traffic, using real-world maps, and providing dense traffic. SUMMIT can also import maps that are created in SUMO. Some simulators like Autonovi-sim utilize more advanced motion models but are limited to predetermined maps, whereas SUMO supports real-world maps but uses basic rule-based behaviors [CLLH20].

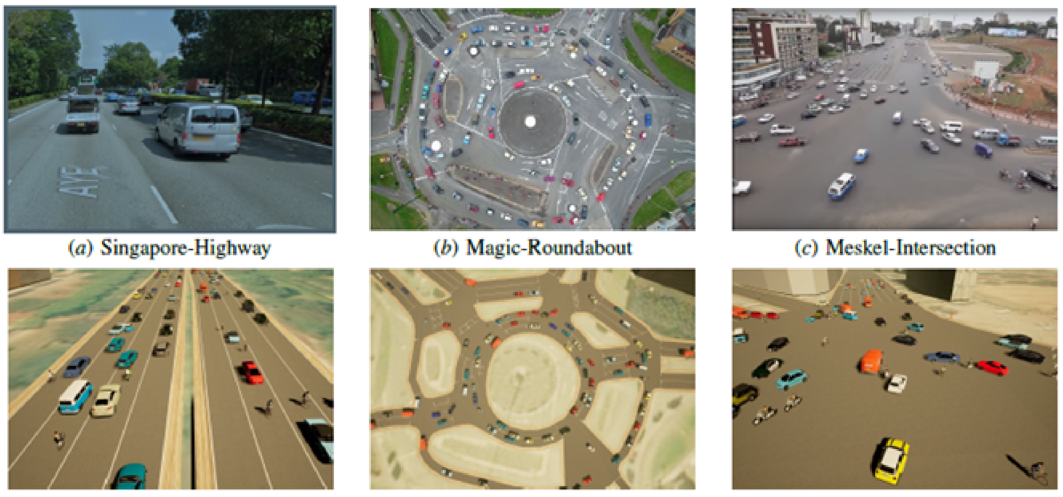

(a) Singapore-Highway
(b) Magic-Roundabout
(c) Meskel-Intersection
(d) Singapore-Highway in SUMMIT
(e) Magic-Roundabout in SUMMIT
(f) Meskel-Intersection in SUMMIT

Figure 3.12: Scenes in SUMMIT based on real world map [CLLH20]

4 Simulation System and Concept

This chapter delineates the system's model and architecture, as well as elaborates how our requirements are fulfilled. The architecture is illustrated using the Unified Modelling Language (UML), use case diagrams, sequence diagrams, and block diagrams to provide a guideline for the implementation section. It is essential to define the concept explicitly since a simulator for connected car scenarios is developed from scratch and provides the fundamentals for future improvements.

4.1 Software System Architecture

The system architecture analyzes components of the software system and the interaction between its components. The system is comprised of five parts:

- Graphical User Interface: A user can create scenarios and configure scenario and vehicle-related values using this component.
- Simulator: This component provides a physical and rendering engine for the system.
- Data Generator: This component generates realistic values for vehicles.
- Data Record: It ensures data that is generated data from Simulator and Data Generator is stored in a uniform way.
- Vehicle Control: This module maintains a distributed control option over vehicles during the run of the simulator.

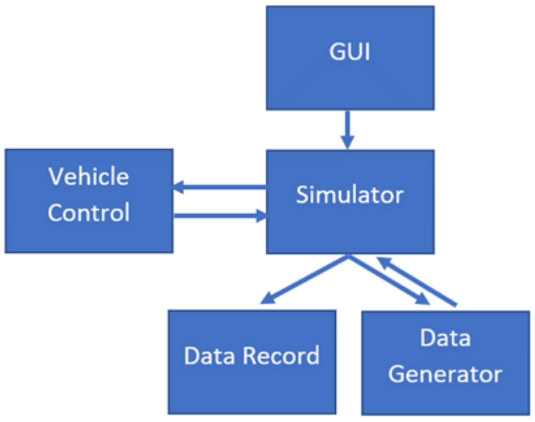


Figure 4.1: Software Architecture of a Simulator for Connected Car Scenarios

4.1.1 Graphical User Interface

The user interacts with this component to start the simulation. Besides that, the user can configure parameters for other components using this module via buttons, checkboxes, text boxes, and combo boxes. It enables a user-friendly interface to set up desired scenarios. The user can create a custom map with ease in different sizes. It helps the user to define vehicles' initial position, speed, and heading values. Also, the course of vehicles can be adjusted in the GUI. The user can select which values are recorded and the storage location for generated data. Options for various types of vehicles are essential to create a more realistic simulation because not every vehicle has the same features on the streets. Hence, the user can select different car types and make changes in car properties.

One advantage of the GUI is that the user does not need to make any changes in the code of the system in order to create a scenario. It removes the chance of making typos and coding mistakes for scenario creation. Additionally, the user can save the scenario configuration in the personal storage. This gives the possibility to rerun and make changes on scenarios without setting all values again from scratch.

This component interacts with the simulator component. It transfers the data to the simulator component in order to start a custom scenario. It has no rendering and physics engine. It just presents an environment to the user for setting the values for other modules.

It is expected that the GUI can run on an average computer without the need for extra computing power. This also means that it shall be compatible with at least one common operating system and is lightweight enough to run on a decent personal computer.

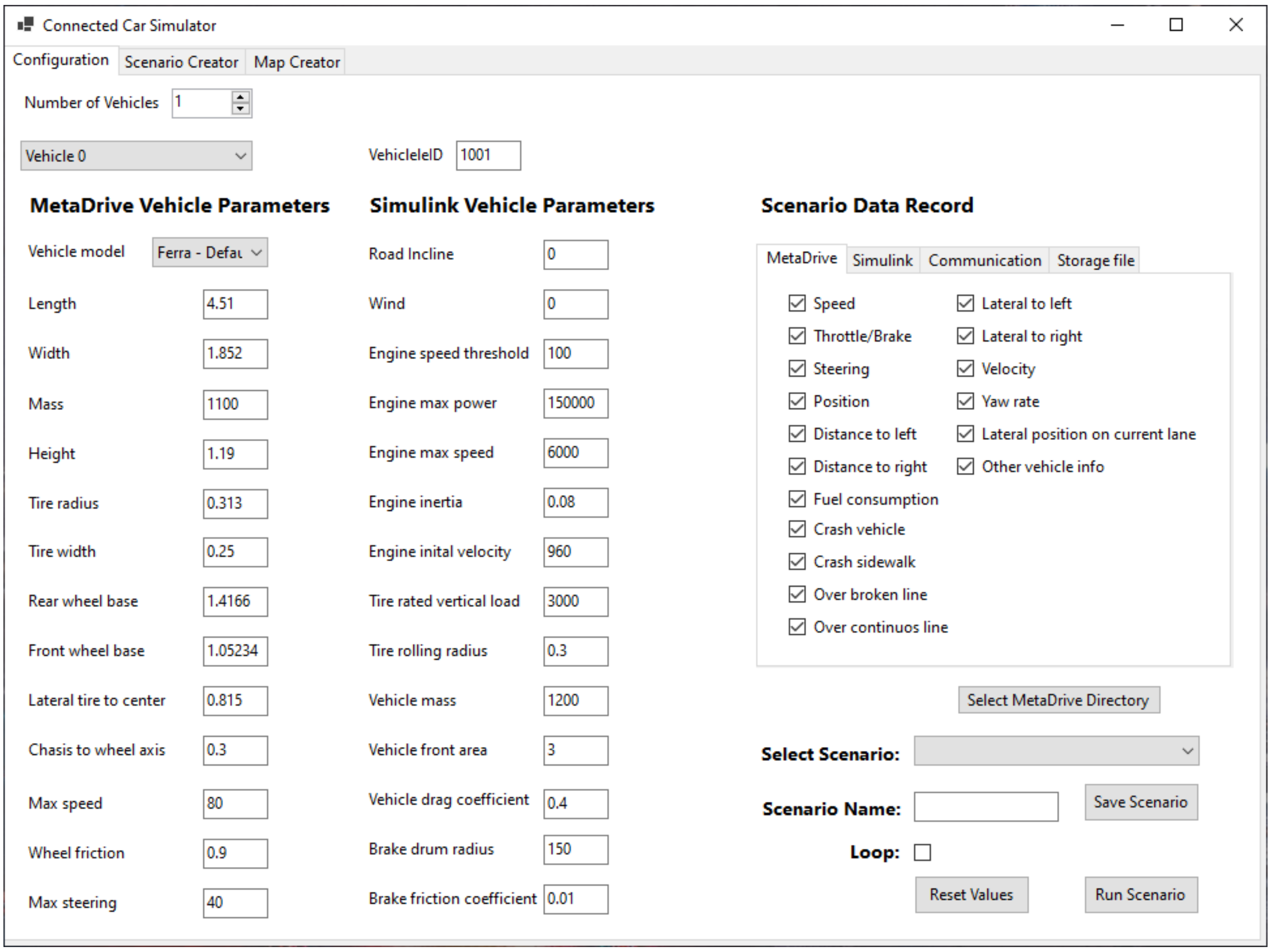
The GUI is presented in Figure 4.2.

Fundamental Design Decisions
Since it is required that the Graphical User Interface runs on a common operating system, it was decided to develop the GUI as a Windows Form Application (.Net Framework). Microsoft Visual Studio 2019 is preferred as an integrated development environment. Windows Form Application has adequate support, tools, and libraries to realize the GUI. The GUI can be installed on other computers using the Setup Wizard feature and it decreases the hardship of the setup process. C# is the programming language for development. The reason for choosing C# is that it is a highly popular programming language and various resources about C# can be found on online platforms. This makes it uncomplicated to extend the GUI for future studies.

4.1.2 Simulator

The simulator component is the heart of the system to that all other components are connected. It provides an environment to simulate the behaviors of vehicles and observe their data. It provides an interface that the user can control vehicles and connects to a remote server to receive commands for vehicles and transfer sensor data of vehicles. This part runs in conjunction with the Vehicle Control component.

Vehicle behaviors regarding the commands are in immense relation with the physic engine provided by the Simulator component. The response to the brake, throttle, and steering is managed by the physic engine.



Number of Vehicles: 1

Vehicle 0

VehicleleID: 1001

MetaDrive Vehicle Parameters

Vehicle model	Ferra - Defau
Length	4.51
Width	1.852
Mass	1100
Height	1.19
Tire radius	0.313
Tire width	0.25
Rear wheel base	1.4166
Front wheel base	1.05234
Lateral tire to center	0.815
Chasis to wheel axis	0.3
Max speed	80
Wheel friction	0.9
Max steering	40

Simulink Vehicle Parameters

Road Incline	0
Wind	0
Engine speed threshold	100
Engine max power	150000
Engine max speed	6000
Engine inertia	0.08
Engine inital velocity	960
Tire rated vertical load	3000
Tire rolling radius	0.3
Vehicle mass	1200
Vehicle front area	3
Vehicle drag coefficient	0.4
Brake drum radius	150
Brake friction coefficient	0.01

Scenario Data Record

✓ Speed	✓ Lateral to left
✓ Throttle/Brake	✓ Lateral to right
✓ Steering	✓ Velocity
✓ Position	✓ Yaw rate
✓ Distance to left	✓ Lateral position on current lane
✓ Distance to right	✓ Other vehicle info
✓ Fuel consumption	
✓ Crash vehicle	
✓ Crash sidewalk	
✓ Over broken line	
✓ Over continuos line	

Select Scenario:

Scenario Name:

Loop: ☐

Figure 4.2: GUI of a Simulator for Connected Car Scenarios

The environmental model is an element of this component and it enable to make changes in the environment, like rain or wind, are provided by the simulator component.

Several vehicle models in the Simulator component shall give the user the freedom to make diverse scenarios. Vehicles can have different shapes, speed limits, braking responses, and fuel consumption standards. These are realized in the vehicle model part of this component.

Also, it is required to generate vehicle data as realistic as possible. Some simulators are very robust at this point and have their own high fidelity sensor models. However, they consume a huge amount of computing power, and they are not supported by decent computers. Thus, the balance between computing requirements for data generation and providing realistic data is critical. This topic is also related to the Data Generator component. This task can be realized at the Simulator component or data can be generated in another tool/software.

Fundamental Design Decisions

The simulator component is the base of the system. Thus, spotting the optimal simulator is a crucial task. Existing vehicle simulators are discussed in Chapter 3 Related Work. Since a simulator for connected cars is built up over an existing simulator, features of them are listed in Table 4.1 to find the most advantageous one for the thesis.

Each simulator has pros and cons that contribute to a variety of scientific studies. Commercial ones are not preferable. Hence, CarSim is not suitable. MATLAB lets researchers and students use some extensions free, however, terms for the Automated Driving Toolbox extension are unknown. Torc is neither a convenient platform for this thesis since a limited number of vehicles, sensors, and maps.

Even though Gazebo provides huge flexibility and varied sensor suites, it is requiring plenty of time to design and generate complex, large-scale test environments.

AirSim, CARLA, LGSVL has a satisfactory capacity to test autonomous vehicles. They compromise miscellaneous features to help researchers to create challenging test environments. They are still in the development phase, and this can occasionally cause bugs or errors. As Airsim was not initially designed for automobiles, it has less community and tutorials related to driving than LGSVL and CARLA.

LGSVL and CARLA are best fits for a connected car simulator. A new simulator can be built upon them. The advantage of LGSVL over CARLA is providing sensor plugins that researchers can create unique sensors for V2X, V2V, and V2I. Nevertheless, it has less documentation and community than CARLA. Besides, several software platforms are built on CARLA or co-work with it.

However, CARLA and LGSVL require large graphical processing units and computing processing units. CARLA needs at least 6 gigabytes of GPU. 8 gigabytes are the suggested size. Thus, these simulators cannot work on a decent computer. This limitation is one of the root causes to choose MetaDrive for this thesis. MetaDrive is not advanced in rendering, data generation, and environmental features. Since MetaDrive is launched a short time ago, it is not mature and it is likely to deal with some bugs. Nevertheless, it is a lightweight simulator and has main features to construct the thesis on it.

Table 4.1: Features of Simulators-Part 1

Features	Microsoft AirSim	Gazebo	Carla	LGSVL
Realistic Graphics	Y	N	Y	Y
Controlling Environment	Y	N	Y	Y
Controlling Multiple Car	Y[62]	Y	Y	Y
Communication between multiple cars	U	U	U	Y[31]
Reproducibility [1]	U	U	Y	Y
Scenario Creation	N	Y	Y	Y[61]
Traffic flow	Y, but not mature	N	Y	Y
Vehicle Dynamics	Y	Y	Y	Y
Sensors	Camera, Barometer, Imu, Gps,Magnetometer, Distance Sensor, Lidar	Camera, Laser, IMU, Lidar, Radar, GPS, depth-camera . . .	Camera, Laser, IMU, Lidar, Radar, GPS, depth-camera . . .	Camera, Lidar, Imu, GPS, Radar, CanBus Sensor
Sensor Creation Support	N	Y	Y	Y
Data Recorder	Y	N	Y	Y
Interfaces to other software	U	Y	Y	Y
Level of Maturity*	3	4	3	2
License	MIT License	Open source, Apache 2.0	MIT License	Open source

Table 4.2: Features of Simulators-Part 2

Features	MATLAB Automated Driving Toolbox	Torcs	CarSim	MetaDrive
Realistic Graphics	Y, combination with Unreal Engine	N	Y	N
Controlling Environment	N	N	N	N
Controlling Multiple Car	U	Y	Y	Y
Communication between multiple cars	U	U	U	N
Reproducibility [1]	Y	N	U	U
Scenario Creation	Y	N	Y	N
Traffic flow	Y	N	Y	Y**
Vehicle Dynamics	Y	N	Y	Y
Sensors	Camera, Radar, Lidar	Camera	Up to 99 motion and ADAS sensors	Camera, Lidar-like sensor, Gyroscope, GPS
Sensor Creation Support	N	N	U	N
Data Recorder	U	N	Y	N
Interfaces to other software	Y	Y	Y	N
Level of Maturity*	3	2	4	1
License	Commercial	Open source	Commercial	Open source

*Y: Yes , N:No, U: Unknown - 1 shows the lowest level and 4 the highest ** It has limited capacity and tends to crash with other vehicles.

4.1.3 Vehicle Control Interface

Providing an interface for controlling vehicles in distributed servers is a desired feature. In this way, users can develop and run their main connected car algorithm on another computer and test their algorithm using the interface. The vehicle control interface shall be integrated into users' main connected car algorithm with less effort. Transferring data in a standard way and mapping the variables to the same limits is a task of this component. The simulator components send the vehicle-related data, such as their speed, heading, location, and sensor values. The vehicle component transmits the values for the steering wheel throttle and brake to control the vehicle's course.

Fundamental Design Decisions
In such a case, message queueing is a solution for transmitting messaging and decoupling software. These kinds of applications are also called message brokers. There are numerous messaging applications, such as RabbitMQ and ZeroMQ. They mainly provide reliability, flexibility, and tracing. They can be considered as a middleman between services and deal with the definition of queues and where and how queues connect.

RabbitMQ is a messaging broker which is a non-blocking asynchronous application. Direct, Headers, Fanout, and Topic are exchange types of RabbitMQ [Rab22].

ZeroMQ is a lightweight communication application based on sockets. It is a library for asynchronous messaging that is utilized in a variety of messaging services. The ZeroMQ runs without the specialized message broker [Pri].

On the other hand, MQ Telemetry Transport (MQTT) is an option to realize this component. It is a messaging protocol widely used in Internet of Things applications. The appealing aspect of MQTT is its being lightweight [MQT22]. It uses a publish/subscribe architecture.

The main difference between RabbitMQ and ZeroMQ is transmission speed and persistence. ZeroMQ is almost 6 times faster, but RabbitMQ still transfers 4-10K messages/sec [Pri]. This rate is already more than enough for realizing the component. ZeroMQ has no persistence feature which means the message can be lost when one side of the communication becomes offline for a while. Besides, RabbitMQ is a better option for extendibility and has more features and sources that users can easily integrate into their algorithm.

Sophisticated message routing is not handled by MQTT. Nevertheless, RabbitMQ is very robust on this point. MQTT is more basic and used for moderate applications. Since it is also easy to implement and integrate RabbitMQ, there is not a huge difference between them regarding this thesis' goals.

It was decided to realize the component using RabbitMQ because of the stated reasons above.

4.1.4 Data Record

It is vital to store data of the simulation in order to assess a main connected car algorithm. In this way, users can comprehend variations of their algorithm's effects on vehicles. Storing data in a standard way makes it easier to compare simulation data and has an advantage in extendibility. Users can increase the number of parameters in stored data in a handy way and without affecting other parts of the simulator

Fundamental Design Decisions
There are two main message formats in IoT and in computer applications. The first of them is JSON, which is a data interchange format. It is a text format and highly human-readable. A JSON object consists of a key and value chain as in the example in Listing 4.1.Value can be an integer, string, boolean, null, and a JSON object again. The second main message format is XML which is

Listing 4.1 JSON example

```
{"students":[
  { "firstName":"Bedirhan", "lastName":"Keskin", "number":3440094},
  { "firstName":"Zeynep", "lastName":"Keskin", "number":4050},
  { "firstName":"George", "lastName":"Vettel", "number":54321}
]}
```

a markup language as stated in its name. It is also human-readable but not easy to understand as a JSON object. XML definition requires an end tag which is not needed in a JSON objects. The same JSON object above can be defined in XML in this way: Since JSON is shorter, more popular,

Listing 4.2 XML example

```
<students>
  <student>
    <firstName>Bedirhan</firstName> <lastName>Keskin</lastName> <number>3440094</number>
  </student >
  <student >
    <firstName>Zeynep</firstName> <lastName>Keskin</lastName><number>4050</number>
  </student >
  <student >
    <firstName> George </firstName> <lastName>Vettel</lastName><number>54321</number>
  </student >
</students >
```

and quicker to read and write, it is a good choice when the application is not complex concerning data exchange. It is concluded to store data in JSON format because of its advantages over XML to reach to goals of this thesis.

4.1.5 Data Generator

As MetaDrive was selected as a Simulator component, an external Data Generator component is indispensable to support the system. MetaDrive primarily has a sensor suite for RL related tasks, such as a Camera, Lidar-like sensor, Gyroscope, GPS. However, a modern vehicle has various sensors which are examined in the Sensors in Modern Car section of Chapter 2 Theory and Fundamentals.

Creating a sensor model in high fidelity is an arduous task and there are several approaches as it is analyzed in Chapter 2 Theory and Fundamentals. One method is creating the mathematical model of the sensor and another method is machine learning and deep learning algorithms that can be utilized to understand sensor behavior.

