Master's thesis

Prototyping Connection Between Digital Twin and Physical Twin for Autonomous Driving to Support Experimentation.

Øyvind Soma

Thesis submitted for the degree of
Master in Informatics: Robotics and Intelligent Systems
30 credits

Institute For Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2021

Master's thesis

Prototyping Connection Between Digital Twin and Physical Twin for Autonomous Driving to Support Experimentation.

Øyvind Soma

Abstract

This thesis will focus on developing a Digital Twin prototype for an autonomous vehicle. The goal for the project was to discover how OpenModelica and Carla can be connected and configured to serve as a Digital Twin. It was acquired two methodologies of connecting OpenModelica models to a Digital Twin. The first method got used to simulate a mechanical model with historical data to re-simulate the scenarios. Further, the second method got used to simulate a mathematical model to add a feature from the physical environment to the Digital Twin. To begin the Digital Twin configuration, Carla got used to generating the states representing the forward speed and acceleration for a vehicle so the OpenModelica models could get tested.

It was looked at possibilities to extend the configuration to use Carla as the visual simulation tool to combine OpenModelica and Carla as the Digital Twin. This thesis designed one configuration, which introduced the concept of having the Digital Twin one step in front of the Physical Twin. The LGSVL-simulator was introduced as the Physical Twin to generate similar parameters as a vehicle situated in a physical environment to test this configuration. The project did not explore methods to integrating the same 3D environment in Carla and the LGSVL-simulate. Which led to that the experiments while the LGSVL-simulator served as a Physical Twin got limited. The configuration between Carla and OpenModelica could not explore the concept of returning an action to the Physical Twin. So, a second configuration diagram is presented in the thesis, based on the knowledge learned in this project, which possibly can be further researched.

The thesis concludes with that the methodologies acquired throughout the project present a good base for further research related to developing a Digital Twin by combining Carla and OpenModelica.

Acknowledgements

I want to express my special thanks of gratitude to my supervisors Tao Yue and Shaukat Ali. I have learned a lot about Digital Twin technology by working on this project, and my interest in the technology has increased. I was able to gain hands-on experience with the technology, which was my personal goal for the master project. Thank you for all the encouragement throughout the project and the support.

I would like to extend my sincere thanks to my friends and fellow students Daniel Woldegirogis, Sjamil Gazimagamaev, Eivind Brastad Dammen and Song Luu. It has been a pleasure sharing the same goal of pursuing a master's degree with you.

I am also grateful to Chengjie Lu for giving me access to his restful API.

Special thanks to Massimo Ceraolo for designing the DragF block and sharing his material as open-source.

I also want to thank the creators of both Carla and OpenModelica for creating such good tools available for open-source research.

Contents

List of Figures

List of Tables

Abbreviations

DT	Digital Twin
PT	Physical Twin
DM	Digital Model
OM	OpenModelica
DS	Digital Shadow
EV	electric vehicle
RQ	research questions
AV	autonomous vehicle
AS	autonomous system

Chapter 1

Introduction

This chapter contains of five parts, the chapters starts with a description of the motivation for the project.Then the delimitations for the project is set and the research questions (RQ) gets introduced. Further, the approach of how the research questions got resolved through the project is explained. At last, the outline of the thesis is presented.

1.1 Motivation

Digital Twin(DT) technology is an evolving technology, and the potential and the different use cases of the technology are being explored in both the academia and in the industry. One of the use cases can be in autonomous systems (AS) like autonomous vehicles (AV). The power of DT technology in AS is that it is possible to monitor data, control actions, and predict future outcomes. The DT can gather data in real-time, learn from or classify this data through simulations, calculations, or learning algorithms, and then implement solutions back to the physical environment. DT technology can also contribute to developing more advanced AS, because it is possible to gather data from real scenarios and further use this data to develop the systems and improve the systems throughout it's life-cycle.[13][7].

Another aspect of DT technology in AS is the integration of humans and technology. One example can be the moral dilemma, where an AV must choose between two actions, where both outcomes put humans at risk. With a DT, it could be possible to simulate different scenarios and predict different outcomes before each state is returned to the PE. If the AV could track the angle of the foot and the predicted walking speed of the pedestrians with data from cameras and sensors the DT could use this data to simulate possible outcomes to find out where the pedestrian will be located in a moment of time, with a possible high accuracy. With this method, the DT may present possible outcomes that put no human at risk. This is an example of the potential this technology can have, and the future potential for DT technology is the primary motivation of choosing this topic for me as the writer to gain fundamental understanding and hands-on experience with the technology.

Before the potential of the technology can become a reality, there has to be developed a connection between the digital environment and the physical environment. This connection makes it possible to fetch data from the physical environment and use it as input data to the digital environment, which can be done with different modeling and simulation tools. In this project, the research will combine two Open-Source tools called Carla and OpenModelica (OM), to develop a prototype for a DT. This is also a motivation for me as the writer because I want to participate in Open-Source projects as my technical skills develop.

1.2 Project description

This thesis is being written with guidance from Simula Research Laboratory and a collaborating university in China called Nanjing University of Aeronautics and Astronautics. They have access to Mars Rovers at this university, and they are currently developing DTs for these Mars Rovers. The work done in this thesis will hopefully contribute to the development of the DTs that is being developed in the lab at Nanjing University. The Mars Rovers is located in China, so the prototype that is being developed will not be connected to a physical system in this project. In addition the connection between OM and Carla has to be configured to serve as a DT before it can be connected to a physical system. However, the concept of connecting the prototype that is being developed in this project to a Physical Twin (PT) will be tested with a restAPI which is designed by Chengjie Lu at the Nanjing University. The restAPI returns states that represents the movements of a vehicle that is situated in the LGSV-simulator. The returned states will be used as input data to the DT when the connection between Carla and OpenModelica is designed.

1.3 Delimitations

Digital Twin technology covers many fields[3], so this thesis will focus on prototyping the connection of a DT and PT for autonomous driving. There had to be set delimitations to keep the project within the time line of one semester (30 credits).