Fundamental Design Decisions
Developing sensor models for a vehicle is not specifically one of the goals of this thesis. Thus, existing sensor and vehicle models were researched. MATLAB has numerous solutions for engineering problems particularly related to simulation and modeling. Simulink is a service of MATLAB for simulation, analyzing, and designing systems. Besides, users can access them with an academic license. Two different models in Simulink were evaluated for realizing the Data Generator component:

- Conventional Vehicle Reference Application
- Complete Vehicle Model

Conventional Vehicle Reference Application is under the Powertrain Blockset of MATLAB. It has four main segments inside it. These are environment, longitudinal driver, controllers, and passenger car

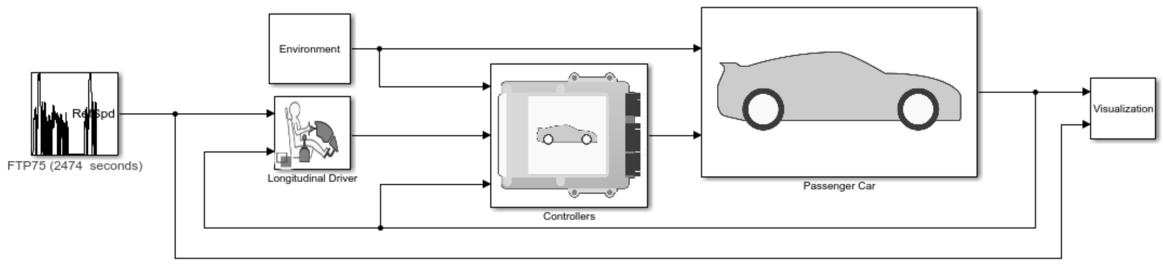


Figure 4.3: Conventional Vehicle Reference Application [MAT22d]

Its main area of usage is power and energy analysis. It consists of models for a combustion engine, powertrain, and transmission. The engine block under the Passenger car segment has an Engine module and Engine controller module. Spark ignition (petrol) and Compression-ignition (diesel) engines are available.

Compression-ignition engine provides data for:

- Power
- Air

- Fuel
- Temperature
- Efficiency
- Hydrocarbon emissions
- Carbon monoxide emissions
- Nitric oxide and nitrogen dioxide emissions
- Carbon dioxide emissions
- Particulate matter emissions

The disadvantage of the model is that it requires a substantial number of inputs. The input parameters are listed below [MAT22e]:

- Reference vehicle velocity
- Enable acceleration command override
- Acceleration override command
- Acceleration hold
- Disable acceleration command
- Enable deceleration command override
- Deceleration override command
- Deceleration hold
- Disable deceleration command
- Gear
- Longitudinal vehicle velocity
- Road grade angle

On the other hand, the Complete Vehicle Model requires just brake and throttle inputs to run. It is under the Simspace Driveline blockset. It comprises an engine, transmission, shift logic, torque converter, and vehicle body segments. The Engine model generates less data than Compression-ignition engine. Eleven data values of a vehicle can be generated:

- Speed
- Gear
- Engine power
- Engine fuel consumption
- Impeller speed
- Turbine torque
- Turbine speed

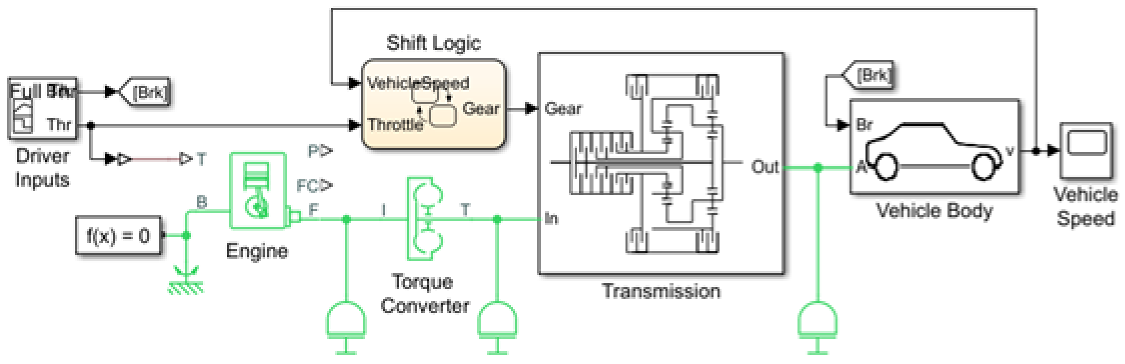


Figure 4.4: Complete Vehicle Model [MAT22c]

- Right tire slip
- Left tire slip
- Rear-axle normal force

The parameters below can be changed to simulate and generate diverse data for a vehicle. They are listed below with their standard values in the model:

- Road incline value = 0
- Wind value = 0
- Engine speed threshold = 100 RPM
- Engine max power=150000 RPM
- Engine max speed=6000 RPM
- Engine inertia=0.08 $kg*^2$
- Engine initial velocity = 960 RPM
- Engine fuel consumption constant = 25 milligram/rev Constant per revaluation
- Tire rated vertical load = 3000 Newton (N)
- Tire rolling radius=0.3 meter (m)
- Vehicle mass=1200 kilogram
- Vehicle front area=3 m^2
- Vehicle drag coefficient=0.4
- Brake drum radius=150 millimeter
- Brake friction coefficient = 0.01 $N * m/(rad/s)$

Since the Conventional Vehicle Reference Application is complex and demands more input than the Complete Vehicle Model, it is not the best choice for this thesis. Also, it requires more computing power which is a limitation. Further, the Complete Vehicle Model is equipped with sufficient data models. For these reasons, the Complete Vehicle Model is utilized to accomplish the Data Generator component. After this section, the Complete Vehicle Model is also referred to as Simulink Model.

4.2 Software System Model

The general perspective and functionalities of the system are described in this section using use case diagrams and sequence diagrams. The listed use cases below were specified to meet the requirements:

- Scenario Creation in GUI
- Vehicle Data Generation

4.2.1 Use Case: Scenario Creation in GUI

The user creates a scenario and configures parameters in the GUI. This use case is the only one in that the user interacts with Simulator for Connected Cars.

Partial Use Cases:

- Map selection
  - Map Creation
    - Save Map
  - Load Map
- Vehicles' initial value adjustment
- Vehicles' course configuration
- Vehicles' type configuration
  - MetaDrive vehicle configuration
  - Simulink model configuration
- Save Scenario
- Load Scenario

Scenario creation entails multiple sub steps. The user creates a new map or selects one default map. After that, the user creates a course for vehicles. There are two options for course definition. The first of them is the "Run&Create" mode. In this mode, the user drives a vehicle in the simulator using a keyboard. During the drive, vehicle behavior and commands are recorded and utilized for course creation. Another method is that the user can enter steering, throttle, and brake commands to text boxes in the GUI for any simulation steps. Vehicles apply these commands when simulation

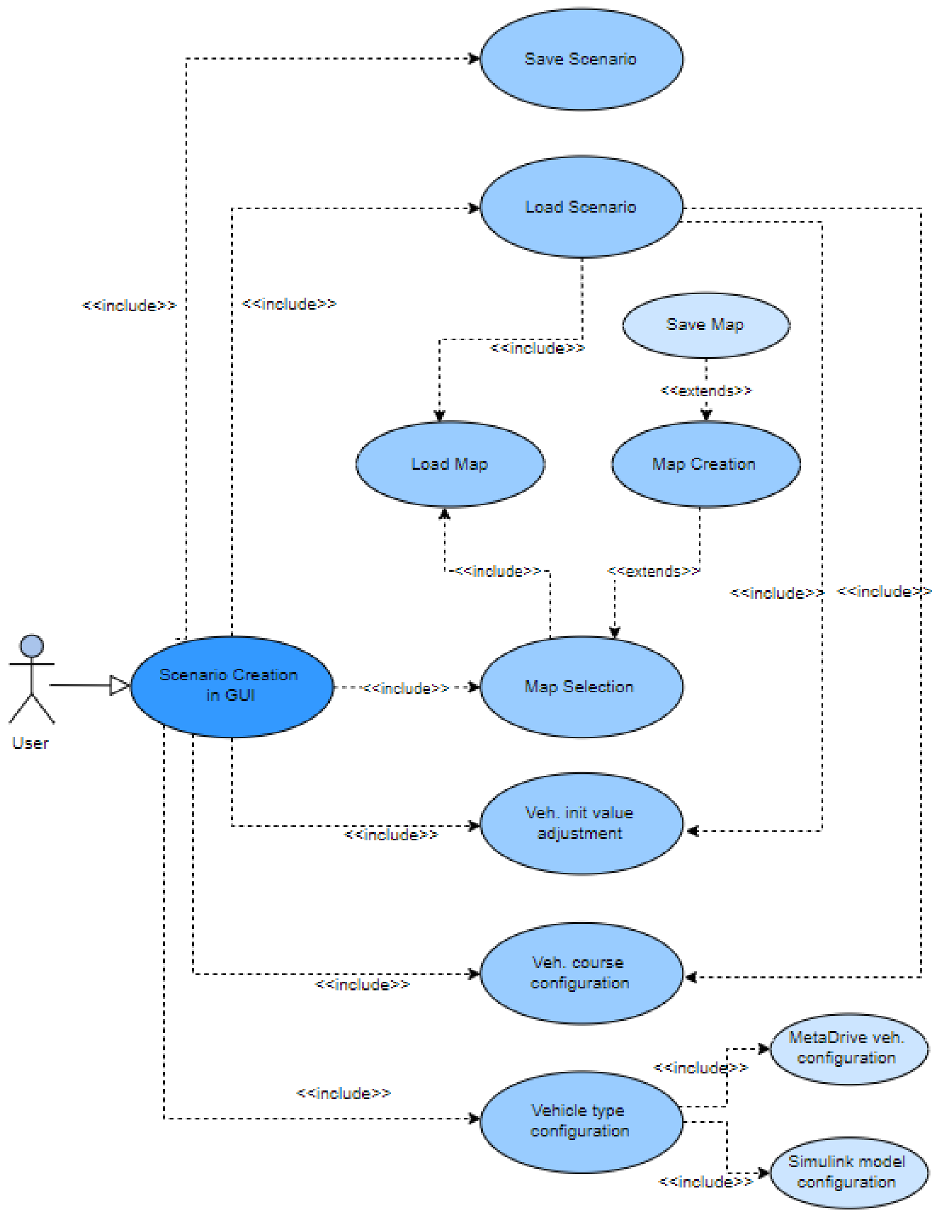

Save Scenario
Load Scenario
Save Map
Load Map
Map Creation
User
Scenario Creation in GUI
Map Selection
Veh. init value adjustment
Veh. course configuration
MetaDrive veh. configuration
Vehicle type configuration
Simulink model configuration
<<include>>
<<extends>>

Figure 4.5: Use Case: Scenario Creation in GUI

steps are reached. In this option, the user also sets vehicle initial values to define the heading, speed, and location values of vehicles. Lastly, the user sets up vehicle types and their parameters such as mass, brake friction coefficient. These values have an impact on the Data Generator Component.

Table 4.3: Use Case - Scenario Creation in GUI

USE CASE	Scenario Creation in GUI
Objective	Definition of the scenario by the user, configuration of parameters
Category	Primary
External Actors	User
Preconditions	No preconditions
Success End Condition	The simulation runs with desired parameters.
Failed End Condition	The simulation does not run, and the command window shows the error. If parameters are out of limits, show the dialog box before the simulation starts.
Trigger	The user opens the Graphical User Interface

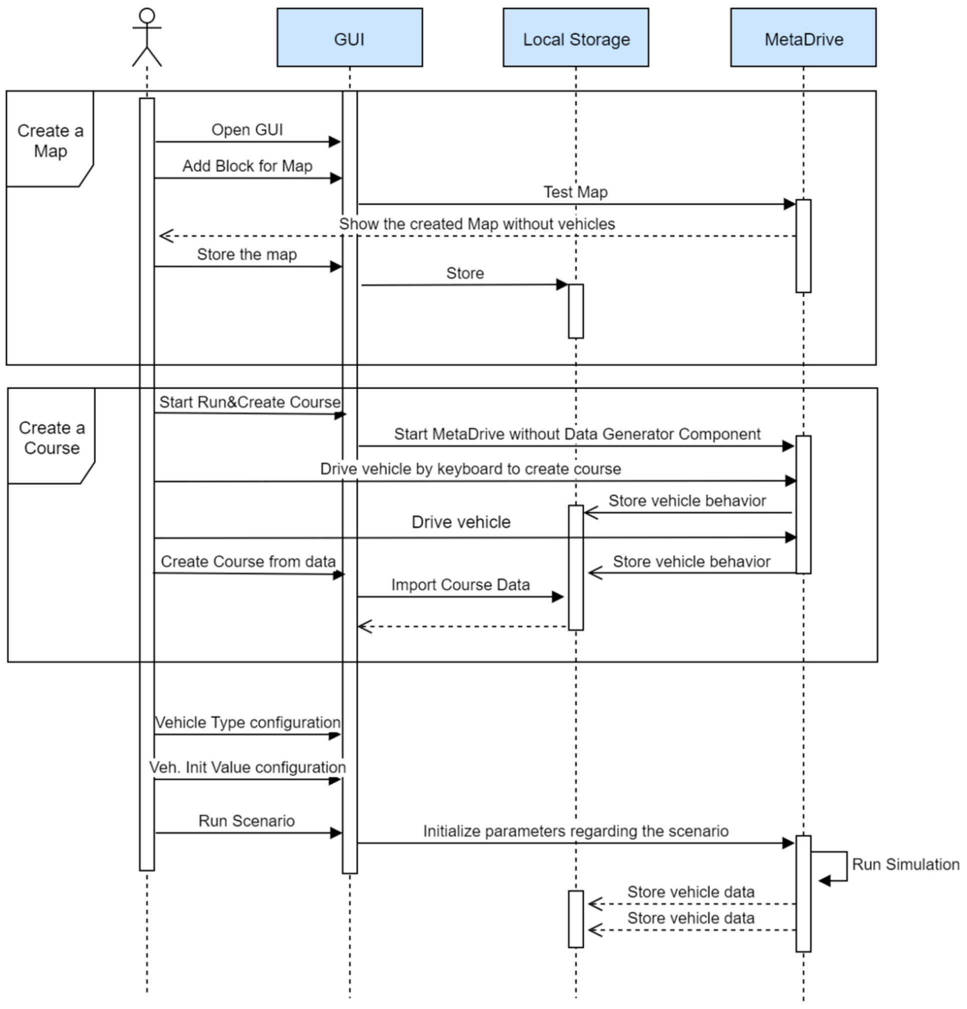


Figure 4.6: Sequence Diagram of Scenario Creation in GUI

4.2.2 Use Case: Vehicle Data Generation

Vehicle data generation is performed by two parts of the simulator. MetaDrive provides various sensor values for a vehicle and the Complete Vehicle Model in Simulink is responsible to produce data. Data is generated with respected vehicle configuration parameters and vehicle commands. MetaDrive which has an application programming interface in Python is the actor.

Partial Use Cases:

- Generate Data in MetaDrive
- Generate Data in Simulink Model
- Establish Communication
- Transfer Data/Command

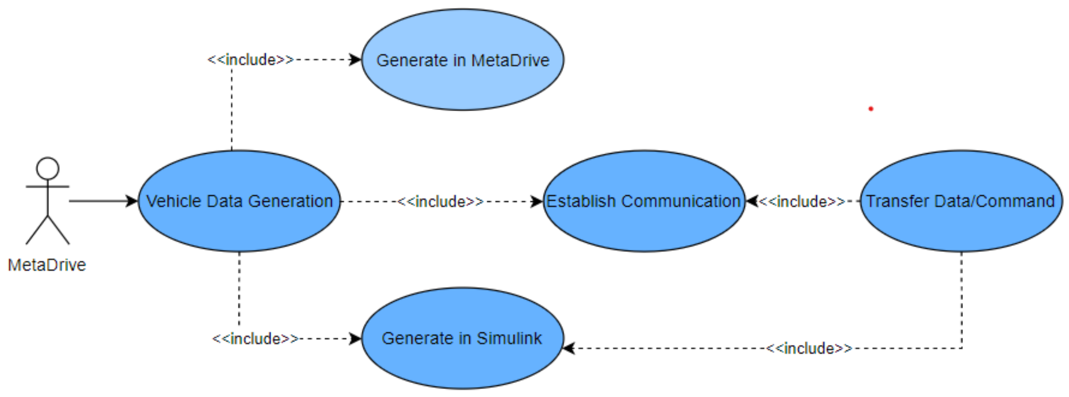


Figure 4.7: Use Case Vehicle Data Generation

In the sequence diagram below, vehicles' commands are read from the MetaDrive. However, it is not the only option for accessing commands. The vehicle Control component can provide commands from a remote server via RabbitMQ. Firstly the communication is established and it shall be tested before simulation is started. After that all process runs inside a loop. Generated data can be recorded after a message from the Simulink Model is received. This enables to store both data from MetaDrive and Simulink into a same JSON object.

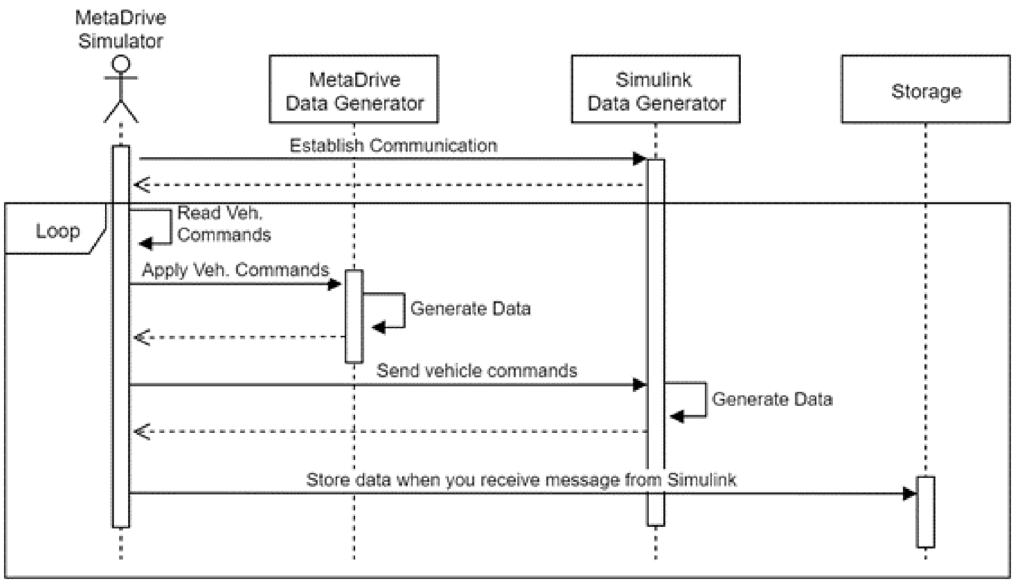


Figure 4.8: Sequence Diagram of Vehicle Data Generation

Table 4.4: Use Case – Vehicle Data Generation

USE CASE	Vehicle Data Generation
Objective	Data for each vehicle in the simulation is generated.
Category	Primary
External Actors	MetaDrive
Preconditions	The connection between MetaDrive and Simulink is established.
Success End Condition	Data from both Simulink and MetaDrive are generated.
Failed End Condition	MATLAB dialog box shows the exception when there is an error in the Simulink Model. The user sees the error and exception in the command windows when there is an error in MetaDrive
Trigger	GUI runs the simulation.

5 Implementation

This chapter covers the realization of system components with respect to the definition in Chapter 4 Simulation System Concept. The implementation details of each component are explained in the subsections.

5.1 Graphical User Interface

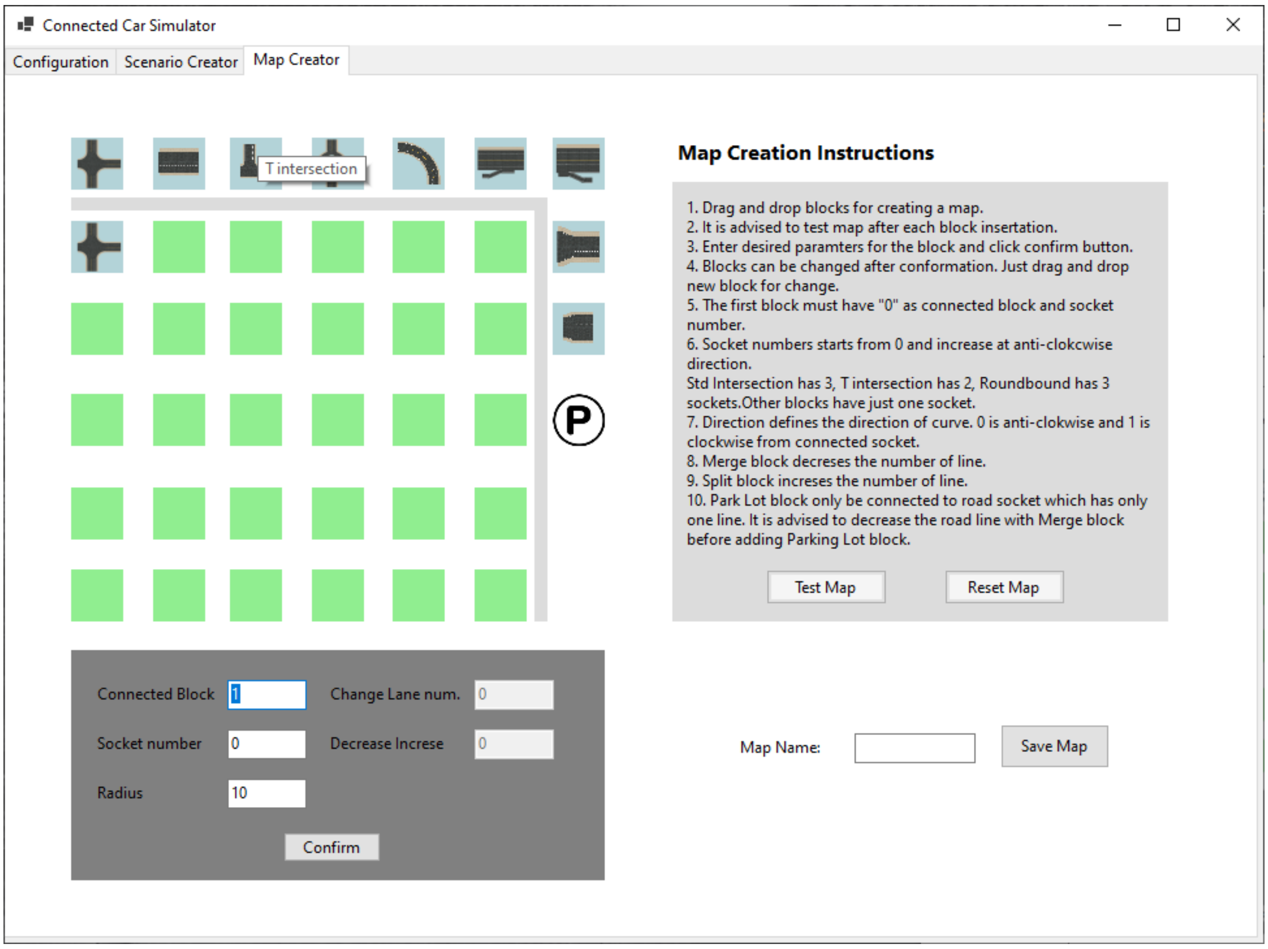
The Graphical User Interface must provide configuration options for every component of the simulator as the user creates and configures scenarios only using the GUI. For this reason, three sections inside the GUI were defined:

- Configuration: This section is responsible for the vehicle parameter configuration of MetaDrive and Simulink Model. In addition to that, the user can determine which data value is being recorded during the simulation. The user can control the data transmission rate between Simulink Model and MetaDrive and storage options.
- Scenario Creator: The user can create their scenario by selecting a map, entering vehicles' initial values, and defining a course.
- Map Creator: It is possible to produce and test custom maps in this section. These maps are selected in Scenario Creation.