The delimitations are:

- ⇒ The DT that are going to be developed in this project is a prototype.
- ⇒ This project will mainly focus on the connection between Carla and OM, and how these tools can be combined as a DT.
- ⇒ The parameters transmitted between OpenModelica and Carla will be the acceleration and the forward speed to configure the connection for the prototype.
- ⇒ The models in OM will not include all the mechanical parts of a AV nor any of the sensors, there will only be used simple models to represent the forward movement of the vehicle.
- ⇒ In the configuration where OM and Carla gets connected to the restAPI, the 3D environment in Carla will not be a replica of the environment from the restAPI.

1.4 Research Questions

When the delimitations were set, the research questions for the project could be established, which are illustrated in table 1.1.

Research Questions	
RQ1	How can OpenModelica be connected to the Digital Twin by adding mechanical- or mathematical models to the digital environment?
RQ2	How can Carla and OpenModelica be configured to serve as a Digital Twin?

Table 1.1: This table shows the research questions that were established for the project.

1.5 Approach

The approach to respond to the research questions will be explained in this section. The approach is illustrated in Figure 1.1, so the reader can get an overview of the chapters attached to the RQs.

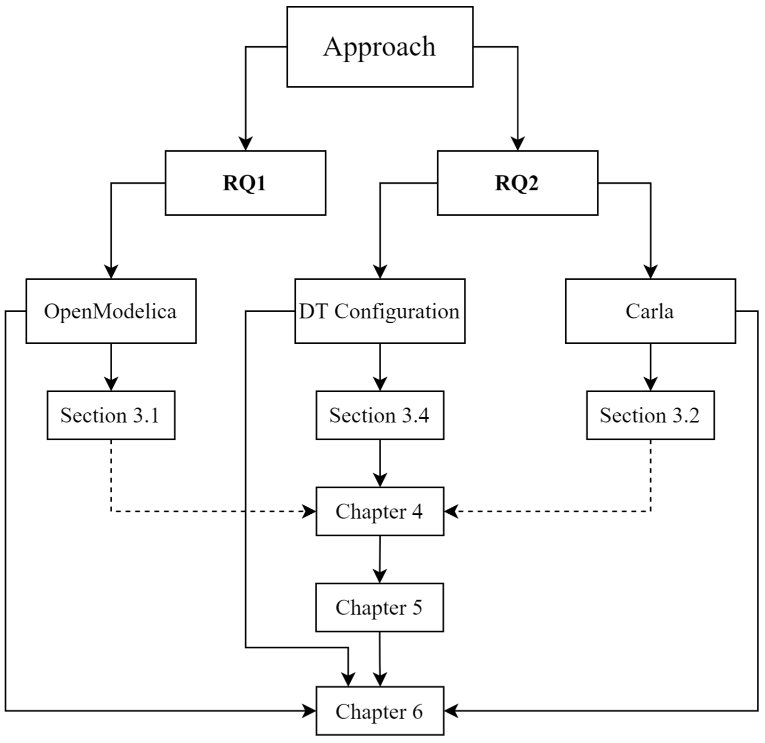


Figure 1.1: The approach of how the RQs will follow the thesis

1. To respond to RQ1, OpenModelica had to fulfill specific requirements. The requirements are listed below.

   ⇒ OM has to add a feature to Carla, to make the connection between the tools in the DT useful.

   ⇒ There have to be methods of bidirectionally working with data in the OM environment by using Python.

   The methodology of how these requirements got resolved is discussed in section 3.1, because this section explains the methods of how the tool got connected to the DT. Further, in section chapter 6, the pros and cons for the tool will be clarified, together with a discussion regarding RQ1.

2. In order to respond to RQ2, there were also set a requirement for Carla, which is listed below.

   ⇒ There have to be methods of bidirectionally working with data that describes the vehicle in Carla by using Python to connect it to the DT.

   The methodology of how Carla got connected to the DT will be discussed in section 3.2, together with the functions that are used in Carla to fulfill the requirement. Further, in section 3.4 the DT configuration between Carla and OpenModelica will be presented, with an explanation regarding the methodology for the configuration, and the configurations will be designed in chapter 4. Finally, in chapter 6 there will be a discussion that responds to RQ2, regarding the configuration that were developed for the prototype and how it possibly can be further developed.

1.6 Outline of thesis

Chapter 1 gives the reader a introduction to the project together with a explanation on how the three research questions that where set will be resolved throughout the thesis.

Chapter 2 gives the reader the necessary background theory that was needed in the development of the prototype of the DT.

Chapter 3 presents the initializing phase of each tool, together with the methodology of how each tool got connected to the project togheter with the configuration that where set for the DT and the PT.

Chapter 4 contains the designing phase of the DT, this chapter combines all the methodologies from chapter 3 to create the prototype for the DT.

Chapter 5 addressees the experiments that where done after the prototype were designed.

Chapter 6 contains the discussion concerning the RQs and the assessment of methodologies.

Chapter 2

Background and Theory

This chapter begins by explaining the background of Digital Twin technology that was studied throughout the project. The technology is evolving, so there were mainly looked into literature reviews that set the guideline for further research related to the technology. There will also be a short section related to ongoing work in the industry and academia. Further, the tools that were used in the project will be introduced. Lastly, the drag force equation will be presented, because this equation can add features from the physical environment to the digital environment.

2.1 Digital Twin Technology

There is no agreed-upon definition in the industry, or the academia of what a DT is [14][15][13], which may be because many of the technologies that are involved in DTs are also evolving technologies, such as Artificial Intelligence, the Internet of Things, and Cyber-Physical Systems. The real potential of DT technology is if it is combined with other technologies like the ones just mentioned. So, there will not be included a definition of the technology in this thesis. However, the technology will be viewed as a digital representation of a physical system with bidirectionally data exchange between the digital environment and the physcial environment[12][14][13][2], as illustrated in figure 2.1.

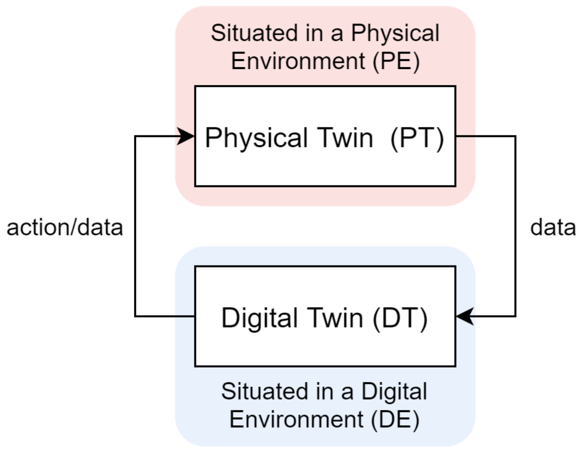

Situated in a Physical Environment (PE)
Physical Twin (PT)
action/data
data
Digital Twin (DT)
Situated in a Digital Environment (DE)

Figure 2.1: This figure illustrates a bidirectionally data exchange between the PT and the DT.

There is some misconception of what a DT is that are addressed by Fuller et al. in [12]. The two misconceptions that are mentioned in this article are Digital Models (DM), and Digitla Shadows (DS) [12]. The DM is a digital version of a PT without any data exchange between the physical- and the digitla environment[12]; this may be a good approach if the goal is to re-design a physical system. The DS can be viewed as a more advanced DM because changes in the physical environment will update the model that the DS represents[12]; this can be a good approach if the goal is to monitor a physical system. Anyhow, these two models do not got bidirectional data exchange between the physical- and the digital environment, and that is the main feature of a DT[12][14][13][2]. There are also misconceptions that a DT needs to become a PTs exact 3D model[12]. One of the features of Carla in this project is to represent the 3D environment of the DT, but the technology should not be limited to including 3D models. The data in the digital environment can be illustrated with other methods like, for example, excel sheets or graphs, which depends on the task and the reason for the development of the DT.

In this project, the DT will be developed for autonomous driving, so a 3D model will benefit the DT. This is because if it is possible to create a 3D environment of the physical environment that the PT is situated in, the DT can simulate scenarios in the same environment, and this is a good feature while working with DTs for fast-phased physical systems. By including 3D models to the DT, the 3D models should be separated from the mechanical or the mathematical models of the DT, because the 3D models should be able to be reused by different DT models[24]. This is the reason for this project is based on two different tools OM and Carla. Carla will be mainly used as the 3D model, and OM will be used to design the mathematical and/or mechanical models for the DT. It should be possible to design different models in OpenModelica for different use cases for the DT system, which will be explained further in section 3.1. The idea of implementing several models to the DT was addressed by Yue, Arcaini and Ali in [24], where they pointed out that Golomb in [1] suggested that: 'Don't limit yourself to a single model, more than one may be useful for understanding different aspects of the same phenomenon'.

While developing a DT for a AV, it is essential to consider the connection between the PT and the DT. This is because the physical environment is fast phased, and there will be quick changes in the environment that the PT is situated in. Two possible configurations can be set between the twins, and these are PT-To-DT and DT-To-PT[24][13]. In a PT-To-DT configuration, the PT will feed the DT with input data, and in a DT-To-PT configuration, the DT will feed the PT with input data and possibly deploy an action to the DT[24][13]. Again, this configuration depends on why the DT are being developed, but a combination of these two configurations can be suitable in for autonomous driving. Let us say that it is one DT model that tracks the movement of an AV. If the AV is moving at a steady phase and there are no rapid changes in the physical environment, a PT-To-DT configuration could be a good implementation. While this AV is moving at a steady phase, let us imagine that there suddenly is a moose that walks into the road. This can be a potentially dangerous situation, so the DT system should automatically change to a DT-To-PT configuration in situations like this, or there could be implemented another DT model with a PT-To-DT configuration that kicks in if the systems track a potentially dangerous situation. A DT-To-PT configuration could benefit the AV in situations like this because the DT could simulate several scenarios to locate a scenario where the AV avoids the moose. The concept of having the DT one step ahead of the PT can allow the DT to perform real-time 'what if?' analyses, and that is an interesting feature of DT technology. In this project, a PT-To-DT configuration will only be implemented, where the restAPI is the PT and OM, together with Carla, is the DT. So, this project will not explore having the DT one step ahead of the PT because this will require more knowledge, research, and time as addressed in section 1.2. There is a characteristic in DT technology that is called twinning rate, and this rate describes how frequently the data exchange between the PT and the DT is updated[24][13]. In this prototype, the twinning rate will be based on how often the DT get updated in the loop that connects it to the PT, so the twinning rate will be represented by iterations.

2.2 Ongoing work in the industry and academia

The interest in DT technology has had growth in both the industry and academia the last couple of years[13]. The concept of developing DTs for PTs where introduced by Grieves in 2003[13][2]. There has been an increase in research in academia the last couple of years, and Jones et al. published an article in 2020, were they did a literature review on 96 papers related to DT technology from 2009 till 2018.[13] This systematic literature review gathers all these articles to set a guideline for further research. So, the topic is still evolving, and there is much research related to DT technology these days; the European Space Agency are even developing a DT for the earth[24][16]. There are also ongoing projects were big technology companies team up with Formula One teams like Ferrari and Mac Lauren for case studies and the development of DTs. The concept of DT technology with human interaction has been in the Formula One industry for a long time, where there is a team that monitors hundreds of different sensors of the car and communicates with the driver throughout the race. There is possible a lot regarding DT technology for AVs that can be learned by Formula One engineers. A case study article is available online regarding a project with Palantir and Ferrari, but there is no information available about their findings that could inspire this project.

2.3 Tools

This section will present a short introduction to the different tools that will be used in this project. Table 2.1 is designed by Fuller et al. in [12], and the table illustrated the enabling technologies for DT systems[12]. This project will focus on the Application Domain from table 2.1 to start developing the prototype. Each tool will be presented in this section and a description that matches with the enabling technologies from the application domain.