5.1.1 Map Creator

This section enables the user to create their custom map using ten different roadblocks. Roadblocks are utilized to define a map in MetaDrive. Since MetaDrive was developed for RL Tasks, there is limited support for forming a map. The user can just define the roadblock type in Python API but not their direction or values, such as the length. However, this section in the GUI gives the user opportunity to decide every parameter of a roadblock. Ten roadblocks are located around green squares. The green ones show empty roadblocks. The user can generate a map with up to thirty roadblocks. The user identifies a type of roadblock when hovering the mouse over it, such as the "T intersection" in Figure 5.1. The user drags and drops a roadblock on any green square to add this block to the map. After that, the dark grey section appears for setting the parameters of the roadblock. Each roadblock has distinct parameters. Roadblocks and their parameters are listed below:

- STD intersection
  - Radius



Map Creation Instructions

1. Drag and drop blocks for creating a map.
2. It is advised to test map after each block insertion.
3. Enter desired paramters for the block and click confirm button.
4. Blocks can be changed after conformation. Just drag and drop new block for change.
5. The first block must have "0" as connected block and socket number.
6. Socket numbers starts from 0 and increase at anti-clokcwise direction.
Std Intersection has 3, T intersection has 2, Roundbound has 3 sockets.Other blocks have just one socket.
7. Direction defines the direction of curve. 0 is anti-clokwise and 1 is clockwise from connected socket.
8. Merge block decreses the number of line.
9. Split block increses the number of line.
10. Park Lot block only be connected to road socket which has only one line. It is advised to decrease the road line with Merge block before adding Parking Lot block.

Connected Block	1	Change Lane num.	0
Socket number	0	Decrease Increse	0
Radius	10		

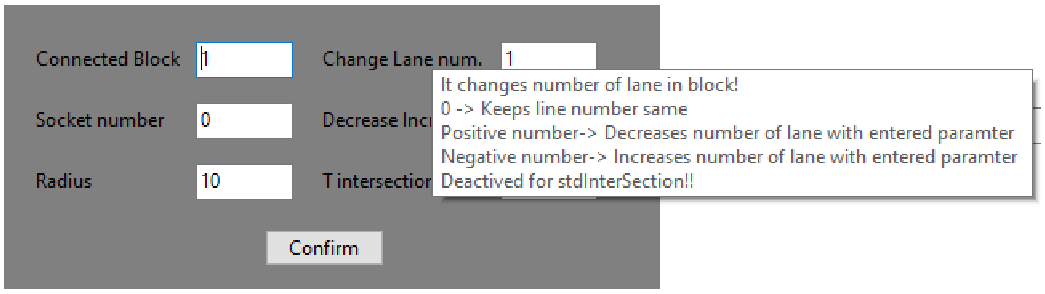
Map Name:

Figure 5.1: Map Creator Section in the GUI

- Straight
  - Length
- Roundabout
  - Radius exit
  - Radius inner
  - Angle
- Curve
  - Radius
  - Direction
  - Angle
  - Length
- In ramp

  - Length
- Out ramp
  - Length
- Merge
  - Length
- Split
  - Length
- Parking Lot

The user can access more information related to parameters by hovering the mouse on parameters as shown in Figure 5.2 and in the Map Creation Instructions section. The user needs to click on the



It changes number of lane in block!
0 -> Keeps line number same
Positive number-> Decreases number of lane with entered paramter
Negative number-> Increases number of lane with entered paramter
Deactived for stdInterSection!!

Figure 5.2: Tooltip for a parameter of T intersection

"Confirm" button to add the block after setting parameters. The next block can be appended in a similar way, but the user must enter the value for the "Connected Block" and the "Socket Number". These two parameters are required because multiple roadblocks can be connected to one single block. For instance, the STD intersection block has three different sockets which are numbered anti-clockwise as shown in Figure 5.3. This means that any three roadblocks can be linked to this STD roadblock at the same time. The socket which is marked with "Start" is the connection point of the block with the previous roadblock. The previous block is defined via "Connected Block" parameter. When a roadblock is created, the user can observe the parameter values and the number of the block using the tooltip by hovering the cursor over the block. Figure 5.4 shows the parameters of the Curve block. The user can test a map in each step after adding a roadblock via the "Test Map" button. This helps the user to make changes in the map in every step because the user can adjust the created roadblock with any other block. The user follows the same procedure in the same way by adding new roadblocks. For example, it is possible to drag and drop a Roundbound block on the existing Curve block in Figure 5.4.

The "TestMap" button generates Python code in "[MetaDrive Directory]/connectedCars/test_map.py" considering the map configuration. Then it runs the code using Windows Command Line that launches the created map in the MetaDrive environment. MetaDrive directory can be set in the Configuration section of the GUI. The test map does not show any vehicle and object because its only purpose is that the user can inspect the map. Figure 5.5 shows the created map according to the configuration in Figure 5.4. It consists of STD intersection and Curve blocks. Most of the map

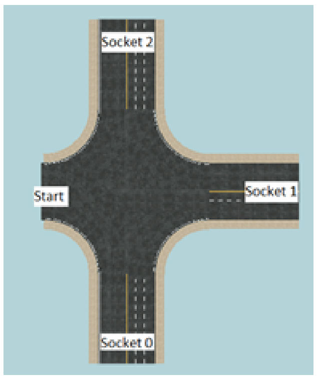
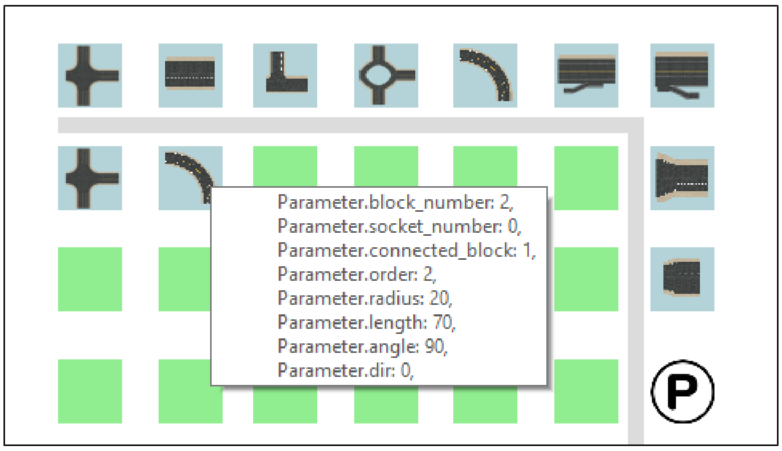


Figure 5.3: Sockets of STD intersection



Parameter.block_number: 2,
Parameter.socket_number: 0,
Parameter.connected_block: 1,
Parameter.order: 2,
Parameter.radius: 20,
Parameter.length: 70,
Parameter.angle: 90,
Parameter.dir: 0,

Figure 5.4: Tooltip for Curve block

contains more than two blocks. This example was kept simple to provide easier comprehension. "Reset Map" which is seen in Figure 5.1 clears created blocks on the GUI and the user can start

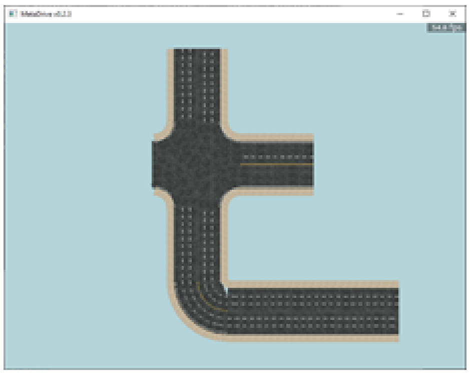


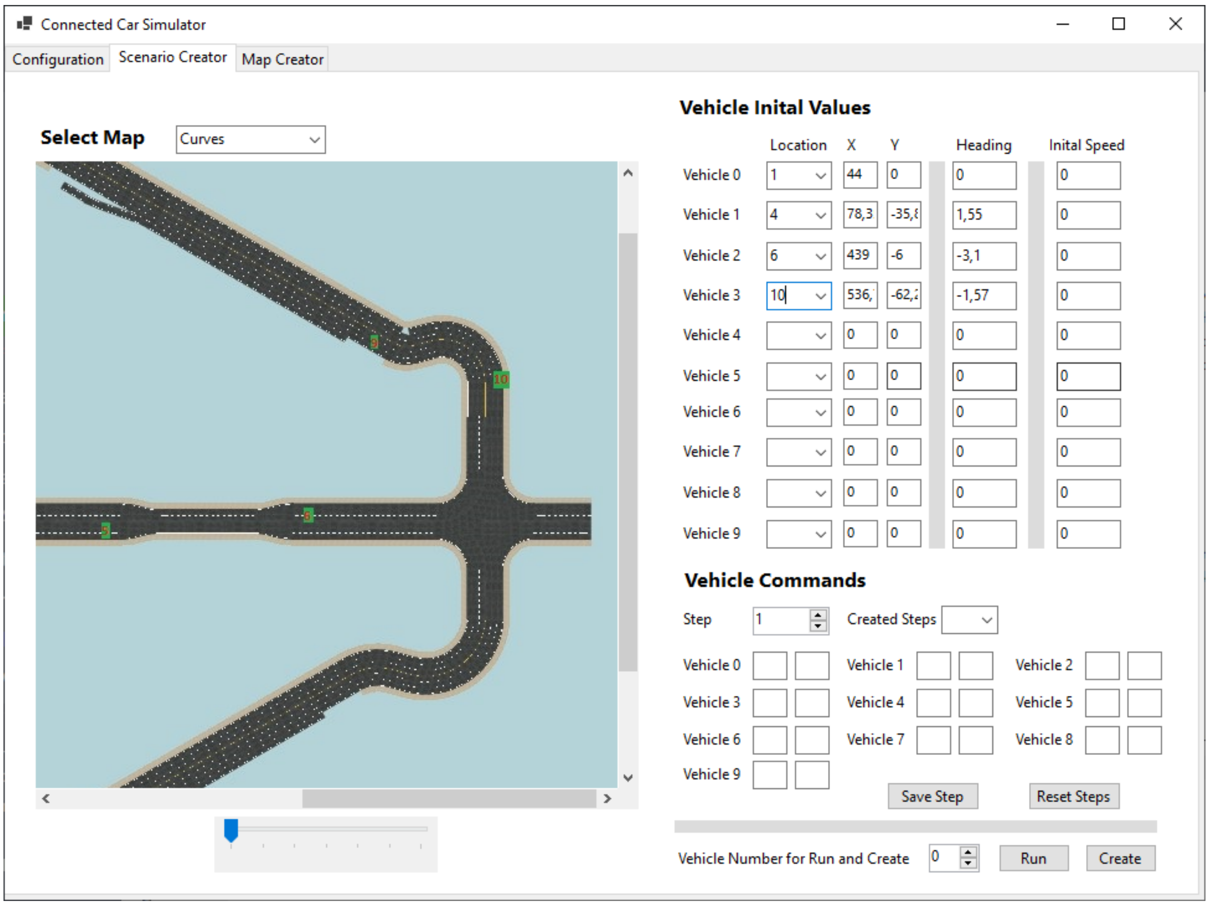
Figure 5.5: Test of Created Map

designing a map from the beginning. The user is obligated to save the custom map by entering a name and clicking the "Save Map" button to use it in the Scenario Creator section of GUI. This button generates a Python file located at "[MetaDrive Directory]/connectedCars/Maps/[Map

Name].py”. Generated Python codes from clicking “Test Map” and “Save Map” are not the same but have similarities to define blocks. The user finds the fundamental information to create a map from the “Map Creation Instructions” section on the right side of the GUI.

5.1.2 Scenario Creator

The Scenario Creator section in the GUI is where the user selects the map, locates the cars, and defines their courses. It is displayed in Figure 5.6. Firstly, the user selects the map from the combo



Select Map: Curves

Vehicle Inital Values

	Location	X	Y	Heading	Inital Speed
Vehicle 0	1	44	0	0	0
Vehicle 1	4	78,3	-35,8	1,55	0
Vehicle 2	6	439	-6	-3,1	0
Vehicle 3	10	536,	-62,2	-1,57	0
Vehicle 4		0	0	0	0
Vehicle 5		0	0	0	0
Vehicle 6		0	0	0	0
Vehicle 7		0	0	0	0
Vehicle 8		0	0	0	0
Vehicle 9		0	0	0	0

Vehicle Commands

Step: 1   Created Steps:

Vehicle 0   Vehicle 1   Vehicle 2
Vehicle 3   Vehicle 4   Vehicle 5
Vehicle 6   Vehicle 7   Vehicle 8
Vehicle 9

Vehicle Number for Run and Create: 0

Figure 5.6: Scenario Creation Section of GUI

box near “Select Map” which consists of three default maps, as seen in Figure 5.7. Custom maps are added to the combo box when the user saves them in the Map Creator section. Each default map has a green-colored location marker that can be used to define the locations of vehicles. The user may select these locations from combo boxes under “Location” at the “Vehicle Initial Values” part. When the user selects a number for a location, text boxes under X, Y, and Heading are set regarding a marker. The user does not have to choose a location from combo boxes, they can enter values directly to textboxes for X, Y, Heading, and Initial Speed. These four values determine the initial behavior of the vehicles that is applied in the first simulation step. The user can form a course for vehicles in the “Vehicle Commands” part. There are two ways to specify a vehicle course. The first of them is that the user can enter the values for the steering wheel and the pedal value (throttle and brake) for any step number of a scenario. The steering wheel value can range between -1 and 1 where negative values represent the right direction and positive values are utilized for the left direction. A pedal value has the same limits. Negative values represent the brake and positive values are used for the throttle. This information is provided to users as a tooltip that they can spot when they hover the mouse over boxes. After entering values, the user needs to click on the "Save Step" button. In this method, the user can make fine adjustments for a course, but it is often time-consuming and hectic.

For this reason, the second method to define a course was introduced which is an effortless option. It is called Run&Create. The user selects the vehicle number near "Vehicle Number for Run and Create" and clicks on the "Run" button This opens the MetaDrive Interface and locates vehicles

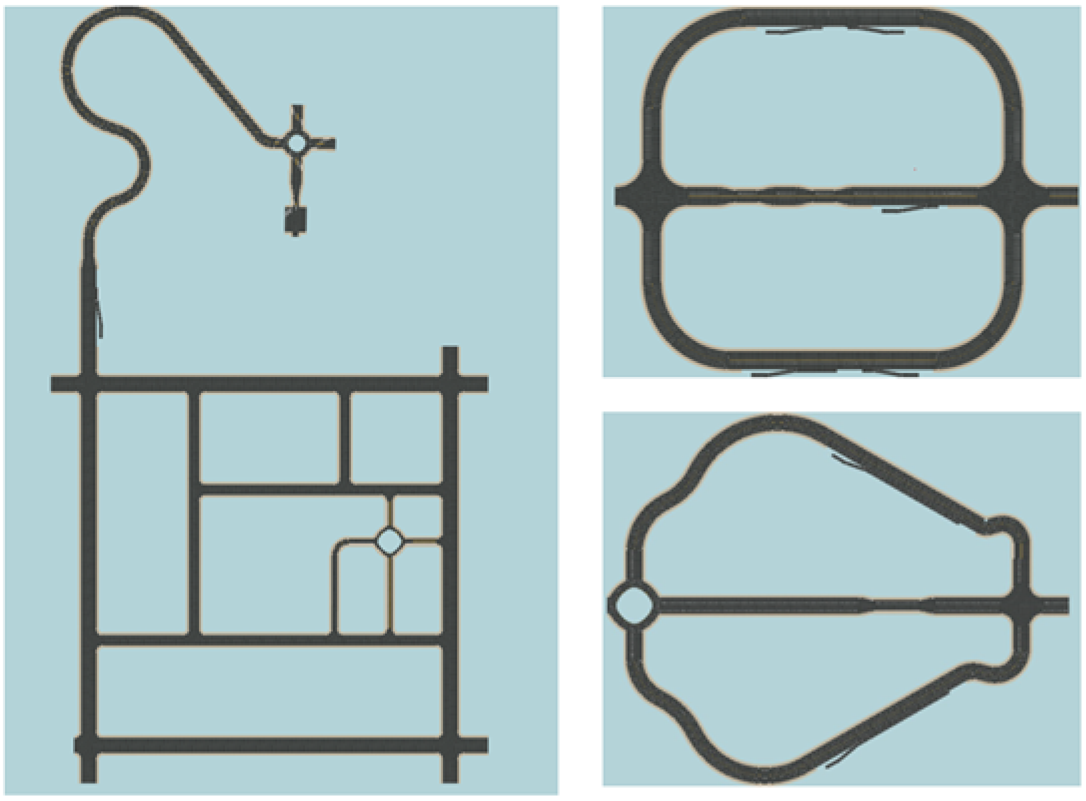


Figure 5.7: Default Maps: Left Big City - Right Top Highway - Right Bottom Curves

regarding the values under "Vehicle Initial Values". After that, the user drives the selected vehicle via keyboard, and commands for a vehicle are stored at "[MetaDrive Directory]/vehicle_steps.csv". The generated file contains three integers separated by commas in every line. The first integer shows the step number, the second shows the steering value, and the third shows the pedal value. The earlier creation approach allows the user to define a course with decimal numbers. However, Run&Create just uses integers. If there are other vehicles which already have courses, they move with respect to their courses during the simulation run for the Run&Create option. This feature enables the user can create a scenario regarding other vehicles' behavior. After the user drives a vehicle for creating a course, they must close the MetaDrive environment and click on the "Create" button on the GUI. This reads the “vehicle_steps.csv” file and imports the values for a vehicle. The user can view each command for vehicles by selecting “Created Step” under “Vehicle Commands” and can make changes to the values. The “Reset Steps” button deletes all steps for every car.

5.1.3 Configuration

This part oversees car parameters arrangement in MetaDrive and Simulink Model. Furthermore, the user chooses which data value is captured during the simulation and specifies settings related to the GUI itself and the simulator. This tab of the GUI is demonstrated in Figure 4.2. Firstly, the user picks the total number of vehicles for simulation. It creates new vehicles in the combo box under the “Number of Vehicles” text. The users choose the vehicle that they want to configure. Each vehicle can have distinctive characteristics. There are two main sections for a vehicle characteristic configuration.

The first of them is “MetaDrive Vehicle Parameters” which sets parameters used in the MetaDrive and mainly affects vehicles behavior. The user can select the vehicle model which adjusts thirteen parameters and the vehicle shape in the simulator. When the user opts for any of the vehicle types, the text boxes are set by default values of the vehicle type. Nevertheless, the user can alter them. There are five vehicle types:

- Ferra – Default
- Truck – XL
- Lada – L
- 130 – M
- Bettle – S

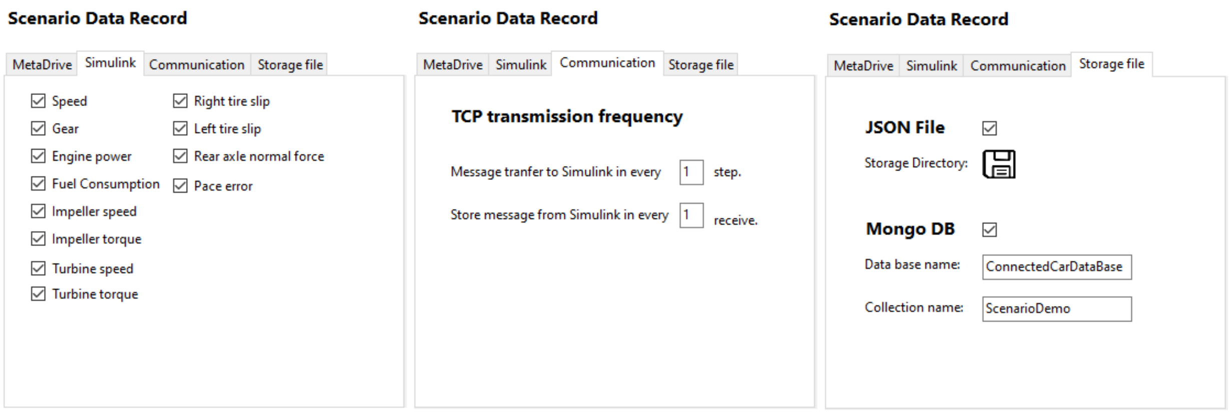
The second main section for vehicle configuration is “Simulink Vehicle Configuration” which has fourteen values. These values are also specific for each car. Simulink Model generates data regarding these parameters. Configuration parameters are listed below:

- MetaDrive: Length, width, mass, height, tire radius, tire width, rear wheelbase, front wheelbase, lateral tire to center, chassis to wheel axis, max speed, wheel friction, max steering
- Simulink: Road incline, wind, engine speed threshold, engine max threshold, engine inertia, engine initial velocity, tire rated vertical load, tire rolling radius, mass, vehicle front area, vehicle drag coefficient, brake drum radius, brake friction coefficient

The user concludes which values are recorded during the simulation under the “Scenario Data Record” section thanks to checkboxes. Besides, the user modifies settings related to communication. There are four tabs in this section:

- MetaDrive: Seventeen various data generated by MetaDrive are listed in the tab which can be seen in Figure 4.2
- Simulink: The Simulink Model provides twelve different data as seen in Figure 5.8

- Communication: The user adjusts communication rate-related values. The user determines how often the MetaDrive transmits a message to Simulink for vehicle commands. In addition, the user sets the value for whether each received message from the Simulink is stored or not.
- Storage file: The user can decide where to store data using checkboxes. There are two options which are JSON File and Mongo DB. The storage directory can be changed by clicking on the disk icon as shown in Figure 5.8. This tab enables the user to make adjustments to Mongo DB which details are explained in Section 5.5 Data Record.


Scenario Data Record

MetaDrive | Simulink | Communication | Storage file

- [✓] Speed
- [✓] Gear
- [✓] Engine power
- [✓] Fuel Consumption
- [✓] Impeller speed
- [✓] Impeller torque
- [✓] Turbine speed
- [✓] Turbine torque
- [✓] Right tire slip
- [✓] Left tire slip
- [✓] Rear axle normal force
- [✓] Pace error

Scenario Data Record

MetaDrive | Simulink | Communication | Storage file

TCP transmission frequency

Message tranfer to Simulink in every 1 step.

Store message from Simulink in every 1 receive.

Scenario Data Record

MetaDrive | Simulink | Communication | Storage file

JSON File [✓]

Storage Directory:

Mongo DB [✓]

Data base name: ConnectedCarDataBase

Collection name: ScenarioDemo

Figure 5.8: Scenario Data Record Section

The Configuration section has a feature to save a scenario. After users configure all settings, they enter a scenario name and save it. This presents great convenience for the users because it is not required to deal with all parameters in every GUI opening. Custom scenarios are added to the combo box near “Select Scenario” after saving them. Four default scenarios can be selected using the combo box:

- Curves – Long
- RoundBound
- RoundBound – Crash
- Highway-Crash Scenario

Details and purposes of scenarios are discussed in further chapters. The scenario can be put into a loop by checking the box near “Loop”. Otherwise, MetaDrive closes itself after a scenario is completed.

5.2 Simulator

The Simulator component is realized using MetaDrive because of the motivations discussed in Section 4.1.2 Simulator. The simulator component provides the main functionality for meeting the requirements. It has a connection with other components. Firstly, MetaDrive architecture is explained for providing base information for changes in MetaDrive to adapt it as a Simulator for Connected Cars.

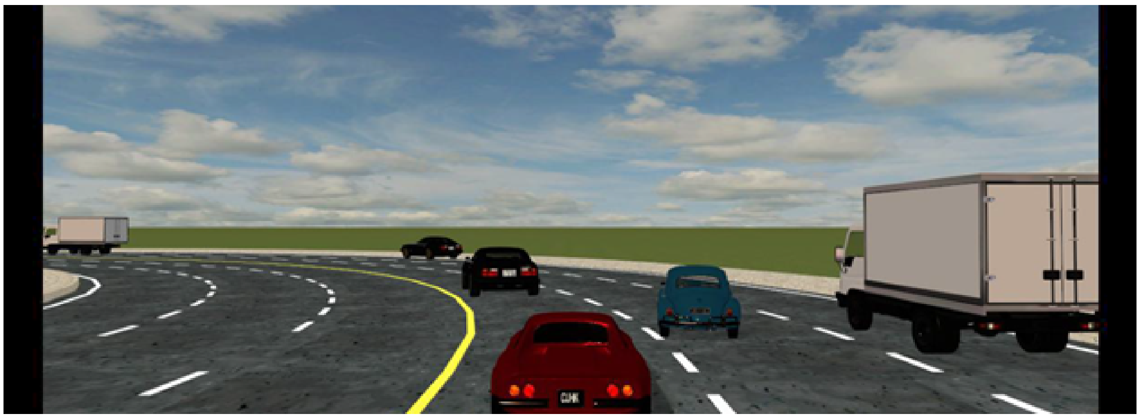


Figure 5.9: MetaDrive [Eng22]

5.2.1 MetaDrive Architecture

MetaDrive has a Python API that abstracts back-end engines for rendering and physics from the high-level classes [Eng22]. It utilizes Panda3D for the rendering engine and Bullet Engine for the physics engine. MetaDrive defines an Object, which is fundamental for the existence of e.g., a vehicle or roadblock. Object manages the connection between API and back-end engines. Hence, a developer does not need to deal with calculations and code changes in engines. In addition to that, the parameter space of engines can be controlled from the API. The configuration of a vehicle in the GUI affects these back-end engines.

Policy is another main component of MetaDrive that evaluates the environment and other objects in order to decide the upcoming behavior of objects. MetaDrive supports traffic follow via this component and manages vehicles' actions. However, it is not robust, and vehicles can crash each other.

Manager is the last main component of MetaDrive. It is responsible for objects and their various roles. There are multiple managers for specific tasks, such as Map Manager, Agent Manager, Obstacle Manager, and Traffic Manager. For example, the Agent Manager oversees respawn of a vehicle when it is terminated, and a map manager is in charge of creating random maps for RL tasks. Manipulation in these components is required to extend MetaDrive features.

MetaDrive has environment classes that control the interaction of objects among each other and the definition of managers. Safe MetaDrive Environment is for controlling one vehicle by the users or their algorithm. Other vehicles are controlled by Traffic Manager which is a ruled-based algorithm for traffic flow. Multi Agent MetaDrive Environment is another class. It consists of multiple vehicles that the user can control. This environment is equipped with more features to realize the Simulator Component. There is no limit for vehicles in MetaDrive, but each vehicle requires more computing resources and it increases the number of models in the Simulink. For this reason, the maximum number of vehicles is determined as ten. It is enough for creating complex connected car scenarios. MetaDrive requires assigning values to parameters of the environment. For instance, one of the parameters is the number of vehicles.

5.2.2 MetaDrive and GUI Interaction

The connection between MetaDrive and the GUI is provided in a very simple and efficient way. The “scenarioConfigurationTemplate.py” file is located in the MetaDrive directory. The GUI reads it and its text is manipulated regarding the user scenario definition via the GUI. Then it is written to the same location as the “scenarioConfiguration.py” file. This Python file consists of multiple variables in miscellaneous data types. Mostly dictionaries and lists are preferred. This Python file contains all information about the configuration of a scenario. The table below exhibits variables in scenarioConfiguration.py.

Table 5.1: Variables in scenarioConfiguration.py

Name	Usage	Data Type	Number of Element	Affected Python file
simulink_ configration_JSON	Configuration of Simulink Model. This dictionary is transmitted to Simulink Model.	Dictionary	10 Dictionaries which have 16 elements each	connected_cars _sim.py
TotalVehicleNumber	Definition of number of vehicles in a scenario	Integer	1	connected_cars _sim.py
vehicle_model_list	Definition of vehicle models such as Default, XL, L, M, and S	List	10	connected_cars _sim.py
drive_and_cmd_create_	Activates RunCreate and deactivates Simulink Model when it is True	Boolean	1	connected_cars _sim.py
drive_and_cmd_ vehicle_text_	Sets which vehicle is selected for RunCreate	String	1	connected_cars _sim.py
vehicle_Model_ configration_dic	Defines characteristics of each vehicle model	Dictionary	5 Dictionaries which have 13 elements each	vehicle_type.py

data_storage_configration_dic	Shows which data is stored during the simulation	Dictionary	29	connected_cars _sim.py
data_storage_ transmission_settings	Defines where and how data is stored	Dictionary	8	connected_cars _sim.py
vehicles_initial_values	Determines vehicles' initial positions and velocity	Dictionary	10 Dictionaries which have 4 elements each	connected_cars _sim.py
vehicles_commands_list	Shows vehicle commands for scenario steps	List	Depends on the scenario length	connected_cars _sim.py

Variables defined above are mostly passed to methods and consumed in "connected_cars_sim.py". The code in "connected_cars_sim.py" assigns variables to parameters of methods or classes that are defined in other Python files. All parameters are written to the file before a simulation starts. This means that interaction between GUI and MetaDrive is terminated after the configuration of parameters and before the MetaDrive interface is started. The user can access vehicle data and control vehicles during a simulation via the Vehicle Control component which is explained in Section 5.4.

5.2.3 MetaDrive Map Definition

The map of the scenario is defined in the '[MetaDrive Directory]/connectedCars/Maps/map_creator.py'. When the user selects a map from the GUI and starts a scenario, the selected map is copied with the map_creator.py name. It contains **createmap** function which specifies the map. Each roadblock is an object of a class provided by MetaDrive. For instance,

cblock1 = Roundabout([Block Number], [Connected Block].get_socket([Socket Number]), global_network, 1)

The code snippet above shows how to build a roadblock object. It requires a block number, connected block name, and socket number which are decided in the GUI. After the object is created, properties of the block are specified in the **construct_from_config** method of the object. A created block is appended to the **blocks** property of a self which refers to the BIG class object. It is defined in the "BIG.py" file. BIG is the algorithm that generates a random map in MetaDrive. However, it was adjusted to use the **createmap** function instead of arbitrary map generation. The **create_map_from_GUI** flag must stay true to realize user-defined maps. Lane number of a roadblock is fixed to the value three in the BIG class.

5.2.4 Vehicle Characterization and Data Generation

As mentioned in earlier chapters, MetaDrive generates vehicle data but it is limited. Because the main aim of MetaDrive is to provide a simulator for RL related tasks. Thus, there is an extra Data Generator component in our system architecture. However, this section just explains the data generated by MetaDrive itself. The generated data depends on the vehicle characterization. Users set variables for a vehicle thanks to our GUI and the variables are stored in a dictionary named **vehicle_Model_configration_dic** in the "scenarioConfiguration.py" file. This dictionary contains five dictionaries for five different vehicle types. A dictionary for a specific car model is imported to the "vehicle_type.py" file. There are five different classes for each vehicle type and values in dictionaries are consumed in these classes.

The data is generated by the physics engine and MetaDrive's algorithm. MetaDrive has an environment class which is explained in the MetaDrive Architecture section. Vehicles are objects in the environment class and generated data for each vehicle are properties of a vehicle object. **create_agents_data** function in the "connected_cars_sim.py" file is responsible for accessing generated data and transferring them to a dictionary. **ConnectedCarsMetaDrive** Environment class in the "connected_cars_metadrive.py" file was introduced to present extended features for the thesis. MetaDrive generates random maps and defines paths for vehicles in its default code. When the vehicles reach the final destination, they are destroyed and respawn from initial positions, or a scenario is completely terminated. However, this is the total opposite of the requirements of the thesis. Thus, the termination of vehicles at the end and behaviors of vehicles regarding the default path rules were changed in the **ConnectedCarsMetaDrive Environment**. Now, vehicles are not terminated when they reach the destination and they do not follow the navigation points to reach destinations. In addition to that, the **ComplexObjectManager** class was added to insert objects and homes on the maps. The positioning of objects and homes is not supported by our GUI and the user cannot activate this feature from the GUI. Nevertheless, the user can examine the Big City map with the object option by uncommenting the class. It provides traffic warning signs, traffic cones, accidents of vehicles, and houses on the map. Houses on the map were not supported by the MetaDrive. It is a new feature implemented in this thesis. Objects, such as homes, traffic cones, and other vehicles can be detected by the Lidar-like sensor and captured by the camera. The **ConnectedCarsMetaDrive** class has **MetaDriveEnv** as a superclass and it requires **BaseEnv** as a superclass. BaseEnv abstract engines from the MetaDrive Python API. It has OpenAI Gym class as a superclass and it represents an environment with unpredictable behind-the-scenes dynamics. These classes play the main role in data generation.

The environment has steps that are provided by the OpenAI Gym class. The properties of objects are updated, and commands of vehicles are applied in every step. The step length is 0.02 seconds which can be changed. It is defined by MetaDrive creators, and the same value is utilized for this thesis. This means MetaDrive generates new data every 0.02 seconds.

5.3 Data Generator

The Data Generator Component contains two main parts. These are MATLAB script and Simulink Model. MATLAB script is responsible for the configuration of the Simulink Model and triggers the start of the Simulink Model. Simulink Model generates the vehicle data. Complete Vehicle

Model was selected to implement the Data Generator. It is a Simulink Model that is developed just for a vehicle. Nevertheless, the Data Generator has to operate up to ten vehicles. Thus, the model was extended to ten vehicles as shown in Figure 5.10. This figure just provides a grand schema for easier comprehension. The subsections of Figure 5.10 are discussed, and functionalities are described in upcoming paragraphs for every part of the Figure. The default Complete Vehicle Model in Figure 4.4 has no feature for interaction with any external component and brake/throttle values depend on the Signal Builder block which is defined before the Simulink simulation is started. However, vehicle brake/throttle commands must be received from the Simulator block. Besides, the Data Generator transmits produced data to the Simulator component. This requires bidirectional communication. Another requirement is that vehicles must be configured regarding input from the GUI. The vehicles' configuration data are kept in **simulink_configration_JSON** located in the "scenarioConfigurator.py" file as explained in the earlier chapter.

The Simulator Component initiates the Data Generator Component. The structure and interface between the two components are clarified in Figure 5.11 using a sequence diagram. The MATLAB script is called by a Python script of the Simulator Component. The MATLAB Engine API is a Python module that allows developers to use MATLAB from their code [MAT22b]. The MATLAB Engine is defined inside a new thread which provides a concurrency inside the Python Code because the start of MATLAB can take time and this thesis aims to run both MATLAB script and Python script at the same time. When the MATLAB engine is initiated, Python scripts run the MATLAB script code developed for this thesis.

Before the simulation is started, the Data Generator must be ready and configured. Thus, the Simulator component creates a connection with the MATLAB Script after starting the MATLAB Engine. Firstly, the Simulator component initializes its User Datagram Protocol (UDP) sockets for the connection and the MATLAB script does the same, as shown in Figure 5.11. After the initialization, Simulator Components transmits the 'TestMessage' string to the MATLAB script. The MATLAB part tries to catch the message in for-loop up to 50 times with a one-second delay between UDP read commands. If it fails, the user receives information about unsuccessful UDP communication. When bidirectional communication is established, the Simulator component transmits JSON object which is created from the **simulink_configration_JSON** dictionary.

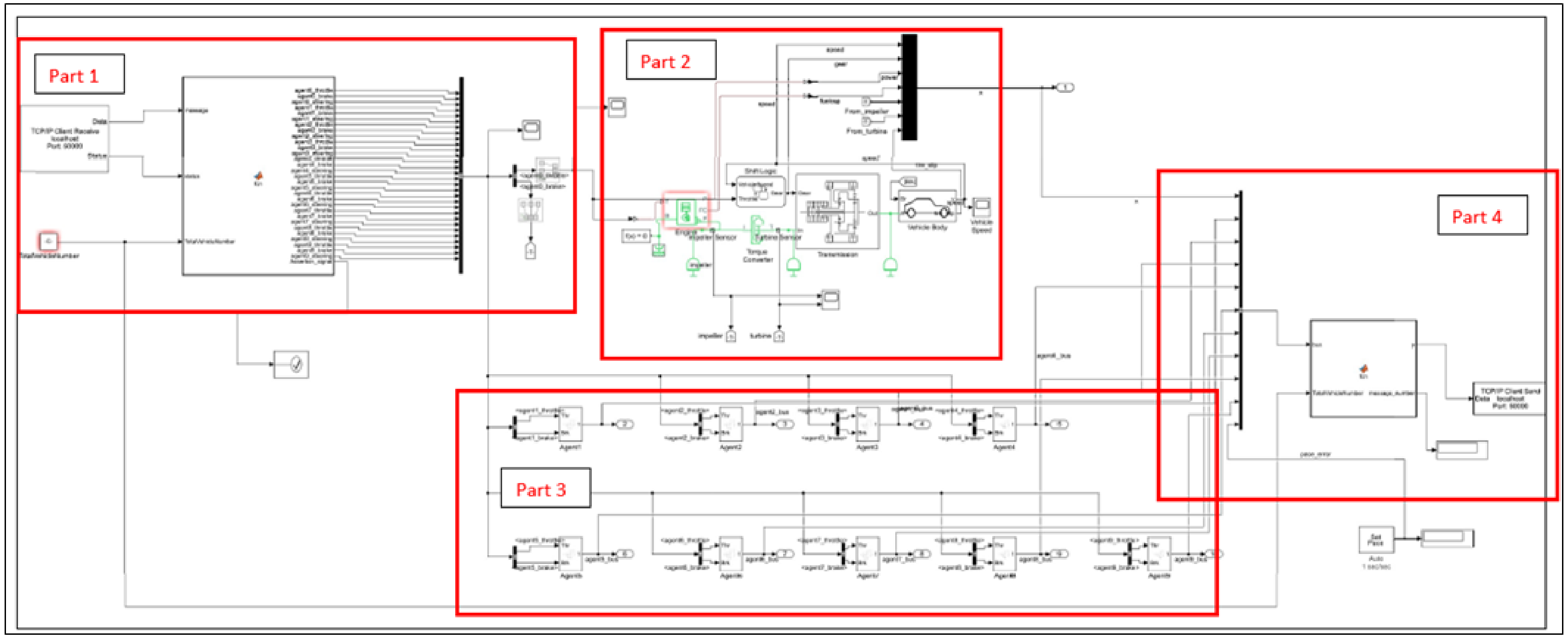


Figure 5.10: Simulink Model

The transmitted JSON object contains ten JSON objects that represent ten vehicle configurations. The key/value pair of an object is shown in Figure 4.2 under Simulink Vehicle Parameters. The message is always sent for ten vehicles but the Matlab Script only processes the message section for simulated vehicles. For example, if there are only five vehicles in a simulation, the MATLAB Scripts only reads the first five objects inside the JSON message and assigns its values to variables. These variables are directly used in the Simulink Model to configure blocks of the model. When the transmitted JSON object is handled in the MATLAB Script, the Simulator Component is informed with a UDP message. After sending the message, the MATLAB Scripts loads the Simulink Model and updates the configuration values of vehicles in the model. At the same time, the Simulator Component initializes Transmission Control Protocol (TCP) sockets for creating communication to Simulink Model. The TCP connection is utilized for sending brake/throttle commands to the Simulink Model. It provides that the message order is preserved. Since hundreds of messages can be sent between the Simulink Model and Simulator Component, message integrity and order must be granted. Thus, TCP is an optimum choice to realize communication.