Domain	**Enabling Technology**
Application Domain	Model Architecture and Visualisation
	Software and APIs
	Data collection and Pre-processing
Middleware domain	Storage Technology
	Data Processing
Networking domain	Communication Technology
	Wireless Communication
Object Domain	Hardware Platform
	Sensor Technology

Table 2.1: Aidan Fuller, Zhong Fan, Charles Day, and Chris Barlow. DT: Enabling technologies, challenges and open research. IEEE Access, 8:108952-108971, 2020. Table 4, Enabling Technologies and Functional Blocks: DT; p. 12.[12]

OpenModelica - Digital Twin

OpenModelica is an Open-Source modeling tool that is based on the coding language Modelica. The tool got an integrated Modelica library which gives the user possibilities of modeling physical systems. The Modelica library gives access to pre-designed blocks that can be combined to model electrical, magnetic, mechanical, fluid, and mathematical models. These models can possibly be used to add features from the PE to the DE, which will be further researched in this project. It is also possible to design blocks from scratch with the coding language Modelica, and there are available models that the Open-Source community has designed that can further be integrated into projects like this. OM also got a Python API, which is convenient when the tool will be connected with other APIs in the DT system by using Python.

Carla - Digital Twin

Carla is also an open-source tool that is powered by Unreal Engine, which was mainly developed for the open-source research for AVs[6]. This tool can be convenient to used as a 3D model for the DT because the infrastructure for spawning cars, maps, and fetch information from the Carla environment is already designed. There is also possible to connect the tool to a Python API, which will be explored to connect Carla to the DT.

restAPI - Nanjing University - Physical Twin

The restAPI is designed by Chengjie Lu at the Nanjing University. The restAPI is designed, so it is possible to return states from the LGSVL-simulator. This simulator is a similar tool to Carla. Even though the restAPI for the LGSVL-simulator does not represent a PT, it is still possible to return parameters that give similar information as a vehicle in a phsyical environment would. The server for the restAPI is located in China, and the 'requests' library in Python will be used to return the states from the API.

2.4 The drag force equation

The drag force equation is an example of a mathematical equation that can be added to the digital environment when developing a DT for a vehicle. Ceraolo in [5] have designed a OM block called *dragF*, that simulates equation 2.1. The *dragF* block will be used throughout the project to model the vehicles resistance to movement, which will be an essential factor in correctly representing the vehicle in a digital environment. There will be a short introduction to the different parameters in this equation in table 2.2. The car that are going to be spawned in Carla is a Tesla Model 3, so the mass, cross-sectional vehicle area, and the longitudinal drag coefficient are set to represent this car. The cross-sectional vehicle area is 2.22 meters, and the drag coefficient is 0.23 [10]. The rolling friction is affected by the wetness of the road and the tires of the vehicle[4]. In this project, the rolling friction parameter will be set to values that are inspired by Ejsmont et al. findings in [4] for the prototyping of how this parameter can add changes to the digital environment. The air density is set to be the average density calculated by the International Union of Pure and Applied Chemistry (IUPAC), which is 0.001225 g/cm^3.

$$R = fmg + \frac{1}{2}\rho S C_x V^2 \tag{2.1}$$

Parameter	**Description**	**Value**
f	Rolling friction	0-0.03
m	Mass of the vehicle	1600 kg
g	Gravity	9.807 m/s^2
S	Cross-sectional vehicle area	2.22 m
Cx	Longitudinal drag coefficient	0.23
V	Vehicle speed	This is a variable
ρ	The air density	0.001225 g/cm^3

Table 2.2: This table shows the parameters that are included in the drag force equation.

Chapter 3

Methods

This chapter explains the methodologies that have been acquired to connect the three different tools that is included in this project together in a Python environment, as Python is the coding language that serve as a link between OM, Carla and the restAPI. This chapter will also introduce the initializing phase of each tool together with the functions that where used to transmit data bidirectionally between the tools. The chapter will begin by presenting the methodologies that were used in OM to connect it to the DT. Further, the methodologies to connect Carla and the functions that were used with the Carla Python API will be presented. Then the methodology of how the prototype of the DT will be connected to the restAPI that represent the PT is explained. Finally, in section 3.4, there will be presented a diagram that shows the configuration between the tools, which will be used to design the prototype of the DT.

3.1 OpenModelica

This section will look at the various methods and models that were used with OM to design the prototype. In 3.1.1 there will be presented a mechanical model and in 3.1.2 there will be presented a mathematical model. The mechanical model is going to be simulated with historical data, and the mathematical model are going to be simulated parallelly with Carla. Further, in 3.1.3, there will be introduced two different methods of simulating these models with input data from outside the OM environment. At last, in section 3.4, the methodology of using the OM Python API will be explained.

3.1.1 Simple Car Model

While learning how to use the tool OM, there were looked at some open source material and exercises in the OM web book *Introduction to Modelica with Examples in Modeling, Technology, and Applications*[8]. In this web book there is a chapter called *Simplified Modelling of Electric and Hybrid Vehicles*, this chapter introduces a simplified electric vehicle (EV) model called *My first EV model* which is designed by Ceraolo[5]. The material is under the Creative Commons Attribution-ShareAlike 4.0 International License as addressed in [8]. One of the delimitation in this project was not to design an advanced OM model, so *My first EV model*[5] could be a good model to integrate into this project to start testing and exploring the potential of combining Carla and OM as a DT. There were looked at opportunities to extend the *My firs EV model*[5] by adding some more mechanical parts. While getting more familiar with the tool, it got decided that there were not going to be built a new mechanical model if it is not possible to track the wear damages of the mechanical parts. This is because from the point of view in this thesis, the DT should be able to monitor these different mechanical parts, and predict or alert when the parts should be changed. To add this feature to the mechanical blocks that is available in the OM-library, it requires research and knowledge in the Modelica coding language that is out of reach regarding this projects timeline. With that in mind, it got decided that it will not be applied any changes to *My first EV model*[5], but it can still be an interesting model to use in this project. The reason for this is because the input data to the model is a text file that holds a table with a driving cycle, and the DT should have the possibility of working with historical data to re-simulate scenarios. The concept of saving historical data from a PT, and further use that as input into the OM environment is a good method to get the fundamental configuration for the prototype started. So, *My first EV model*[5] that is illustrated in figure 3.1 will be connected to the prototype to further explore this concept. The only change that are going to be applied to *My first EV model*[5] is that it will be renamed to the Simple Car Model as a more suitable name for this project.