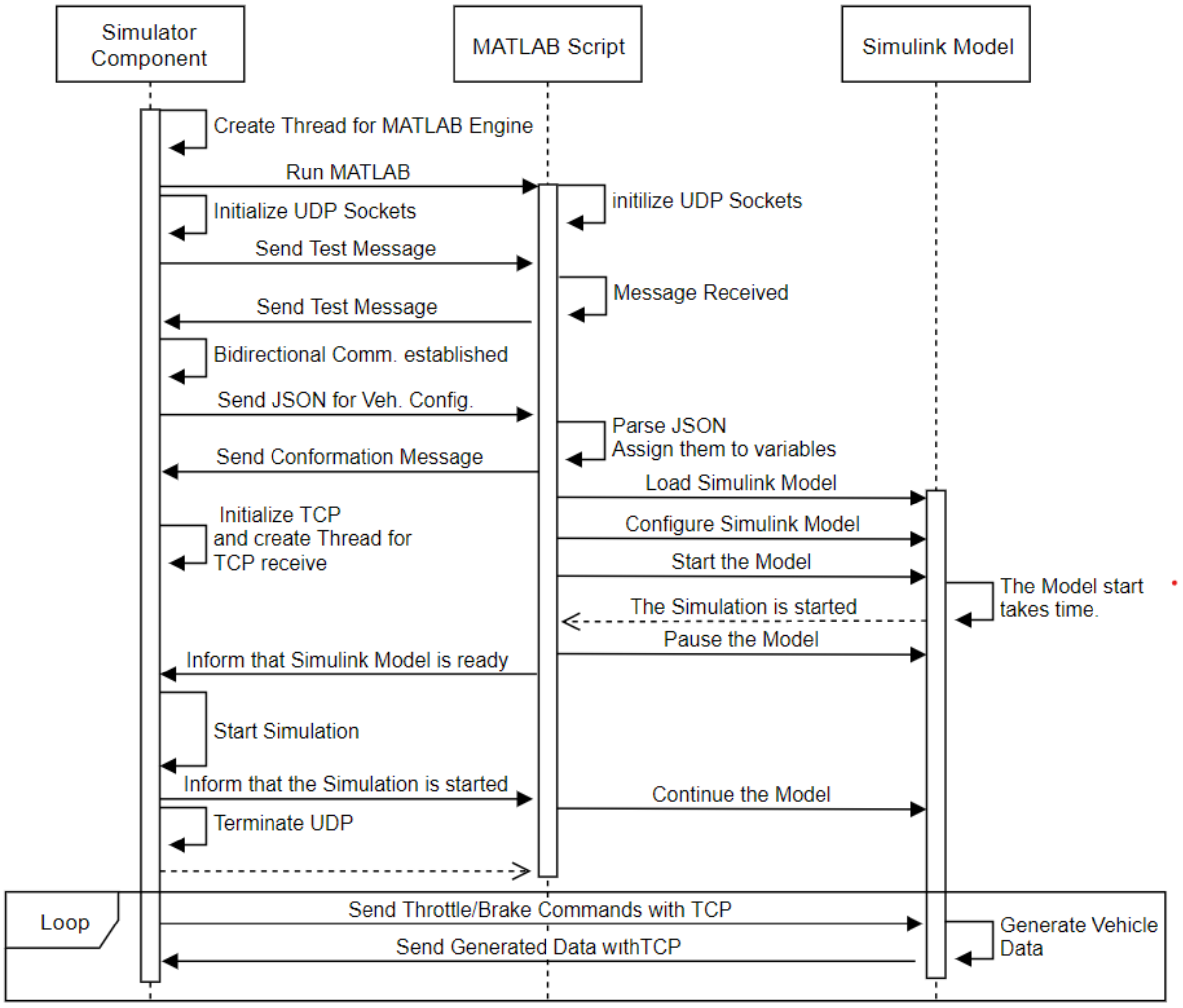


Figure 5.11: Interaction of Simulator and Data Generator components

As soon as the Simulink model is loaded, it is paused by the MATLAB script, and the Simulator Component is notified that the Simulink Model is ready. The aim is to start the simulation and the Simulink model at the same time. The Simulink Model has to wait for a message that is sent from the Simulator Component to the MATLAB Script in order to continue running. This message updates the Data Generator component that the simulation interface of MetaDrive has begun. When this message is received, the MATLAB script completes its tasks. The remaining interaction is only between the Simulink Model and the Simulator Component.

It is possible to write MATLAB code inside the Simulink Model using MATLAB Function Block. The 'fcn' block is a MATLAB Function Block in Figure 5.12 that parses the received messages and assigns them variables for each vehicle. All outputs of the block are transferred to the bus to make a cleaner structure at the Simulink Model. TCP/IP Client Receive Block handles incoming messages. The simulator component transmits the message as a list. The length of the list depends on the number of vehicles in a simulation. The buffer size of TCP communication and handling of messages are modified regarding the array length.

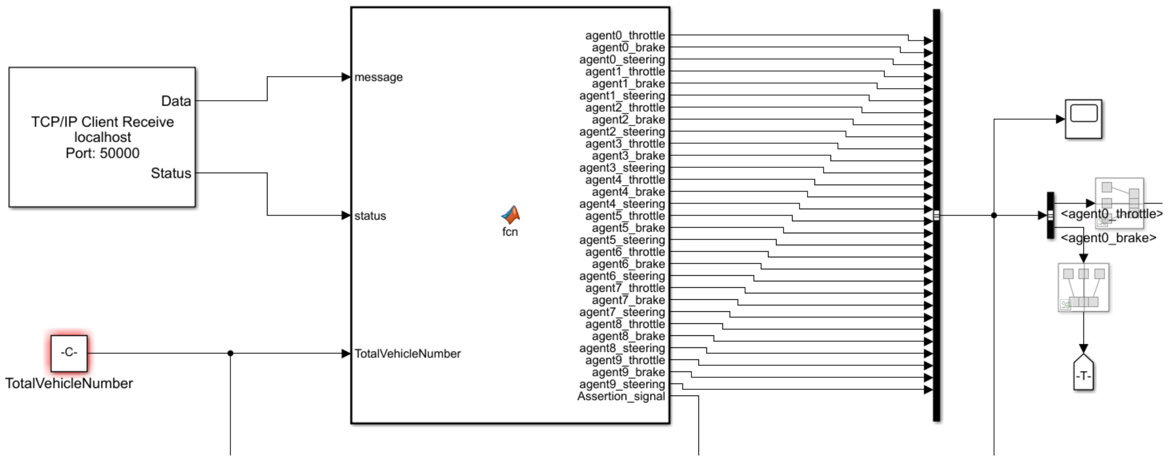


Figure 5.12: Part 1 of the Simulink Model

Part 2 of the Simulink Model in Figure 5.13 shows blocks to replicate a vehicle. The two inputs of the system are visible on the left side of Figure 5.13. They are throttle and brake values. This part contains the engine, shift logic, torque converter, transmission, and vehicle body blocks. Impeller and turbine sensors are introduced to provide extra sensor data for a vehicle. Shift Logic, Transmission, and Vehicle Body blocks contain more blocks inside them. Other blocks in Figure 5.13 are atomic blocks that users can only make a change in parameters of them. The parameters that the user introduces in the GUI are consumed in these atomic blocks. For instance, Engine Block has a 'Max Power' parameter. The value of it is defined with **agents(1).sm_eng_max_power** which is a variable created in the MATLAB script and it is updated to Simulink Model before Simulink Model is started. **agents** is a list that contains ten elements that represent each vehicle. Elements of them are objects of the **simulinkVehicle** class. The class is defined in the "simulinkVehicle.m" file. The generated data for a vehicle are collected at a bus which is named 'x' for this vehicle as shown in Figure 5.13.

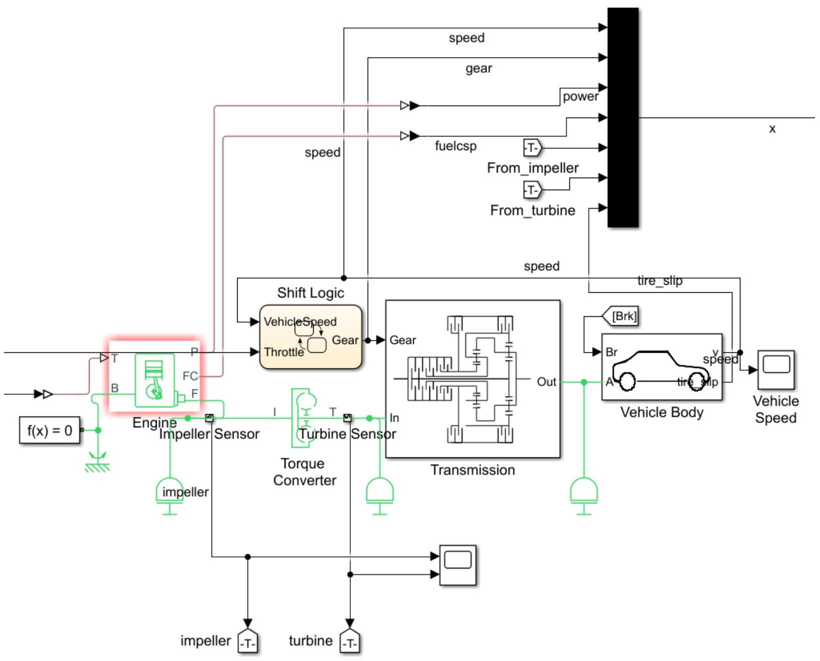


Figure 5.13: Part 2 of the Simulink Model

Figure 5.14 shows the inside of the Vehicle Body Block which consists of Tire Left, Tire Right, Double-Shoe Brake, and Vehicle Body atomic blocks. The two inputs of the block are the brake and the mechanical rotational conserving port for the wheel axle. This block measures the tire slips for two wheels, velocity, and rear axle normal force. Besides, it embodies multiple parameters to configure a vehicle and environment, such as wind velocity, road incline, vehicle mass, and so on.

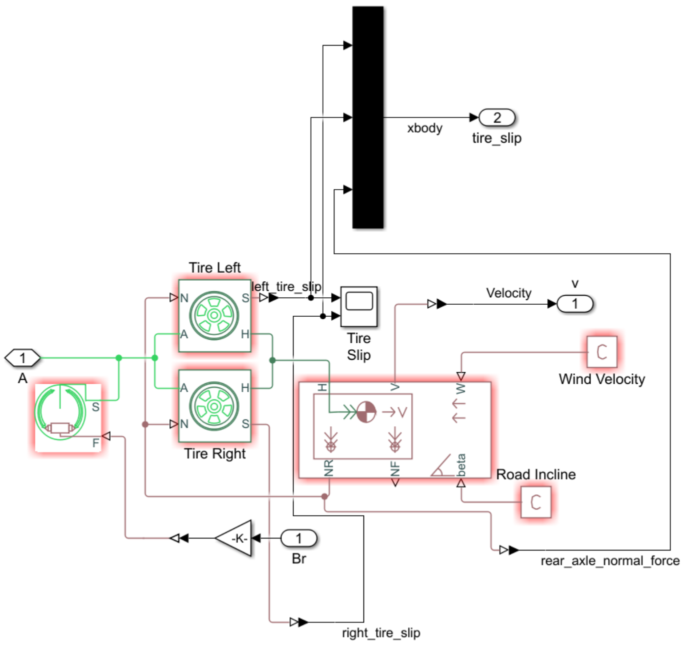


Figure 5.14: Vehicle Body Block

The shift logic block as shown in Figure 5.15 is responsible for changes in gears. It needs vehicle speed and throttle as inputs and provides the gear number as output. It is basically a state machine and acts regarding the inputs. The transmission block is shown in Figure 5.15. Model workspace variables are used to specify the clutch and planetary gear parameters in the blocks. It conveys the motion and torque from one shaft to another while also transferring inertia.

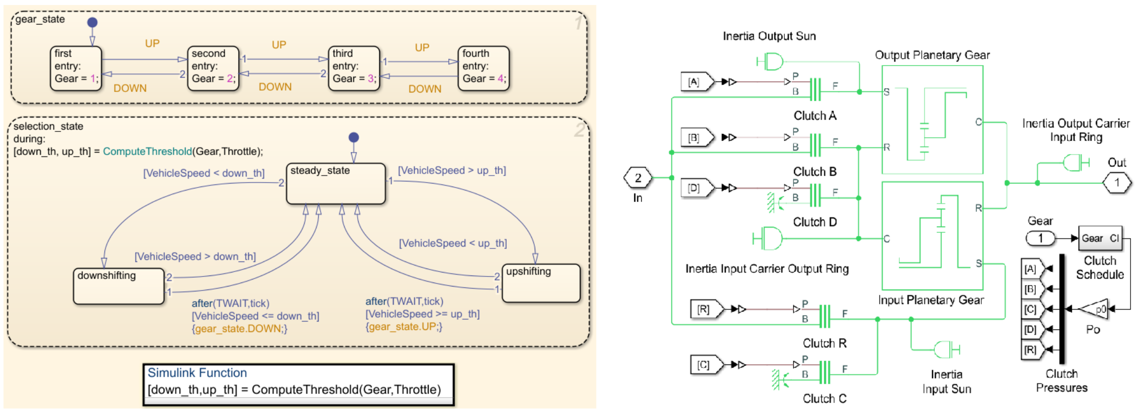


Figure 5.15: 39 Shift Logic Block (Left) and Transmission Block (Right)

Part 3 of the Simulink Model in Figure 5.16 contains blocks with agent names. One block encapsulates all blocks explained in Part 2 of the Simulink Model. There are a total of nine blocks. These blocks represent vehicle models. When you count the model in Part 2 of the Simulink Model, there are ten vehicle models in total. These models are commented out by MATLAB Script when they are not active in the scenario. This increases the performance of the simulator.

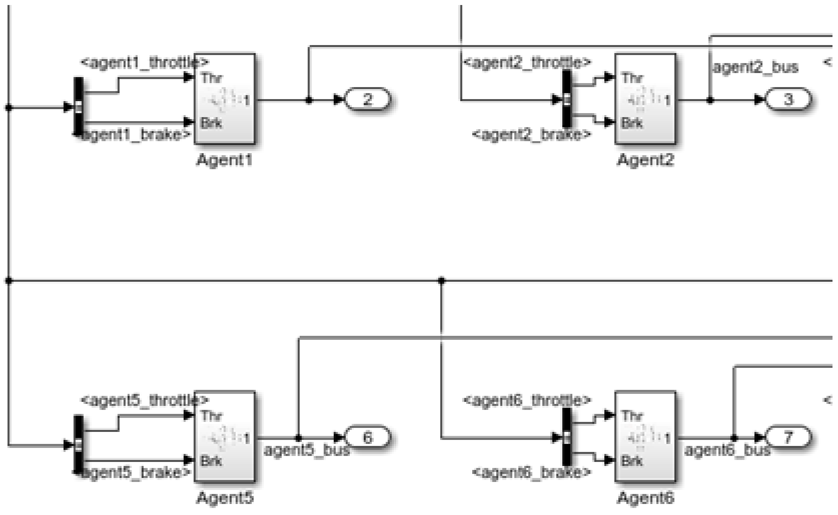


Figure 5.16: Part 3 of the Simulink Model

Part 4 of the Simulink Model is responsible for the transmission of generated vehicle data. For this reason, it has a MATLAB Function block and TCP/IP Client Send block. The MATLAB Function Block gathers data for vehicles and assigns them to a list regarding the total number of vehicles. The message number is also counted and is transmitted as the last element of the list. The Simulation Pace block was also introduced. It does not force the simulation to operate in real-time. Instead, it tries to match the average simulation pace to the chosen clock speed. Clock speed was determined as one per clock per second. This helps the Simulink Model to run almost synchronously with the MetaDrive Simulation. It also provides an output pace error in the second unit. This data is also transferred to Simulator Component via TCP. TCP Receive and TCP Send blocks use the big-endian order. The sampling time of the MATLAB Function Block is defined as discrete, and the sampling time is 0.5 seconds. This affects the TCP Send Block transmission rate because TCP/IP Client Send Block transfers the message when it has a new list on Data input. This means generated data is transferred to the Simulator Component two times in every second.

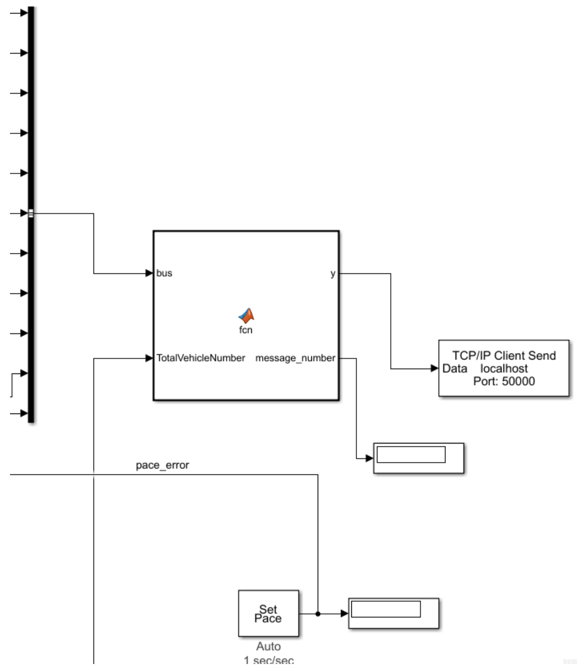


Figure 5.17: Part 4 of the Simulink Model

5.4 Vehicle Control

It provides an interface for operating automobiles on distributed servers. Hence, the user verifies and develops their core connected vehicle algorithm over another device. It was realized using RabbitMQ, a message broker that allows to send and receive messages between applications. Pika module is a client library for Python to implement RabbitMQ. This module is preferred in the thesis. It must be installed onto the servers.

One sender and one consumer are defined for Vehicle Control Component, but users can increase the number regarding their needs. The command sender side was developed to test the component and give an example code to the user. Vehicle commands are hardcoded on this side. The user's core connected car algorithm must change this hardcoded part and sends commands regarding desired behavior of vehicles. Firstly, the connection is created, defined as a blocking connection on the sender side with a host name of 'LocalHost'. The queue name is declared as 'vehicleCommands'.

Messages for commands are transferred to this queue. If there is no queue with the name of ‘vehicleCommands” the code ensures that the queue with this name is created. The message of vehicle commands is a list that has twenty elements. One element for throttle/brake and one element for steering value are defined for a vehicle. The list size is always twenty and does not change regarding the total number of vehicles in the simulation. After each message transmission from the sender side to the receiver side, the connection is closed on the sender side. However, the user can put the code in a loop and send messages continuously. The sender side code example is located at “[MetaDrive Directory]/connectedCars/sender_rabbitMQ.py”.

The receiver side accepts the messages from the queue and processes them to apply commands. This part is inside the “connected_cars_sim.py” file. In a similar manner, firstly, the connection must be created in the receiver part and then the queue is declared. This part consumes messages from the queue using the callback function. In the thesis, the callback function runs when the message is received. It stores the message in a global variable and raises the flag to show that message is captured. The message is processed inside the main loop in every step of the simulation. Then, the commands from the message are applied to the vehicles by sending them to the object of the environment class. The initialization and creation of the RabbitMQ receiver and callback function are done by a threat that works concurrently. Since the threats in Python allows sharing data between them, using global parameters is enough to access the message content.

As the remote server needs the generated data for vehicles to make decisions, the Python script transmits this data via RabbitMQ. For this reason, the ‘vehicleData’ queue is created and the data is directed to this queue. The message is in JSON format and contains data generated in Simulink and MetaDrive which is clarified in the next chapter

5.5 Data Record

This component stores the generated data and vehicle commands in JSON format. The Data Generator component has two parts as explained in Section 5.3 Data Generator. These are the Simulink Model and MetaDrive. The data storage frequency is decided by the user in the GUI. Since this parameter affects the MetaDrive performance, the user needs to determine the frequency concerning the needs of a scenario. The data storage frequency just determines when the data is stored. This means it does not decide which message from the Simulink Model is stored. All received messages from the Simulink Model trigger the creation of data inside the MetaDrive. The received message from the Simulink Model and the generated data inside the MetaDrive are assigned a JSON object. This JSON object is always stored and never skipped. However, it may not be written to the file in every simulation step depending on the data storage frequency, and then it is collected in a list. When the storage frequency matches the step number, all JSON objects in the list are stored at once.