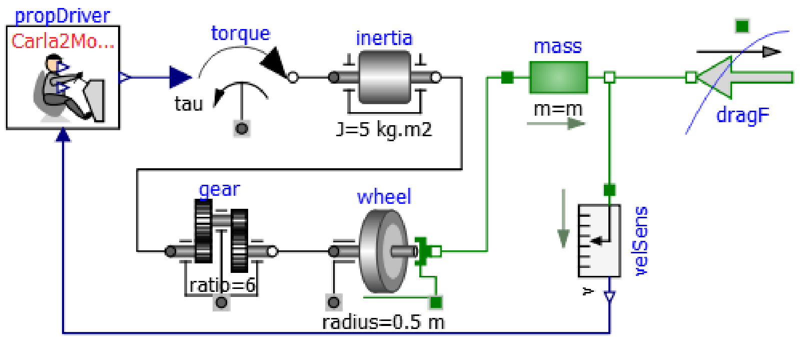


Figure 3.1: Massimo Ceraolo, Simplified Modelling of Electric and Hybrid Vehicles. Version 2017-09-25. Figure 1, A first, very simple, EV Model; http://omwebbook.OM.org/SMEHV. Note: This figure is different from the figure in the chapter, but the model is the same. This figure illustrates the OM model that will be used to simulate historical data from Carla.

The *PropDriver* block is designed by Ceraolo in [5], the block gets it input data over a driving cycle that is saved in a text file. This input data goes into a proportional controller that controls the torque. The velocity sensor called velSens, measures the velocity at the output of the model, so this is the parameter that is in the feedback loop for the proportional controller.[5] The *PropDriver* block will accelerate or brake by changing the output value for the torque dependent on the difference between the data that is fed from the text file and the measured feedback speed[5]. The torque is a measure for the rotational force that will be applied to the inertia block, to generate the power train[5]. The gear is an ideal gear, so the gear ratio is fixed and the wheel is an ideal rolling wheel, which means that the wheel do not have friction or inertia. The mass block represents the mass of the vehicle, this mass will gain a forward speed that is dependent on the rotational force of the torque. The last block is called *dragF* which is also designed by Ceraolo in [5], this block represents the resistance to movement of the vehicle by simulating the drag force equation[5]. There are several parameters that can be tuned in this block, these parameters are: vehicle mass, air density, vehicle cross area, rolling friction and aerodynamic drag coefficient as addressed in section 2.4.

3.1.2 Drag Force Model

This model got designed based on the *dragF* block that is designed by Ceraolo in [5]. The thought behind this model was to only work with the forces in the OM environment without including the mechanical parts, because the forward force of the vehicle is the output of all the mechanical parts in the model combined. This project will not focus on designing a digital replica of a vehicle in OM to generate this force, so the idea is to use the forward force from the vehicle in Carla as input to the OM model. With this approach it is possible to represent the forward force of the vehicle in both Carla and OM with the same parameters, and simulate them parallelly. Another reason for why this model got introduced is because when the wetness increases in Carla, it only affects the RGB sensor[20] and wetness of the road surface can increase the rolling friction[4]. In the *dragF* block it is possible to change the rolling friction[5], so it may be possible to add the vehicles resistance to movement that this block generates back to the Carla environment. Figure 3.2 is an illustration of the Drag Force Model, were start_vel and start_acc are the inputs from Carla. The blocks start_vel and start_acc is connected to the to the mass initialization parameters, which will be further explained in section 3.1.3 (Method 2). The force applied to the mass block will be subtracted by the *dragF* block, and the updated speed of the mass block that represents the vehicle will be returned to Carla.

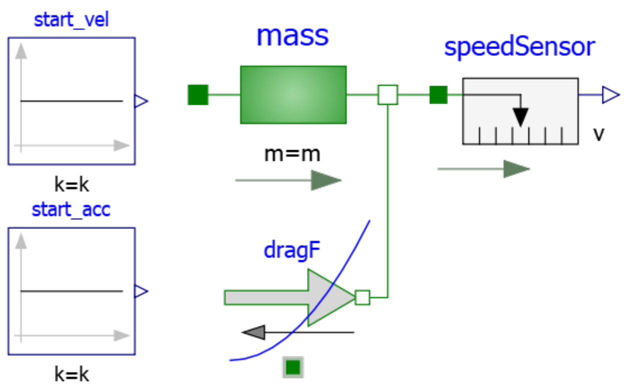


Figure 3.2: The dragF block is designed by Massimo Ceraolo in *Simplified Modelling of Electric and Hybrid Vehicles*[5]. This is figure is an illustration of the OpenModelica model that will be simulated parallelly with Carla.

3.1.3 Input data

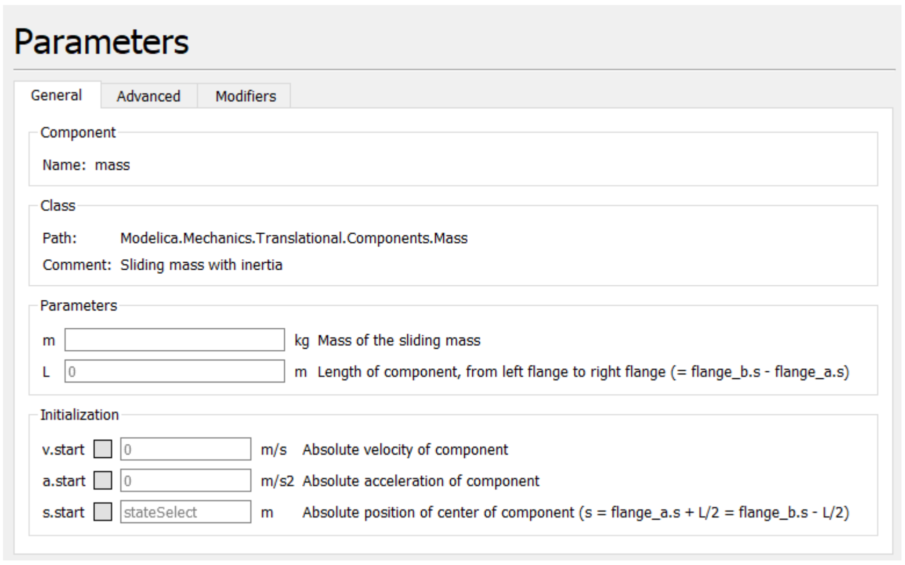
There will be introduced two different methods of working with input data in OM. Method 1 is introduced because the Simple Car Model will be simulated with historical data from the PT, and Method 2 is introduced so it is possible to simulate Carla and the Drag Force Model parallelly.