A stored JSON object is composed of JSON objects for each vehicle. Besides, it has a timestamp, simulation step number, and Simulink pace error. The timestamp and the simulation step number are generated in the MetaDrive, and Simulink Pace error data is obtained from the Simulink Model. A JSON object for each vehicle shows command values for a vehicle, generated data from MetaDrive and Simulink. An example of the JSON object is shown below. The comments were added for explanation. They are not part of the stored data.

```
{
      ”timeStamp”: 0.0, // Time counter starts when the MetaDrive starts the simulation
interface
      ”SimStepNumber”: 1, // It increases at each step of MetaDrive simulation
      ”Simulink_pace_error”: 0.0, // It is received by the Simulink Model
      ”agent0”: { // Each vehicle is defined by a JSON object.
      //The data below until ”Simulink_data” key is generated by MetaDrive
            ”speed”: 0.11939543634725779,  // km/hour
            ”throttle_brake”: 0, // Range -1 to 1. Negative values for brake and Positive
values for throttle
            ”steering”: 0, // Range -1 to 1. Negative values for right and Positive values for
 left direction
            ”last_position”: [
                  333.0, // X axis
                  3.0 // Y axis
            ],
            ”dist_to_left_side”: 4.75, // Distance to right pavement in meter
            ”dist_to_right_side”: 5.75, // Distance to right pavement in meter
            ”energy_consumption”: 3.972024742535274e-05, // Fuel consumption in 100km/L
            ”crash_vehicle”: false, // It becomes True in case of contact a vehicle
            ”crash_sidewalk”: false,  // It becomes True in case of contact a sidewalk
            ”crash_on_broken_line”: false, // It becomes True during driving on broken road
line
            ”crash_on_continuous_line”: false, // It becomes True during driving on solid road
 line
            ”lateral_to_left”: 0.263, // Lateral Position on current lane from left
            ”lateral_to_right”: 0.319, // Lateral Position on current lane from right
            ”yaw_rate”: 5.324950333000972e-06, // Current angular acceleration
            ”velocity”: 0.0110838, // Velocity of vehicle
            ”lat_pos_on_current_lane”: 0.499, // Lateral Position on current lane
            ”other_v_info”: [ // This part is explained more detailed at below.
                  0.449,
                  0.537,
                  0.500,
                  0.499,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0
            ],
            "Simulink_data": { // The generated data from the Simulink
                  "speed": 0.0, // km/hour
                  "gear": 1.0,
                  "eng_power": -1.7857886709804195e-13, // Watt/second
                  "eng_fuelcsp": 0.00039999999999999996, // Milligram/revolution
                  "impeller_speed": 959.9999999999999, // RPM
                  "impeller_trq": 66.86939822111523, // Impeller torque in N*m
                  "turbine_speed": 0.0, // RPM
                  "turbine_trq": -149.2524968295292, // Turbine torque in N*m
                  "right_tire_slip": 0.0, // Unit is not specified
                  "left_tire_slip": 0.0, // Unit is not specified
                  "rear_axle_normal_force": 2746.799 // Physical signal. Unit is not specified
            }
        },"agent1": {  ...
        },"agent2": { ...
        },"agent3": {  ...
        },"agent4": {  ...
        },"agent5": {  ...
        },"agent6": {  ...
        },"agent7": {  ...
        },"agent8": {  ...
        },"agent9": {  ...
        }
}
```

Listing 5.1: Vehicle Data in JSON object

other_v_info key means other vehicle information. This section is generated using the Lidar observer of the MetaDrive. The Lidar observer is limited to generating data for the five closest vehicles around the agent. There is a four-line represents different information for each of these five closest vehicles. The first four values of the list can be explained below:

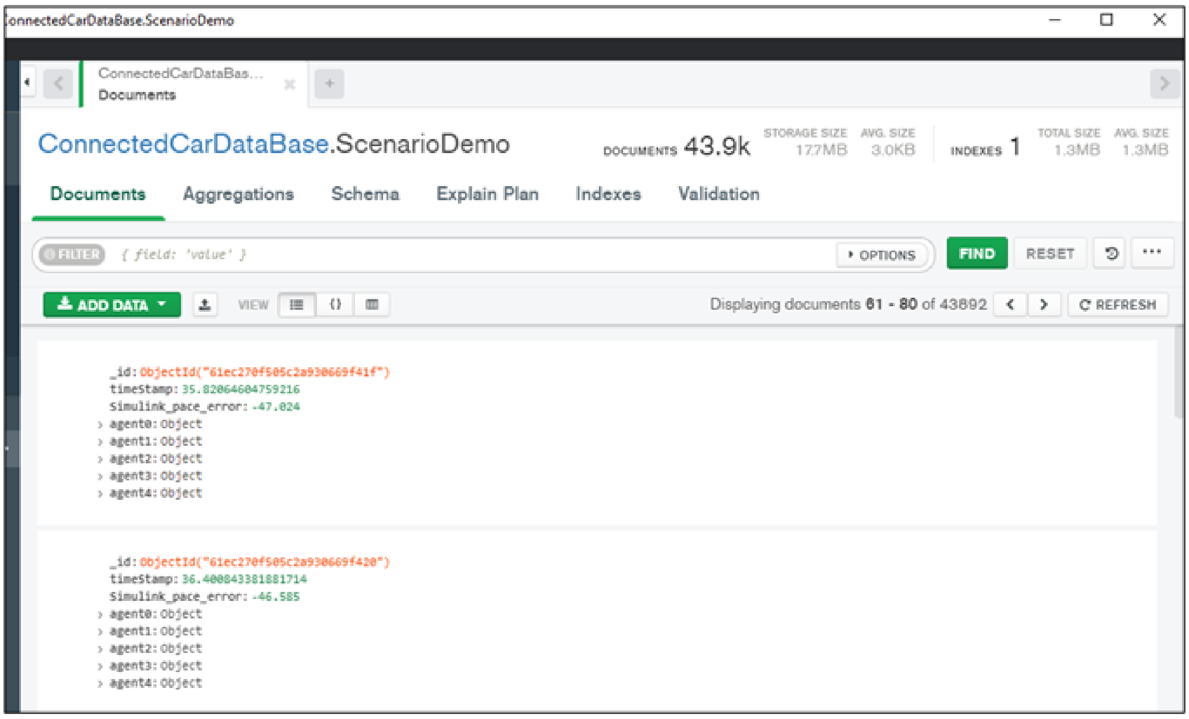
- 0.449 -> Projection of distance between ego and another vehicle on ego vehicle's heading direction [Dec22]
- 0.537 -> Projection of distance between ego and another vehicle on ego vehicle's side direction [Dec22]
- 0.500 -> Projection of speed between ego and another vehicle on ego vehicle's heading direction [Dec22]
- 0.499 -> Projection of speed between ego and another vehicle on ego vehicle's side direction [Dec22]

Two hundred forty Lidar points for a vehicle can be added to the JSON object. However, Lidar points are not contained by the JSON object for simplicity. When other vehicles are not inside the vicinity of the vehicle, the values of these vehicles are written as zero to the JSON object.

The JSON objects are saved in "[MetaDrive Directory]/agentsData.JSON" as a default setting. Nevertheless, the user can change the file directory and the file name using the GUI as it is explained in the earlier chapters.

Another option to store JSON objects is utilizing MongoDB. It is a NoSQL database that saves JSON-like objects on a huge scale. The prominent difference between MongoDB and other databases is that MongoDB manages storage with collections of documents. It is scalable, flexible, and provides hierarchical storage. Structured and unstructured data can be stored in MongoDB, and it provides an easier extension possibility on the connected car simulator. Besides, it is human readable and provides the MongoDB Compass tool to analyze data.

PyMongo package is a Python package that enables to work with MongoDB in Python code. It was installed and imported into the “connected_cars_sim.py” file. Then, the client to store data is created inside the **initialize_mongo** function. The data for this thesis is stored on local host. However, MongoDB supports disturbed data storage. After the creation of the client, the database is constructed with the name entered by the user in the GUI. The collection name inside the database is also written by the user using the GUI. The user can decide to turn on the MongoDB storage option from the GUI. The generated data is inserted into the collection in the same manner as the JSON file storage option. The frequency selection of the users affects both of the storage options. The user can easily access, filter, order, and retrieve data from MongoDB for the core connected car algorithm. It just requires the same PyMongo package and a couple of lines of code. This provides great convenience to the user for extending the core connected car algorithm. The screenshot of the generated data in MongoDB is shown in Figure 5.18.



```
_id: ObjectId("61ec270f505c2a930669f41f")
timeStamp: 35.82064604759216
Simulink_pace_error: -47.024
> agent0: Object
> agent1: Object
> agent2: Object
> agent3: Object
> agent4: Object

_id: ObjectId("61ec270f505c2a930669f420")
timeStamp: 36.400843381881714
Simulink_pace_error: -46.585
> agent0: Object
> agent1: Object
> agent2: Object
> agent3: Object
> agent4: Object
```

Figure 5.18: Generated Data in MongoDB Compass

6 Test and Evaluation

This chapter evaluates the simulator, GUI and implementation details that are explained in Chapter 5. Test and evaluation are described in two sections. The first of them is assessing the GUI and the other one is for the simulator.

6.1 Test of Graphical User Interface

The main assignment of Graphical User Interface is to provide a tool to configure scenarios and create maps. The user can generate a map using the “Map Creator” tab of the GUI. This section was tested by developing three maps displayed in Figure 5.7. These three maps overall contain every roadblock types. They are presented to users as default maps. Therefore, users can use these maps in their scenarios. The map with three roadblocks was generated for the test. Figure 6.1 shows the GUI and MetaDrive Interface for testing the map.

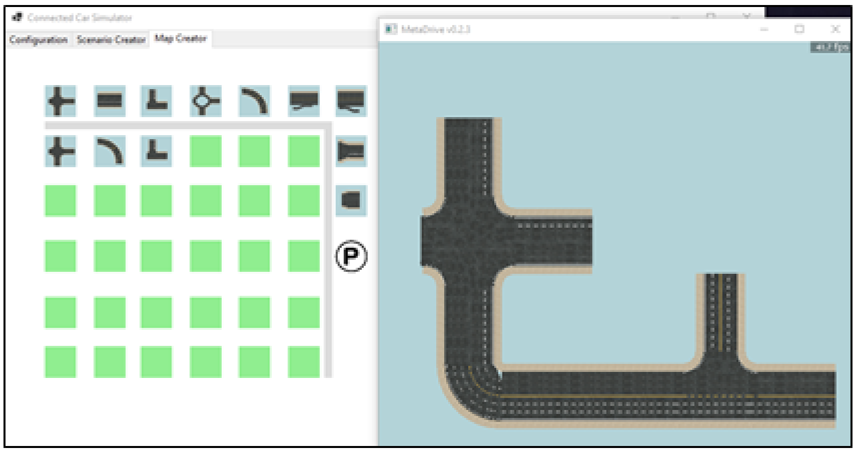


Figure 6.1: Test of Map Creator

The map has an STD Intersection, Curve, and T intersection blocks. The ability to change roadblocks during map creation was tested by putting In Ramp block instead of Curve block. As seen in Figure 6.2, the change was applied to the block in the middle and did not affect other blocks. Then the map was saved with “Test3Block” name. This enables the user to select a map from the Scenario Creator tab.

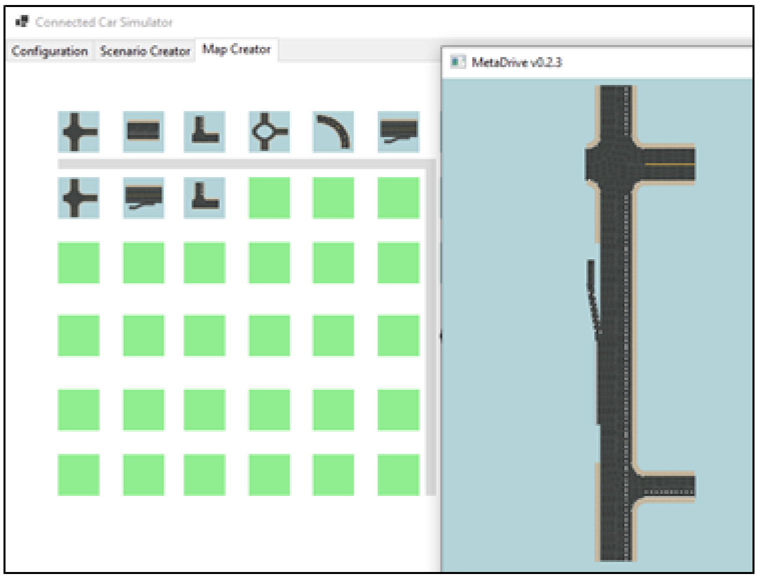
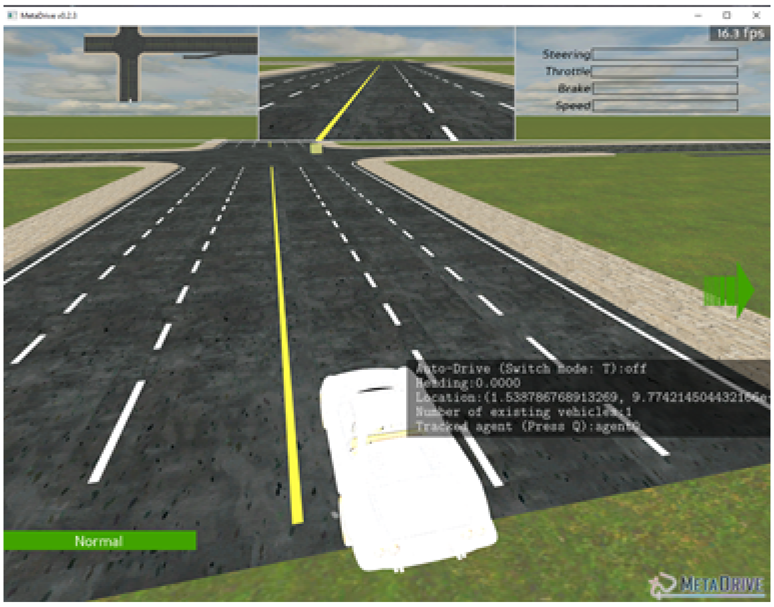


Figure 6.2: Change in Roadblock

After the map was chosen from the combo box at the Scenario Creator tab, the “Run” button is clicked to define a course for a vehicle and test this feature. This opened the MetaDrive Interface to record vehicle behavior. Since the default total number of vehicles is one, it opens the Interface with one car as shown in Figure 6.3. The Interface starts with the bird’s-eye view. The user can press Q on the keyboard to drive the themselves.



```
Auto-Drive (Switch mode: T):off
Heading:0.0000
Location:(1.538786768913269, 9.774214504432166
Number of existing vehicles:1
Tracked agent (Press Q):agent0
```

Figure 6.3: Define a course for a car with Run&Create

After that, the “Create” button imports the generated Comma-separated values (CSV) file which contains the vehicle commands. It is possible to see all steps and modify them using GUI as shown in Figure 6.4. The user can set the initial position of the vehicle. We selected 0,0 for X and Y-axis. The location and heading values of the selected vehicle are displayed on the MetaDrive Interface as in Figure 6.3. The user can detect the location from there for deciding other vehicles’ initial values.

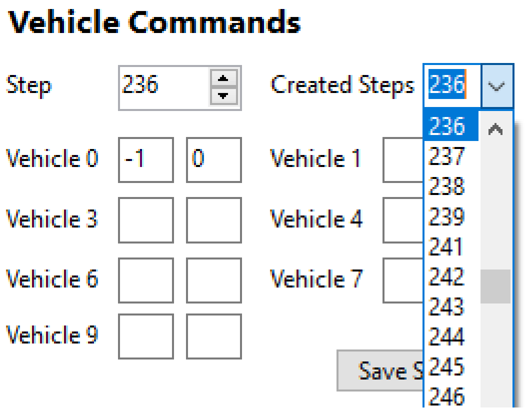
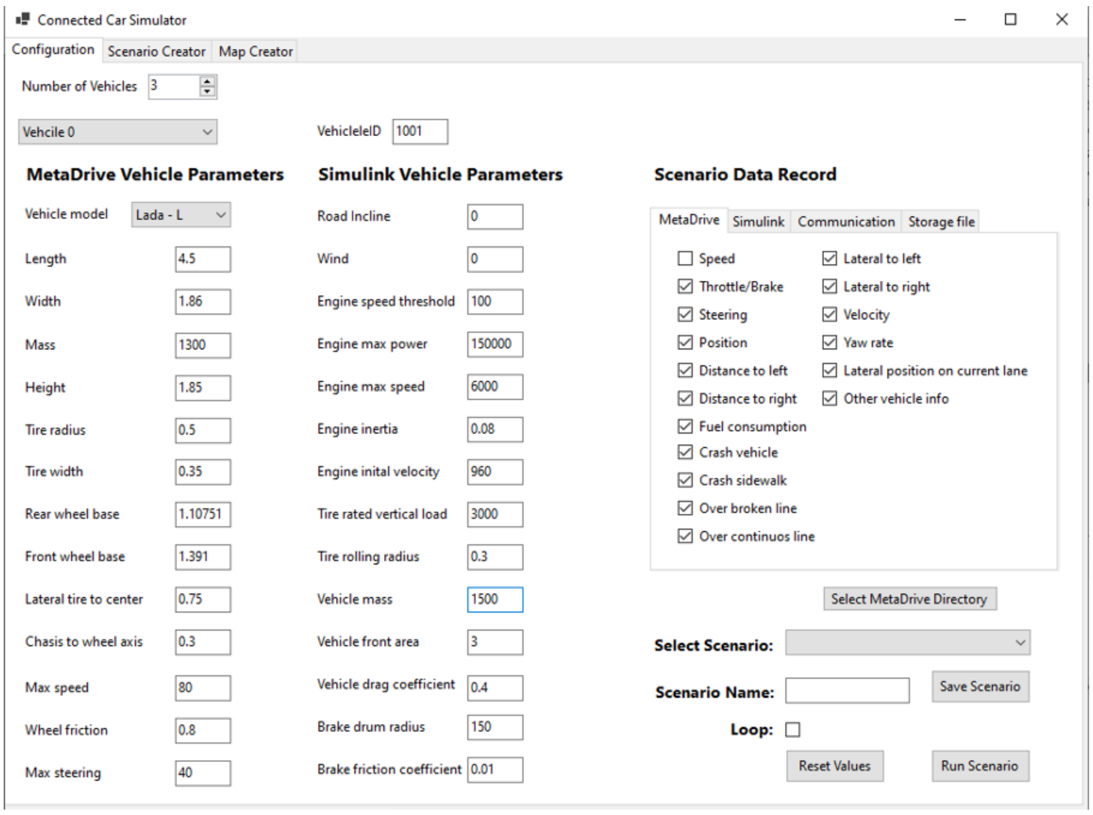


Figure 6.4: Vehicle Commands for created course

Functionalities of Map Creator and Scenario Creator tabs were tested. The remaining section for the test is Configuration Tab. Since it has sixty-eight text and combo boxes for defining a scenario, it is not possible to change them and show their effects in the thesis. Thus, a few changes that affect the scenarioConfiguration.py file and Simulink Model are demonstrated. Firstly, the number of vehicles was increased from one to three. The vehicle model of “Vehicle 0” was selected as “Lada-L” and value of Tire Radius was changed to 0.5 from 0.39. The box for Speed value was unchecked under the Scenario Data Record section of the GUI. For observing the change in Simulink Model, the vehicle mass was set to 1500 instead of 1200. These changes are written to the scenarioConfiguration.py file when the user clicks on the “Run Scenario” button. Figure 6.5 shows the Configuration tab for the scenario and Listing 6.1 shows the modified variables in the scenarioConfiguration.py. This verifies that GUI correctly interacts with the MetaDrive The GUI does not communicate with the Simulink Model directly. The **simulink_configuration_JSON** is transmitted to the MATLAB script via Python code for MetaDrive and the MATLAB Script modifies parameters in Simulink.



Connected Car Simulator

Configuration | Scenario Creator | Map Creator

Number of Vehicles: 3

Vehcile 0

VehicleleID: 1001

MetaDrive Vehicle Parameters

Vehicle model	Lada - L
Length	4.5
Width	1.86
Mass	1300
Height	1.85
Tire radius	0.5
Tire width	0.35
Rear wheel base	1.10751
Front wheel base	1.391
Lateral tire to center	0.75
Chasis to wheel axis	0.3
Max speed	80
Wheel friction	0.8
Max steering	40

Simulink Vehicle Parameters

Road Incline	0
Wind	0
Engine speed threshold	100
Engine max power	150000
Engine max speed	6000
Engine inertia	0.08
Engine inital velocity	960
Tire rated vertical load	3000
Tire rolling radius	0.3
Vehicle mass	1500
Vehicle front area	3
Vehicle drag coefficient	0.4
Brake drum radius	150
Brake friction coefficient	0.01

Scenario Data Record

MetaDrive | Simulink | Communication | Storage file

- ☐ Speed
- ☑ Throttle/Brake
- ☑ Steering
- ☑ Position
- ☑ Distance to left
- ☑ Distance to right
- ☑ Fuel consumption
- ☑ Crash vehicle
- ☑ Crash sidewalk
- ☑ Over broken line
- ☑ Over continuos line
- ☑ Lateral to left
- ☑ Lateral to right
- ☑ Velocity
- ☑ Yaw rate
- ☑ Lateral position on current lane
- ☑ Other vehicle info

Select MetaDrive Directory

Select Scenario:

Scenario Name: | Save Scenario

Loop: ☐

Reset Values | Run Scenario

Figure 6.5: Test of Configuration Tab

Listing 6.1 Test of Configuration Tab - Changes in scenarioConfiguration.py

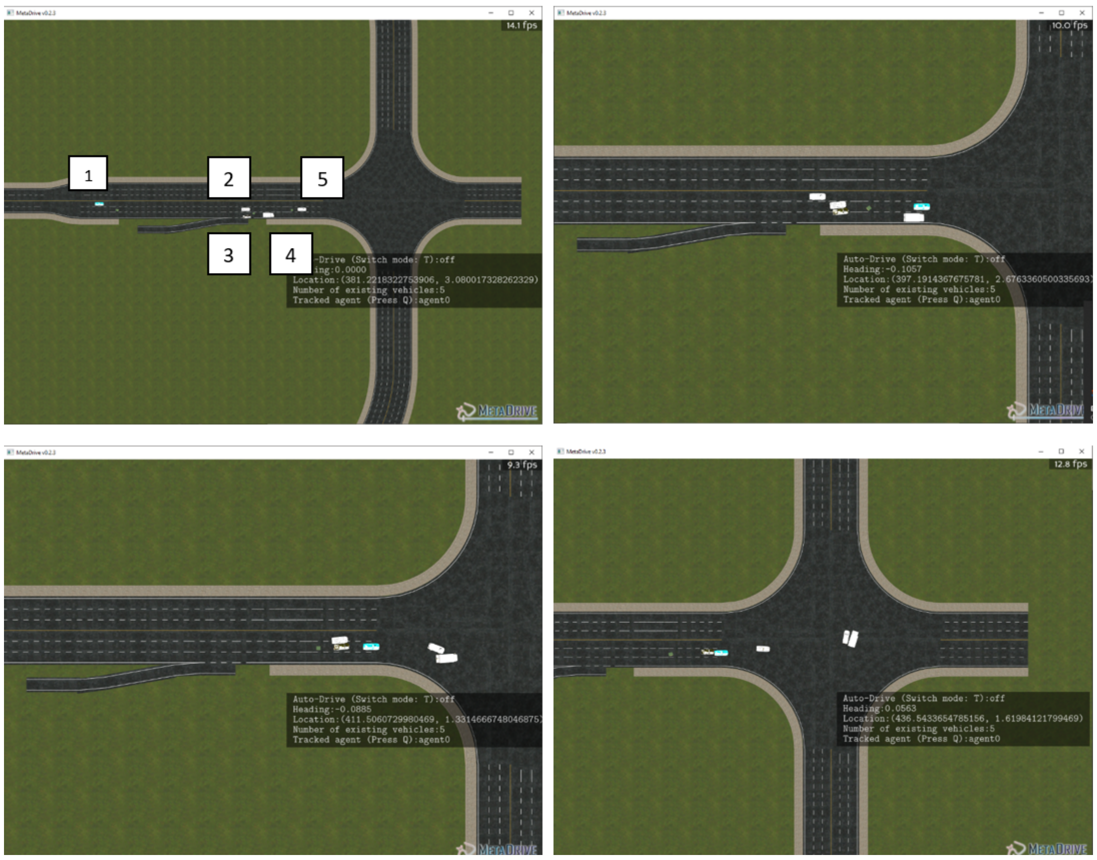
```
{
vehicle_model_list = ["l","default","default","default","default","default","default","default
","default","default",]

simulink_configration_JSON = {
    "SimulinkModel":"Configuration",
    "TotalVehicleNumber":3,
    "Agent0":{
        "VehicleID": 1001,
        "sm_road_incline_value": 0,
        "sm_wind_value": 0,
        "sm_eng_speed_threshold": 100,
        "sm_eng_max_power": 150000,
        "sm_eng_max_speed": 6000,
        "sm_eng_inertia": 0.08,
        "sm_eng_initial_velocity": 960,
        "sm_eng_fuel_csp_constant": 25,
        "sm_tire_rated_vertical_load": 3000,
        "sm_tire_rolling_radius": 0.3,
        "sm_vehicle_mass": 1500,
        "sm_vehicle_front_area": 3,
        "sm_vehicle_drag_coefficient": 0.4,
        "sm_brake_drum_radius": 150,
        "sm_brake_friction_coefficient": 0.01,
        }  ...}

vehicle_Model_configration_dic = {
    ...
    "L":{
    "Length": 4.5,
    "Width": 1.86,
    "Mass": 1300,
    "Height": 1.85,
    "Tire_raduis": 0.5,
    "Tire_width": 0.35,
    "Rear_wheel_base": 1.10751,
    "Front_wheel_base": 1.391,
    "Lateral_tire_center": 0.75,
    "Chasis_to_wheel_axis": 0.3,
    "Max_speed": 80,
    "Wheel_friction": 0.8,
    "Max_steering": 40,
    }, ... }

data_storage_configration_dic = {
    "speed": False,
    "throttle_brake": True,
    "steering": True,
    "last_position": True,
    ...}
```

6.2 Test of Simulator

It is mentioned in Chapter 5 Implementation that there are four default scenarios. This section firstly briefly explains these four scenarios and then focuses on one of them to assess the simulator.

Highway-Crash Scenario

This scenario is implemented on the Highway map and consists of five vehicles. The goal of the scenario is the observation of vehicles that drive close to each other and to avoid an accident. At the end of the scenario, vehicles have a crash. It is expected from the core connected car algorithm to foresee the accident and prevent it. Four different steps of the scenario are displayed in Figure 6.6. The top left image shows the vehicles' initial position. Vehicle number 3 on the ramp tries to join the highway when there is a vehicle near to it and vehicle number 1 in the back drives faster than all other vehicles.



```
-Drive (Switch mode: T):off
ing:0.0000
Location:(381.2218322753906, 3.080017328262329)
Number of existing vehicles:5
Tracked agent (Press Q):agent0
```

```
Auto-Drive (Switch mode: T):off
Heading:-0.1057
Location:(397.1914367675781, 2.676336050033569
Number of existing vehicles:5
Tracked agent (Press Q):agent0
```

```
Auto-Drive (Switch mode: T):off
Heading:-0.0885
Location:(411.5060729980469, 1.331466674804687
Number of existing vehicles:5
Tracked agent (Press Q):agent0
```

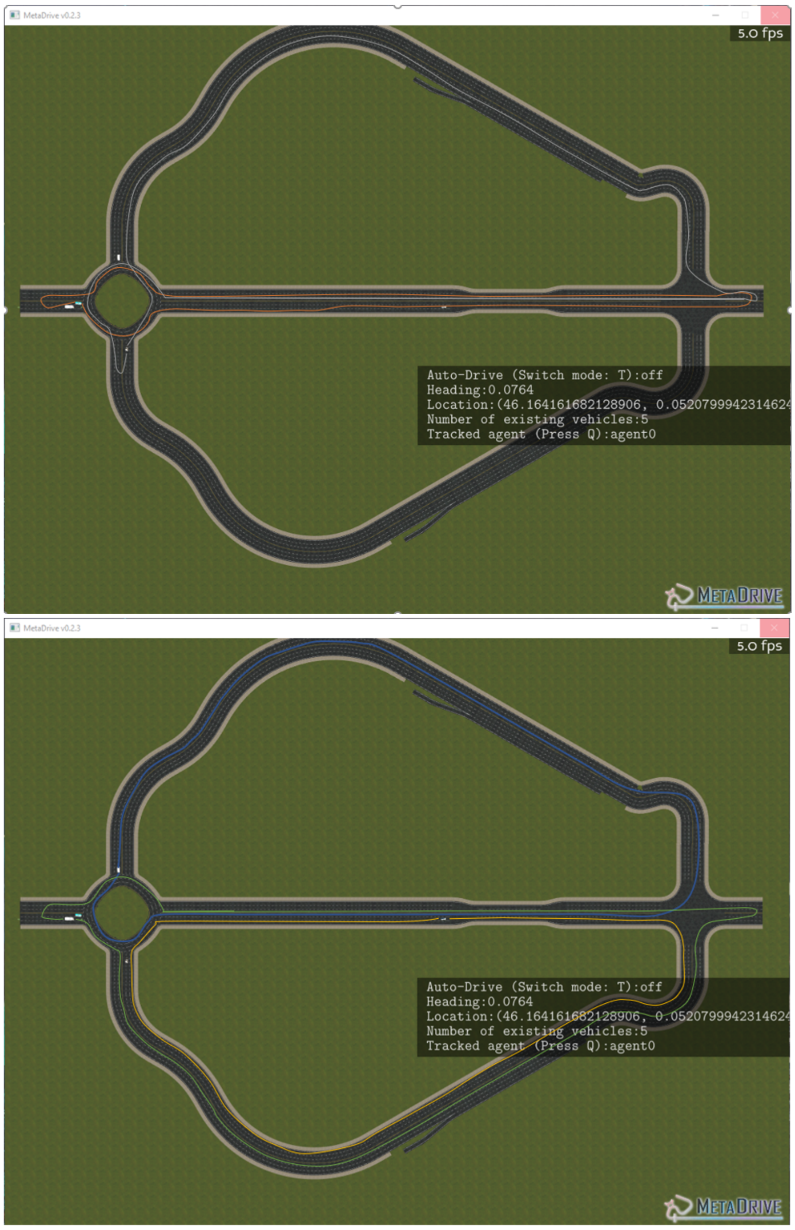
```
Auto-Drive (Switch mode: T):off
Heading:0.0563
Location:(436.5433654785156, 1.61984121799469)
Number of existing vehicles:5
Tracked agent (Press Q):agent0
```

Figure 6.6: Highway Crash Scenario

Vehicle 3 slightly touches vehicle 2 near to it shown in the top right image of the figure. This makes vehicle 2 to contact vehicle 1 which is very fast at that moment. This minor touch changes the direction of vehicle 1 and it crashes into vehicle 4 which tries to turn left as in the right bottom image of the figure. The scenario ends shown the bottom right image. Vehicles' courses are determined using the Run&Create feature of the GUI.

Curves-Long Scenario

This scenario is realized on Curves map with five vehicles. This one is the longest default scenario and it takes around 3 minutes. Vehicles turn back to their initial position at the end of the scenario. When the user activates the Loop option on the GUI, the scenario runs continuously. The aim of the scenario is synthetic data generation for a longer time. The user may utilize the scenario to test and optimize vehicle behaviors with respect to fuel consumption and time. Courses of vehicles are illustrated by colored lines in Figure 6.7. This scenario generates 156 documents in MongoDB which equals to 200.7 kilobytes. Document is a definition of a JSON-like object in MongoDB. Vehicles' courses are generated by the Run&Create feature.



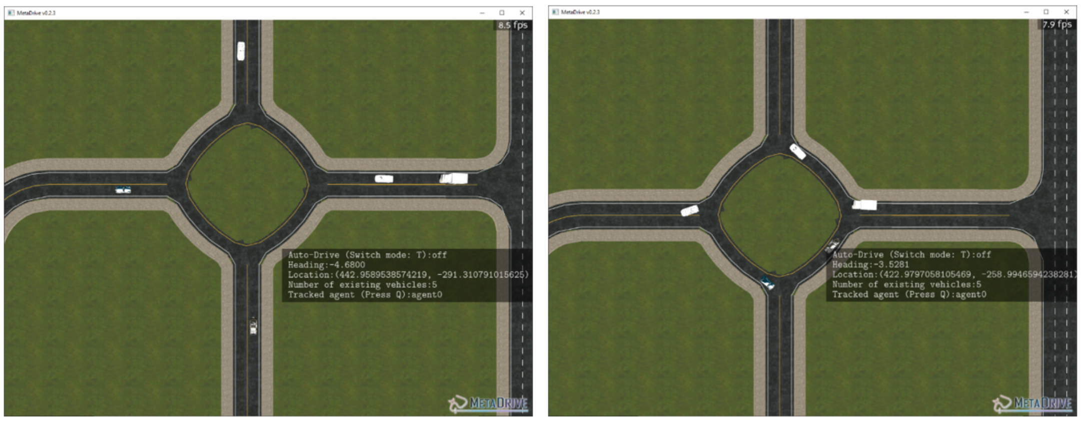
```
Auto-Drive (Switch mode: T):off
Heading:0.0764
Location:(46.164161682128906, 0.0520799942314624
Number of existing vehicles:5
Tracked agent (Press Q):agent0
```

```
Auto-Drive (Switch mode: T):off
Heading:0.0764
Location:(46.164161682128906, 0.0520799942314624
Number of existing vehicles:5
Tracked agent (Press Q):agent0
```

Figure 6.7: Curves-Long Scenario

Roundabout - Scenario

The Roundabout Scenario contains five vehicles and uses the Big City map which is the biggest default map. The five vehicles are located around a roundabout and four of them enter the roundabout at the same time. They take the first exit by turning right. The truck in the back also enters the roundabout and follows a similar course. The goal of the scenario is to analyze the vehicle interaction in a dense roundabout environment. The first step of the scenario is shown in the left image of Figure 6.8. The right image of Figure 6.8 displays vehicles before they turn right. Vehicles' routes were created using text boxes for Vehicle Commands in the GUI, not Run&Create feature.

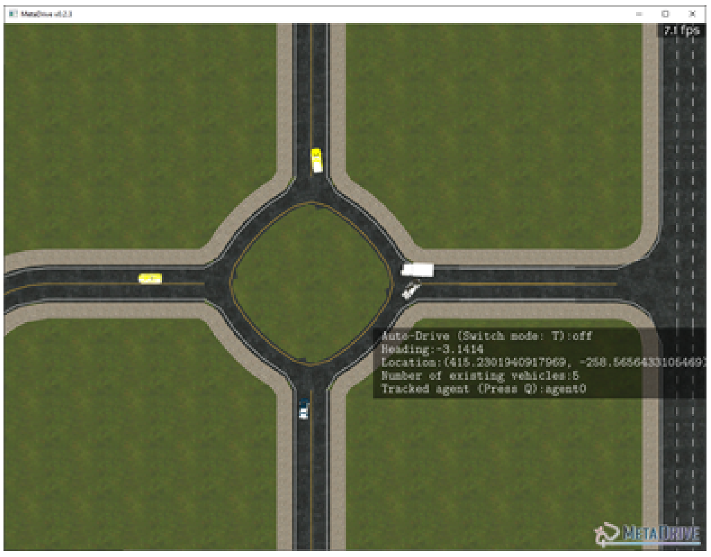


```
Auto-Drive (Switch mode: T):off
Heading:-4.6800
Location:(442.9589538574219, -291.310791015625)
Number of existing vehicles:5
Tracked agent (Press Q):agent0
```

```
Auto-Drive (Switch mode: T):off
Heading:-3.5281
Location:(422.9797058105469, -258.9946594238281
Number of existing vehicles:5
Tracked agent (Press Q):agent0
```

Figure 6.8: Roundabout Scenario

Roundabout - Crash Scenario This scenario is very similar to the Roundabout Scenario above. The only difference is that the vehicle which starts at the bottom hits the truck at the end of the scenario as in Figure 6.9. The scenario contains 100 steps and it takes 22 seconds.



```
Auto-Drive (Switch mode: T):off
Heading:-3.1414
Location:(415.2301940917969, -258.5656433105469
Number of existing vehicles:5
Tracked agent (Press Q):agent0
```

Figure 6.9: Roundabout-Crash Scenario

Generated data of this scenario is inspected to test and evaluate the simulator. Synthetic data at the beginning of the scenario, before the crash, and at the moment of the accident are examined. Vehicles have no initial speed at the start time. Data of vehicles that are collided with each other are analyzed. Vehicle 2 is located on the bottom side of the roundabout and Vehicle 4 is located on the right side of the roundabout as shown in Figure 6.8. Vehicle 4 is the truck that is behind the other vehicle at the start of the scenario. The first recorded JSON for two vehicles is shown below:

```
{
      "timeStamp": 0.0,
      "SimStepNumber": 1,
      "Simulink_pace_error": 0.0,
      "agent0": {...},
      "agent1": {...},
      "agent2": {
            "speed": 0.3572383068432845,
            "throttle_brake": 0.5,
            "steering": 0,
            "last_position": [
                  446.0899963378906,
                  -221.25999450683594
            ],
            "dist_to_left_side": -219.5123748779297,
            "dist_to_right_side": 230.0123748779297,
            "energy_consumption": 7.76389216947102e-05,
            "crash_vehicle": false,
            "crash_sidewalk": false,
            "crash_on_broken_line": false,
            "crash_on_continous_line": false,
            "lateral_to_left": 0.0,
            "lateral_to_right": 1.0,
            "yaw_rate": 0.0,
            "velocity": 0.013438003038052322,
            "lat_pos_on_current_lane": 0.5199991861979367,
            "other_v_info": [
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
            0.0
        ],
        "Simulink_data": {
            "speed": 0.0,
            "gear": 1.0,
            "eng_power": -1.7857886709804195e-13,
            "eng_fuelcsp": 0.00039999999999999996,
            "impeller_speed": 959.9999999999999,
            "impeller_trq": 66.86939822111523,
            "turbine_speed": 0.0,
            "turbine_trq": -149.2524968295292,
            "right_tire_slip": 0.0,
            "left_tire_slip": 0.0,
            "rear_axle_normal_force": 2746.7999999999997
        }
    },
    "agent3": {...},
    "agent4": {
        "speed": 0.4331828514965217,
        "throttle_brake": 0.5,
        "steering": 0,
        "last_position": [
            496.70001220703125,
            -258.8299865722656
        ],
        "dist_to_left_side": -257.0801086425781,
        "dist_to_right_side": 267.5801086425781,
        "energy_consumption": 9.471591298331718e-05,
        "crash_vehicle": false,
        "crash_sidewalk": false,
        "crash_on_broken_line": false,
        "crash_on_continous_line": false,
        "lateral_to_left": 0.0,
        "lateral_to_right": 1.0,
        "yaw_rate": 6.657334537413444e-06,
        "velocity": 0.014189929222737839,
        "lat_pos_on_current_lane": 0.4622463650173611,
        "other_v_info": [
            0.7114705596069215,
            0.49194966640012294,
            0.49985524632328726,
            0.49997978764664475,
            0.0,
            0.0,
            0.0,
            0.0,
            0.0,
            0.0,
            0.0,
            0.0,
            0.0,
            0.0,
            0.0,
                    0.0,
                    0.0,
                    0.0,
                    0.0,
                    0.0
                ],
                "Simulink_data": {
                    "speed": 0.0,
                    "gear": 1.0,
                    "eng_power": -1.7857886709804195e-13,
                    "eng_fuelcsp": 0.00039999999999999996,
                    "impeller_speed": 959.9999999999999,
                    "impeller_trq": 66.86939822111523,
                    "turbine_speed": 0.0,
                    "turbine_trq": -149.2524968295292,
                    "right_tire_slip": 0.0,
                    "left_tire_slip": 0.0,
                    "rear_axle_normal_force": 2746.7999999999997
                }
        }
}
```

Listing 6.1: JSON object from the Scenario

Position, throttle_brake, speed, and other_v_info values play a key role to prevent an accident. Since Vehicle 2 has no other vehicle in its Lidar distance which is defined as 40 meters at the connected_cars_metadrive.py file, the other_v_info list contains just zeros. A connected car simulator uses 70 laser points for each vehicle to simulate Lidar. However, when we look at Vehicle 4, the first four values are not zero. The first two values show the distance from the ego vehicle to the detected object/car. The third and fourth values demonstrate speed differences. These values range from zero to one. Figure 6.10 helps to explain the meaning of values at the other_v_data. The ego vehicle moves in the direction of the blue arrows. When the stationary vehicle is inside the Lidar Radius of the ego vehicle then the first value of other_v_data becomes 1. At the moment of contact, the value is 0.5. When the stationary vehicle is at the edge of the Lidar Radius and the back of the ego vehicle the value becomes closer to 0. Thus, the heading direction is prominent in this value. The second value of the other_v_data provides Lidar information related to the side direction.

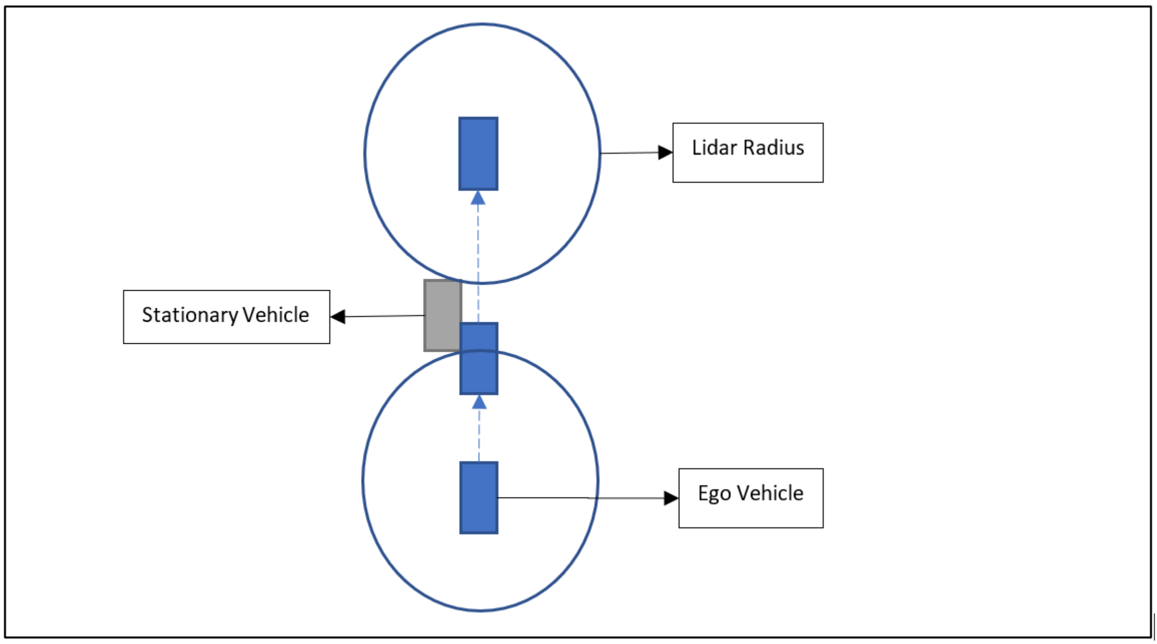


Figure 6.10: Explanation of other_v_data variable

Thus, it is expected to observe 0.5 at the crash moment of the scenario. The collision is firstly detected at step number 97 in the JSON object. This can be understood thanks to crash_vehicle arguments. Its value becomes True during vehicle's contact. The first two values of vehicle 2 other_v_info are 0.538, 0.480. They were zero at the first JSON object. Besides, the location values of the two vehicles indicate that they are contacted.

Vehicle 2: X-axis = 463.17987060546875, Y-axis = -256.17510986328125

Vehicle 4: X-axis = 464.6471862792969 , Y-axis = -260.1637878417969

The Simulink Model generates extra data which is not critical for this scenario. The speed values from Simulink Model and MetaDrive do not match each other in this scenario, because the engines were not defined with the similar features. In addition, it was observed that the speed data from the Simulink Model stays lower than the expected value for short simulations such as, this one. However, it is not the case in long scenarios such as Curve–Long.