Method 1: Read data from a table

The OM block *Modelica.Blocks.Sources.CombiTimeTable* can access tables from text files[21]. This block is connected to the *propDriver* which is designed by Ceraolo in [5]. So, by saving data in a text file that the *CombiTimeTable* can access, it will be possible to re-simulate historical data. For the tables to be portable when using the OM Python API, the tables have to be saved in a resource folder in a OM library. How this library is created will be explained in 3.1.4, where OM Python API is being introduced. For OM to understand the information in the text file, the tables in text files have to be on a specific form[21], so this has to be kept in mind when the script that writes to this file is being designed in section 4.1.

Method 2: set parameters with OM python API

The different blocks in OM usually got initialization parameters and parameters. The parameters can be changed through the OM Python API, an example of how the layout looks and how these parameters can be changed in the OM environment is illustrated in figure 3.3. This figure represents the parameters of the mass block, the parameters m and L can be directly changed with the set function from line 2 in listing 3.1[23].



Parameters

General | Advanced | Modifiers

Component

Name: mass

Class

Path: Modelica.Mechanics.Translational.Components.Mass

Comment: Sliding mass with inertia

Parameters

m		kg	Mass of the sliding mass
L	0	m	Length of component, from left flange to right flange (= flange_b.s - flange_a.s)

Initialization

v.start	☐	0	m/s	Absolute velocity of component
a.start	☐	0	m/s2	Absolute acceleration of component
s.start	☐	stateSelect	m	Absolute position of center of component (s = flange_a.s + L/2 = flange_b.s - L/2)

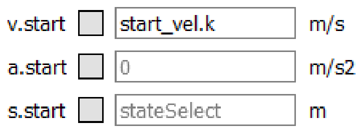
Figure 3.3: This figure shows the parameter window of the OM block Modelica.Mechanics.Translational.Components.Mass.

```
        m = 1700 #Vehicle mass
        mod.setParameters("mass.m{}".format(m))

```

Listing 3.1: The mod.setParameters function is accessed through the OM Python API[23] to set the parameters.

To change the initialization parameters there had to be used a different approach. If the goal is to change v.start from figure 3.3, then it is possible to create a constant block called start_vel that holds the parameter k. This constant block can then be connected to the initialization parameter as shown in figure 3.5. It is then possible to use the set function from listing 3.1 to change the parameter start_vel.k.



v.start	☐	start_vel.k	m/s
a.start	☐	0	m/s2
s.start	☐	stateSelect	m

Figure 3.4: This figure is an example on how the initialization parameter to the block Modelica.Mechanics.Translational.Components.Mass can be changed with a constant block in OM.

3.1.4 OM Python API

To be able to link OM with Carla and the restAPI, there had to be acquired methods of working with the OM Python API. The first step to be able to work with the OM Python API, is to create a library, so this library can be loaded to the Python environment[23]. The tables that are going to be used to simulate historical data have to be saved in a text file in this library to be portable. There were a few steps to create this library, and the method to create the library were found in Adrian Pop's *OMExamples* github[11].

1. The first step is to create a new folder for the library in the OM environment.

2. Inside this folder the resources/data/tables folders can be created, so the text files that holds the tables that represents the historical data can be saved there.

3. To create the library there has to be created a file called package.mo inside the head folder, the path should be set to the library name.

4. Inside package.mo there have to be written Modelica code to create the library, the code can be seen in seen in listing 3.2. Modelica has to be imported as shown in line 8, to be able to use the OM libraries to access the different pre-made blocks in OM.

```
1           within;
2           package DigitalTwinLibrary
3           "The name of the library in this project is
      DigitalTwinLibrary"
4               extends Modelica.Icons.Package;
5               "Customize the icon to the library"
6           annotation (preferredView = "info",
7                       uses(Modelica(version="3.2.3")));
8                       "Include Modelica to
9                       access the OM libraries"
10
11          end DigitalTwinLibrary;
12
```

Listing 3.2: This script shows how to create a library in OM, as presented in the package.mo file that is designed by Adrian Pop in the *FMUResourceExample* library from the *OMExamples* github[11]

5. To be able to create a package inside this library it has to be opened a "New Modelica Class", and in the text-view of this class, the Modelica code that is shown in Listing 3.3 has to be implemented.

```
        within DigitalTwinLibrary;
        package CarlaTwin
        "the Simple Car Model and the Drag Force Model will be inside this package"
        end CarlaTwin;

```

Listing 3.3: Example of how to create a new package in OpenModelica

6. When the library is created, the OM model can be connected to the Python script as shown in Listing 3.4.

```
        from OMPython import ModelicaSystem
        model_path = "C:/Program Files/OM1.16.1-64bit/lib/omlibrary/DigitalTwinLibrary/package.mo"
        #Access the Drag Force Model from the OM Library
        mod = ModelicaSystem(model_path, "DigitalTwinLibrary.CarlaTwin.Model2")

```

Listing 3.4: This script is from the OM Python API user-guide[23].

The OM Python API got several functions that can be used when the API is connected[23], the different functions that are used in this project together with their use cases will now be explained:

1. **mod.setParameters():** This function is used to set parameters [23] as presented in listing 3.1. This function will be used to set and initialize the parameters and will be mainly used with the Drag Force Model because this model are going to get input data from Carla in every iteration.

2. **mod.getParameters():** This is a get function, and it is used to return parameters from the OM environment [23]. This function will mainly be used to read the sensor data, to measure the speed of the mass that represents the vehicle in the Drag Force Model.

3. **mod.setSimulationOptions():** This function is used to set the simulation time [23]. It is desired to simulate the Drag Force Model and Carla for the same period of time in each iteration, so this functions will be used for that.

4. **mod.Simulate():** This function is used to simulate OM models from the Python environment[23].

3.2 Carla

In this section the different methods that were acquired to initialize Carla and prepare the tool before it can be connected to the DT will be explained. In 3.2.1 the methods that were used to spawn a vehicle in Carla will be introduced, together with a discussion of the different choices. There will be introduced two different methods of driving the vehicle in Carla, that is dependent on the configurations in Chapter 4, and these two methods will be explained in section 3.2.2. At last, in 3.2.3 the different Carla Python API functions that are going to be implemented to this project will be presented.

3.2.1 Setup and spawn a vehicle with Carla Python API

To link Carla with OM and the restAPI there had to be acquired methodologies of working with the Carla Python API. This subsection will go through all the steps to initialize Carla to connect the tool to the DT. This is the initialization phase before the driving loop starts in chapter 4, so the listings in this subsection are attached, as illustrated in the line numbers of the different listings.

Carla Python API

The first step is to import Carla, but before that can be done the Carla module has to be found, as shown in listing 3.5. The script to import the module was found in Carla's example files, and the material is under the MIT license[22].

```
#This script is from a .py file that is included when the python API is downloaded.
import glob
import os
import sys

try:
    sys.path.append(glob.glob("../Carla/dist/Carla-*%d.%d-%s.egg" % (
        sys.version_info.major,
        sys.version_info.minor,
        "win-amd64" if os.name ==
        "nt" else "linux-x86_64"))[0])
expect IndexError:
    pass

import Carla

```

Listing 3.5: This is a standard method designed by the Carla creators for finding the module to import Carla. The script were available in the example files that got downloaded with the python API. The material in Carla is under the MIT license.

Client

The client is an essential element in the Carla architecture; it connects to the server and makes it possible to apply changes and send information to the Carla environment.[19]. To set the client, the guide in [17] where followed.

```
15      #Client
16      client = Carla.Client("localhost", 2000)
17      client.set_timeout(10.0)
18
```

Listing 3.6: This script is from the *1st. World and client* guide[17].

World Connection

The layout of the world is not important in this stage of the project, because the plan is to drive the vehicle on autopilot so the process of starting to transmit data between Carla and OpenModelica can begin. So, to get started with the fundamental configurations for the DT the world layout will be the default world that is connected to the function *client.get_world()*[17] from line 20 in listing 3.7. The layout for the world that is used in this project are illustrated in figure 3.5. To set the world, the guide in [17] where followed.

```
19      # Find a world
20      world = client.get_world()
21
```

Listing 3.7: This script is from the *1st. World and client* guide[17].

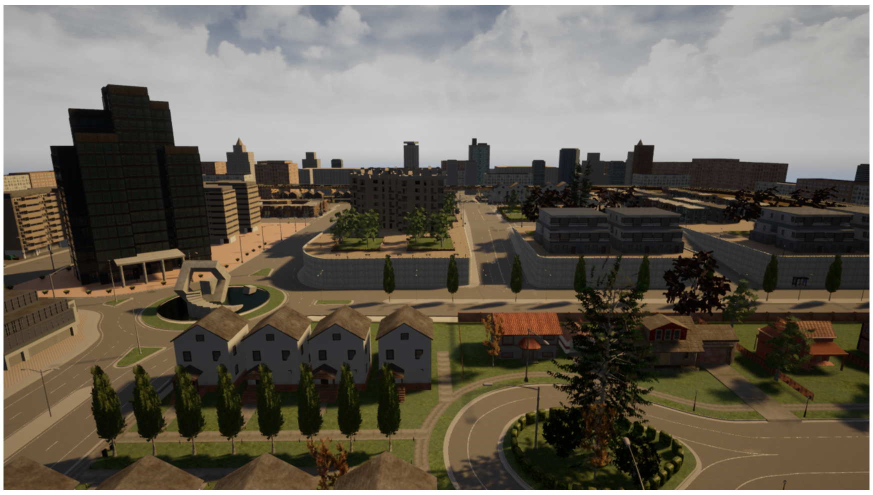


Figure 3.5: This image is taken in CarlaUE4.exe after the world was loaded in Listing 3.7.

Actors and blueprints

When the world is set, the next step is to decide which actors and blueprints libraries that are going to be added to the world object in the Carla environment. In Carla a actor can be a vehicle, pedestrian, sensor, traffic signs and traffic lights[19], Carla got already created blueprints that can be used as actors, and these can be accessed through the Carla blueprint library[18], as illustrated in listing 3.8 line 25. The only actor that is needed to start the prototyping for the DT configuration is a vehicle, because the goal is to be able to work with the data that describes the movements of this car. In Carla's blueprint library there is a blueprint of a Tesla model 3, and that is the vehicle that will be used in this project as illustrated in line 28. The actors should be deleted at the end of the simulation, to clean up the environment before the next simulation as illustrated in line 34. The spawning point for the vehicle will be set to random to begin with as illustrated in line 29-30, so it is possible to see how the vehicle acts in different driving cycles in both the Carla- and the OM environment. To spawn the vehicle, the *2nd. Actors and blueprints* guide [18] were followed.

```
23          import random
24          #Access the blueprint library
25          blueprint_library = world.get_blueprint_library()
26          # Pick car
27          bp = blueprint_library.filter("model 3")[0]
28          # Spawning the car
29          spawn_point = random.choice(world.get_map().
     get_spawn_points())
30          vehicle = world.spawn_actor(bp, spawn_point)
31          # |
32          # | Driving cycle
33          # |
34           vehicle.destroy()
35
```

Listing 3.8: This script is based on the *2nd. Actors and blueprints* guide [18].