The scenario was run three times to evaluate reproducibility which is an essential requirement for the thesis. The vehicle behaviors were the same at these three trials. The first run of the scenario generates 23 JSON objects in 44.03kilobyte data size, the second one 26 JSON objects in 49.07 kilobyte data size, and the last one also had 23 JSON objects. Vehicle 0 data from scenario trial 1 and 2 are shown in Table 6.1.The left section shows data from the Trail 1 and the right section shows data from the Trial 2. They are selected from closer steps. Since the data storage does not happen in every simulation step, the JSON objects are from steps number 7 and 8 for two different scenario trials. As it is seen in the table, values are almost identical: This testifies that the reproducibility requirement is fulfilled.

Table 6.1: JSON Comparison of Scenarios

```
"timeStamp": 7.8866868019104,
      "SimStepNumber": 11,
      "Simulink_pace_error": 0.0,
      "agent0": {
            "speed": 5.468129894105523,
            "throttle_brake": 0.5,
            "steering": 0,
            "last_position": [
                  442.9589538574219,
                  -291.310791015625
            ],
            "dist_to_left_side": -289.4200134277344,
            "dist_to_right_side": 299.9200134277344,
            "energy_consumption": 0.026721136519312332,
            "crash_vehicle": false ,
            "crash_sidewalk": false ,
            "crash_on_broken_line": false ,
            "crash_on_continous_line": false ,
            "lateral_to_left": 0.0,
            "lateral_to_right": 1.0,
            "yaw_rate": 0.0,
            "velocity": 0.06404089004064875,
            "lat_pos_on_current_lane": 0.5101318359375,
            "other_v_info": [
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0

            ],
            "Simulink_data": {
                  "speed": 3.3991506548576556e-05,
                  "gear": 1.0,
                  "eng_power": 2.350315311591788e-21,
                  "eng_fuelcsp": 6.900268403020609e-07,
                  "impeller_speed": 1.6560644167249459,
                  "impeller_trq": 0.00019896338888595593,
                  "turbine_speed": 0.0013738732060668208,
                  "turbine_trq": -0.00044385988730293843,
                  "right_tire_slip": 2.148741063928254e-08,
                  "left_tire_slip": 2.148741063928254e-08,
                  "rear_axle_normal_force": 2746.80035200028
```

```
"timeStamp": 6.177844524383545,
      "SimStepNumber": 12,
      "Simulink_pace_error": 0.0,
      "agent0": {
            "speed": 5.965230408482458,
            "throttle_brake": 0.5,
            "steering": 0,
            "last_position": [
                  442.95440673828125,
                  -291.1700134277344
            ],
            "dist_to_left_side": -289.26544189453125,
            "dist_to_right_side": 299.76544189453125,
            "energy_consumption": 0.03205629200752965,
            "crash_vehicle": false ,
            "crash_sidewalk": false ,
            "crash_on_broken_line": false ,
            "crash_on_continous_line": false ,
            "lateral_to_left": 0.0,
            "lateral_to_right": 1.0,
            "yaw_rate": 0.0,
            "velocity": 0.06896267731170751,
            "lat_pos_on_current_lane": 0.5112440321180556,
            "other_v_info": [
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0,
                  0.0

            ],
            "Simulink_data": {
                  "speed": 1.1286553099976474e-05,
                  "gear": 1.0,
                  "eng_power": 3.16257394967325e-21,
                  "eng_fuelcsp": 6.189980549848496e-07,
                  "impeller_speed": 1.4855953319636388,
                  "impeller_trq": 0.00016012586204318716,
                  "turbine_speed": 0.00045627177625058484,
                  "turbine_trq": -0.0003573334694081453,
                  "right_tire_slip": 1.7136982340528924e-08,
                  "left_tire_slip": 1.7136982340528924e-08,
                  "rear_axle_normal_force": 2746.800280732874
```

There are limitations regarding the computing resource and the Simulink Model. MetaDrive puts no restrictions on the number of vehicles. Nevertheless, sufficient computing resources must be served for MetaDrive and Simulink Model. A scenario can contain up to ten vehicles. The problem related to data generation from Simulink Model during short scenarios is mentioned as a limitation in previous paragraphs. Another limitation is that Simulink Model cannot generate realistic synthetic data after the crash. Because the inputs of the model are throttle and brake values. There is no other possibility to control the engine and speed just after the crash. A collision affects the speed of vehicles. This deceleration cannot be simulated by Simulink Model. However, many vehicle sensors fail and their data can be corrupted in real-life accidents.

7 Conclusion and Future Work

The application of IoT on vehicles maintains numerous advantages in safety, efficiency, and entertainment services. Distributed communication between multiple vehicles is defined as a "connected car" and plays a significant role in autonomously driving vehicles. The challenge is processing and storing vast data that is obtained from vehicles and the environment. In addition, synchronization must be granted to provide reliable data. Handling the data is vital for developing such a system and it should be tested in a simulation environment. This thesis firstly analyzed the requirements for a simulator for connected car scenarios. After the requirements were established, the architecture for the simulator was designed, and fundamental design decisions concerning technologies and standards were concluded. The simulator was realized by extending the MetaDrive. Diverse scenarios can be created by configuring vehicles and environments thanks to Graphical User Interface. A MATLAB script was developed, and Simulink Model was modified to generate extra synthetic data. MongoDB was preferred to store generated data. RabbitMQ was opted to share generated data with distributed systems and receive commands for vehicles from them. Multiple scenarios were produced using the GUI to evaluate the simulator. This thesis accomplished the goal of not only to developing a data generator for miscellaneous applications but also provided a simulation environment for diverse algorithms that processes data from vehicles and controls their behaviors.

To sum up, the requirement analyses, conceptual model design, and the simulator implementation were performed from scratch and a prototype for the simulator was developed. This simulator contains vital features and fulfills the requirements. If the computing resource limitation is not a case for further theses, the Simulator Component can be augmented with CARLA Simulator. It is more robust than the MetaDrive for data generation and supports environmental control. Also, the Data Generator component can be set on a distributed system to increase performance. Lastly, Simulink Model can be extended to provide more data and make the simulator more realistic.

Bibliography

[Air22] AirSim. *Welcome to AirSim*. https://microsoft.github.io/AirSim/, Last accessed on 2022-01-05. 2022 (cit. on p. 31).

[AKC+15] C. Aguero, N. Koenig, I. Chen, H. Boyer, S. Peters, J. Hsu, B. Gerkey, S. Paepcke, J. Rivero, J. Manzo, E. Krotkov, G. Pratt. "Inside the Virtual Robotics Challenge: Simulating Real-Time Robotic Disaster Response". In: *Automation Science and Engineering, IEEE Transactions on* 12.2 (Apr. 2015), pp. 494–506. ISSN: 1545-5955. DOI: 10.1109/TASE.2014.2368997 (cit. on p. 32).

[AKGT20] A. Afzal, D. S. Katz, C. L. Goues, C. S. Timperley. "A study on the challenges of using robotics simulators for testing". In: *arXiv preprint arXiv:2004.07368* (2020) (cit. on p. 22).

[Bri21] Britannica, T. Editors of Encyclopaedia. *Computer Simulation*. https://www.britannica.com/technology/computer-simulation, Last accessed on 2022-01-03. 2021 (cit. on p. 20).

[Bru13] Bruce D. Lightner. *MAP- and MAF-Based Air/Fuel Flow Calculator*. http://www.lightner.net/obd2guru/IMAP_AFcalc.html, Last accessed on 2022-01-05. 2013 (cit. on pp. 25, 27).

[BSB18] S. Boussoufa-Lahlah, F. Semchedine, L. Bouallouche-Medjkoune. "Geographic routing protocols for Vehicular Ad hoc NETworks (VANETs): A survey". In: *Vehicular Communications* 11 (2018), pp. 20–31. ISSN: 2214-2096. DOI: https://doi.org/10.1016/j.vehcom.2018.01.006. URL: https://www.sciencedirect.com/science/article/pii/S2214209616300183 (cit. on p. 17).

[Car] CarSim. *Introduction to CarSim*. https://www.carsim.com/downloads/pdf/CarSim_Introduction.pdf, Last accessed on 2022-01-14 (cit. on p. 35).

[CAR22a] CARLA. *CARLA Documentation*. https://carla.readthedocs.io/, Last accessed on 2022-01-06. 2022 (cit. on p. 33).

[CAR22b] CARLA. *ScenarioRunner*. https://carla-scenariorunner.readthedocs.io/en/latest/, Last accessed on 2022-01-05. 2022 (cit. on p. 32).

[CLLH20] P. Cai, Y. Lee, Y. Luo, D. Hsu. "Summit: A simulator for urban driving in massive mixed traffic". In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2020, pp. 4023–4029 (cit. on pp. 36, 38).

[co422] co4e. *Eclipse SUMO*. https://co4e.com/sumo, Last accessed on 2022-01-05. 2022 (cit. on p. 37).

[CSKX15] C. Chen, A. Seff, A. Kornhauser, J. Xiao. "Deepdriving: Learning affordance for direct perception in autonomous driving". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 2722–2730 (cit. on p. 35).

[Dec22] DecisionForce. *MetaDrive: Composing Diverse Driving Scenarios for Generalizable RL*. https://github.com/decisionforce/metadrive, Last accessed on 2022-04-05. 2022 (cit. on p. 76).

[DRC+17] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, V. Koltun. “CARLA: An open urban driving simulator”. In: *Conference on robot learning*. PMLR. 2017, pp. 1–16 (cit. on pp. 32, 33).

[Dur90] H. F. Durrant-Whyte. “Sensor models and multisensor integration”. In: *Autonomous robot vehicles*. Springer, 1990, pp. 73–89 (cit. on p. 23).

[Eng22] Engineers Edge. *Braking Torque Equation and Calculator*. https://www.engineersedge.com/mechanics_machines/braking_torque_13635.htm, Last accessed on 2022-02-04. 2022 (cit. on pp. 28, 63).

[FFD20] A. Fuller, Z. Fan, C. Day. “Digital Twin: Enabling Technologies, Challenges and Open Research”. In: (May 2020) (cit. on p. 20).

[FRST19] F. Farroni, M. Russo, A. Sakhnevych, F. Timpone. “TRT EVO: Advances in real-time thermodynamic tire modeling for vehicle dynamics simulations”. In: *Proceedings of the Institution of Mechanical Engineers, Part D: Journal of Automobile Engineering* 233.1 (2019), pp. 121–135 (cit. on p. 28).

[GM19] U. Gandhi, P. M K. “A survey on internet of vehicles: applications, technologies, challenges and opportunities”. In: *International Journal of Advanced Intelligence Paradigms* 12 (Jan. 2019), p. 98. DOI: 10.1504/IJAIP.2019.10017745 (cit. on p. 17).

[GMD09] A. Grundstein, V. Meentemeyer, J. Dowd. “Maximum vehicle cabin temperatures under different meteorological conditions”. In: *International journal of biometeorology* 53.3 (2009), pp. 255–261 (cit. on p. 27).

[Gri14] M. Grieves. “Digital twin: manufacturing excellence through virtual factory replication”. In: *White paper* 1 (2014), pp. 1–7 (cit. on p. 19).

[GS12] E. Glaessgen, D. Stargel. “The digital twin paradigm for future NASA and US Air Force vehicles”. In: *53rd AIAA/ASME/ASCE/AHS/ASC structures, structural dynamics and materials conference 20th AIAA/ASME/AHS adaptive structures conference 14th AIAA*. 2012, p. 1818 (cit. on p. 19).

[HAF21] A. Hbaieb, S. AYED, L. Fourati. “Internet of Vehicles and Connected Smart Vehicles Communication System Towards Autonomous Driving”. In: (May 2021). DOI: 10.21203/rs.3.rs-493419/v1 (cit. on p. 17).

[HGG03] J. Hesse, J. Gardner, W. Göpel. *Sensors for automotive technology*. Wiley-VCH Verlag Weinheim, 2003 (cit. on pp. 27, 29).

[IoT21] IoT-Analytics. *State of IoT 2021*. https://iot-analytics.com/number-connected-iot-devices, Last accessed on 2021-12-26. 2021 (cit. on p. 15).

[JOH11] JOHN VOELCKER. *It’s Official: We Now Have One Billion Vehicles On The Planet*. https://www.greencarreports.com/news/1065070_its-official-we-now-have-one-billion-vehicles-on-the-planet, Last accessed on 2021-12-28. 2011 (cit. on p. 17).

[KTTS21] P. Kaur, S. Taghavi, Z. Tian, W. Shi. “A survey on simulators for testing self-driving cars”. In: *2021 Fourth International Conference on Connected and Autonomous Driving (MetroCAD)*. IEEE. 2021, pp. 62–70 (cit. on pp. 22, 32–35).

[Kyl18] Kyle Wiggers. *Microsoft's open-source AirSim platform joins Unity*. https://venturebeat.com/2018/11/14/microsofts-open-source-airsim-platform-comes-to-unity/, Last accessed on 2022-01-05. 2018 (cit. on p. 31).

[LBB+18] P. A. Lopez, M. Behrisch, L. Bieker-Walz, J. Erdmann, Y.-P. Flötteröd, R. Hilbrich, L. Lücken, J. Rummel, P. Wagner, E. Wießner. "Microscopic Traffic Simulation using SUMO". In: *The 21st IEEE International Conference on Intelligent Transportation Systems*. IEEE, 2018. URL: https://elib.dlr.de/124092/ (cit. on p. 37).

[LPX+21] Q. Li, Z. Peng, Z. Xue, Q. Zhang, B. Zhou. "Metadrive: Composing diverse driving scenarios for generalizable reinforcement learning". In: *arXiv preprint arXiv:2109.12674* (2021) (cit. on pp. 36, 37).

[MAT22a] MATLAB. *Automated Driving Toolbox*. https://www.mathworks.com/products/automated-driving.html, Last accessed on 2022-01-05. 2022 (cit. on pp. 34, 35).

[MAT22b] MATLAB. *Call MATLAB from Python*. https://www.mathworks.com/help/matlab/matlab-engine-for-python.html, Last accessed on 2022-01-05. 2022 (cit. on p. 67).

[MAT22c] MATLAB. *Complete Vehicle Model*. https://www.mathworks.com/help/physmod/sdl/ug/about-the-complete-vehicle-model.html, Last accessed on 2022-02-16. 2022 (cit. on p. 49).

[MAT22d] MATLAB. *Conventional Vehicle Reference Application*. https://www.mathworks.com/help/autoblks/ug/conventional-vehicle-reference-application.html, Last accessed on 2022-02-16. 2022 (cit. on p. 47).

[MAT22e] MATLAB. *Longitudinal Driver*. https://www.mathworks.com/help/autoblks/ref/longitudinaldriver.html, Last accessed on 2022-01-05. 2022 (cit. on p. 48).

[Mer18] Mertens,Josephine. *Generating Data to Train a Deep Neural Network End-To-End within a Simulated Environment*. https://www.mi.fu-berlin.de/inf/groups/ag-ki/Theses/Completed-theses/Master_Diploma-theses/2018/Mertens/index.html, Last accessed on 2022-01-05. 2018 (cit. on p. 35).

[MQT22] MQTT. *MQTT*. https://mqtt.org/, Last accessed on 2022-03-05. 2022 (cit. on p. 45).

[Nat10] National Highway Traffic Safety Administration. *Vehicle-to-vehicle Communication*. https://www.nhtsa.gov/technology-innovation/vehicle-vehicle-communication, Last accessed on 2022-01-02. 2010 (cit. on p. 17).

[ND18] M. Nyman, M. D'Orto. *Validation of a vehicle motion planner in an open-source simulator*. 2018 (cit. on pp. 31, 32, 37).

[Nik21] S. I. Nikolenko. "Synthetic Simulated Environments". In: *Synthetic Data for Deep Learning*. Cham: Springer International Publishing, 2021, pp. 195–215. ISBN: 978-3-030-75178-4. DOI: 10.1007/978-3-030-75178-4_7. URL: https://doi.org/10.1007/978-3-030-75178-4_7 (cit. on pp. 20, 32).

[Pat20] K. K. Patel. "A simulation environment with reduced reality gap for testing autonomous vehicles". PhD thesis. University of Windsor (Canada), 2020 (cit. on pp. 20, 21).

[PR12] J. L. Pereira, R. J. Rossetti. "An integrated architecture for autonomous vehicles simulation". In: *Proceedings of the 27th annual ACM symposium on applied computing*. 2012, pp. 286–292 (cit. on pp. 20, 37).

[Pri] Priya Pedamkar. *ZeroMQ vs RabbitMQ*. https://www.educba.com/zeromq-vs-rabbitmq/, Last accessed on 2022-01-06 (cit. on p. 45).

[Rab22] RabbitMQ. *What can RabbitMQ do for you?* https://www.rabbitmq.com/features.html, Last accessed on 2022-03-06. 2022 (cit. on p. 45).

[Rib17] W. Ribbens. *Understanding automotive electronics: an engineering perspective*. Butterworth-heinemann, 2017 (cit. on pp. 23–25).

[RST+20] G. Rong, B. H. Shin, H. Tabatabaee, Q. Lu, S. Lemke, M. Možeiko, E. Boise, G. Uhm, M. Gerow, S. Mehta, et al. “Lgsvl simulator: A high fidelity simulator for autonomous driving”. In: *2020 IEEE 23rd International conference on intelligent transportation systems (ITSC)*. IEEE. 2020, pp. 1–6 (cit. on pp. 33, 34).

[SDLK18] S. Shah, D. Dey, C. Lovett, A. Kapoor. “Airsim: High-fidelity visual and physical simulation for autonomous vehicles”. In: *Field and service robotics*. Springer. 2018, pp. 621–635 (cit. on p. 31).

[Shi15] Shibashis Ghosh. *How to calculate vehicle speed from engine RPM and vice versa.* , Last accessed on 2022-02-2. 2015 (cit. on p. 25).

[SJP18] H. S. Song, S. P. Jung, T. W. Park. “Simulation of temperature rise within a rolling tire by using FE analysis”. In: *Journal of Mechanical Science and Technology* 32.7 (2018), pp. 3419–3425 (cit. on p. 28).

[SK21] S. Sharma, B. Kaushik. “A survey on nature-inspired algorithms and its applications in the Internet of Vehicles”. In: *International Journal of Communication Systems* 34 (Aug. 2021). DOI: 10.1002/dac.4895 (cit. on p. 17).

[SMS+20] B. Schlager, S. Muckenhuber, S. Schmidt, H. Holzer, R. Rott, F. M. Maier, K. Saad, M. Kirchengast, G. Stettinger, D. Watzenig, et al. “State-of-the-art sensor models for virtual testing of advanced driver assistance systems/autonomous driving functions”. In: *SAE International Journal of Connected and Automated Vehicles* 3.12-03-03-0018 (2020), pp. 233–261 (cit. on pp. 23–25).

[Sta17] Statisca. *Number of passenger cars and commercial vehicles in use worldwide from 2006 to 2015 in*. https://www.statista.com/statistics/200002/international-car-sales-since-1990, Last accessed on 2021-12-28. 2017 (cit. on p. 17).

[Sta21] Statisca. *Number of cars sold worldwide between 2010 and 2021*. https://www.statista.com/statistics/200002/international-car-sales-since-1990, Last accessed on 2021-12-26. 2021 (cit. on p. 15).

[STM18] M. Sadiku, M. Tembely, S. Musa. “INTERNET OF VEHICLES: AN INTRODUCTION”. In: *International Journal of Advanced Research in Computer Science and Software Engineering* 8 (Feb. 2018), p. 11. DOI: 10.23956/ijarcsse.v8i1.512 (cit. on p. 3).

[SUM22] SUMMIT. *https://adacompnus.github.io/summit-docs/*. https://adacompnus.github.io/summit-docs/, Last accessed on 2022-01-16. 2022 (cit. on p. 38).

[Tom20] Tom Harbid. *A Guide on Everything You Need to Know About Sensors in Cars*. https://www.cashcarsbuyer.com/sensors-in-cars/, Last accessed on 2022-01-25. 2020 (cit. on pp. 23, 25).

[Tul17] Tully Foote. *Simulated Car Demo*. https://www.osrfoundation.org/simulated-car-demo/, Last accessed on 2022-01-05. 2017 (cit. on p. 32).

[Wym20] Wymann,Bernhard. *TORCS - The Open Racing Car Simulator*. https://sourceforge.net/projects/torcs/, Last accessed on 2022-01-05. 2020 (cit. on p. 35).

[YLLW17] F. Yang, J. Li, T. Lei, S. Wang. "Architecture and key technologies for Internet of Vehicles: a survey". In: *Journal of Communications and Information Networks* 2 (June 2017). DOI: 10.1007/s41650-017-0018-6 (cit. on pp. 18, 19).

[YSBM00] K. Yoo, K. Simpson, M. Bell, S. Majkowski. "An engine coolant temperature model and application for cooling system diagnosis". In: *SAE transactions* (2000), pp. 950–960 (cit. on pp. 25, 26).

[YXM+21] G. Yang, Y. Xue, L. Meng, P. Wang, Y. Shi, Q. Yang, Q. Dong. "Survey on Autonomous Vehicle Simulation Platforms". In: *2021 8th International Conference on Dependable Systems and Their Applications (DSA)*. IEEE. 2021, pp. 692–699 (cit. on pp. 21, 22, 37).

Declaration