3.2.2 Driving algorithm

There will be introduced a simple driving algorithm in this project to configure the connection between OM and Carla, and test the concept of combining these two tools as a DT. When the restAPI gets connected to represent the PT, Carla needs to be controlled by the restAPI. So, the autopilot will then be set to false, and the *vehicle.set_velocity()* function from the *Python API reference*[20] will be implemented. This leads to that the restAPI can control the speed of the vehicle in Carla. This set function will be further explained in 3.2.3.

Autopilot

This method will be used in 4.1 and 4.2 to configure the two OM models with Carla.

```
    vehicle.set_autopilot(True)

```

Listing 3.9: This function were found in the *Python API reference*[20]

3.2.3 Transmit data

There had to be acquired a method of getting the acceleration and the velocity from the Carla environment to simulate the two models in OM. There also had to be acquired a method of setting the velocity in the Carla environment. This is because the Drag Force Model are going to return an action to Carla, and the restAPI are going to control the speed in Carla. So, there were located two GET functions and one SET function from the *Python API reference* [20]. The different functions from the Carla Python API that are going to be used in in this project, and their use cases will now be explained.

get_acceleration() and get_velocity()

These two functions returns the vehicles Carla.Vector3D that represents the acceleration and velocity in x,y and z direction[20]. To be able to input this data to the models in OM, the parameters needs to be represented in one direction. To go from 3D to 1D, it is possible to use Phytagoras, as illustrated in Listing 3.10, but the Carla.Vector3D needs to be converted to a numpy array first, as shown in line 5.

```
1       import numpy as np
2       #Get the acceleration
3       acceleration_3d = vehicle.get_acceleration()
4       #Convert to numpy array
5       a3d_to_a = np.array([acceleration_3d.x,
6                            acceleration_3d.y,
7                            acceleration_3d.z])
8       #Pythagroas to get the accelration in one direction
9       a = round(np.sqrt(np.square(a3d_to_a[0]) +
10                        np.square(a3d_to_a[1]) +
11                        np.square(a3d_to_a[2])),2)
12
```

Listing 3.10: Convert a Carla.Vector3D to a 1D parameter with Phytagoras.

vehicle.set_velocity()

This is a SET function that is used to set the velocity for the vehicle in the Carla environment with a Carla.Vector3D[20]. The function will be used to set the output speed from the Drag Force Model, as the input velocity to the vehicle in the Carla environment. Additionally when the restAPI gets connected as the PT. The speed parameter from OM and the restAPI is represented as a 1D parameter, so it has to be applied a geometric transformation to convert the parameter to a 3D vector. The equation that is presented in eq 3.1 is used for this transformation. The script to do the transformation from 1D to 3D is illustrated in Listing 3.11, and it is based on Alexander Koumis method of going from 3D to 1D in [9]. The rotation matrix in line 5-9 represents the rotation around the yaw axis in the Carla environment, because the vehicle is rotating around this axis when it turns. The one-dimensional speed parameter represents the speed in one direction, which will be assigned to the x value. Accordingly, the y- and z values multiplied with the rotation matrix can be set to zero, as shown in Listing 3.11 line 9. Further, in line 13-15 the new 3D vector is being converted to a Carla.Vector3D, before it can be used with the Carla API function in line 16.

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{3.1}$$

```python
1      import numpy as np
2      #Get the yaw rotation in radians, from the Carla API
3      yaw = np.radians(vehicle.get_transform().rotation.yaw)
4      #Rotation matrix for the yaw axis
5      rotation_m = np.array([
6          [np.cos(yaw), -np.sin(yaw),0],
7          [np.sin(yaw),np.cos(yaw),0],
8          [0,0,1]
9          ])
10     OM_vel_3d = np.array([OM_speed,0,0])
11     #Equation 3.1 with as '@' the matrix multiplication
       operator
12     Carla_input_vel = rotation_m@OM_vel_3d
13     #Transform to Carla.Vector3D
14     Carla_vector_vel = Carla.Vector3D(Carla_input_vel[0],
       Carla_input_vel[1], Carla_input_vel[2])
15     #Set the new velocity with the Carla API
16     vehicle.set_velocity(Carla_vector_vel)
17
```

Listing 3.11: This script transforms a 1D parameter to a Carla.Vector3D which is based on Alexander Koumis method of going from Carla.Vector3D to 1D [9].

3.3 restAPI

The research in this project is mainly focused on the configuration between Carla and OM, but the restAPI is introduced to show that the prototype possibly can be connected to a PT. This section will explain how the DT gets connected to the restAPI, and which states that can be set and returned from the API. To be able to get information from the restAPI, the LGSV environment has to be set first. The available states that can be set through the API are illustrated in table 3.1. Each state is represented by a url, but the url's will not be included in the table. Before the states can be set, the LGSV environment has to be loaded, and an example on how to load the environment is illustrated in listing 3.12.

```python
    #import the requests library
    import requests
    #set the environment
    requests.post("http://119.45.188.204:5000/LGSVL/LoadScene?scene=SanFrancisco&road_num=1")
    #set states
    requests.post("URL:weather information")

```

Listing 3.12: This script connects to the restAPI and sets the environment.

State	Description
Nice weather	No rain
Rain	Ligth, moderate or heavy
Fog	Light, moderate or heavy
Wetness	Light, moderate or heavy
Time of day	Morning, noon or evening

Table 3.1: This table illustrates the states that can be set with the restAPI.

When the environment is loaded, it is possible to get information that describes the vehicle that is situated in the LGSV-simulator with the restAPI. The approach is illustrated in Listing 3.13. In line 2, the LGSVL-simulator is set to run for t=1, so the simulator can generate the state before it is returned. Further, in line 4 the response object that represent the speed is returned. Lastly, in line 6 the JSON content for the response object is returned. The states that can be returned from the LGSVL-simulator are illustrated in table 3.2.

```
    #Let the LGSVL run for t=1 to generate the states
    requests.post("http://119.45.188.204:5000/LGSVL/Run?t=1")
    #Get the speed
    speed_req = requests.post("http://119.45.188.204:5000/LGSVL
   /Status/EGOVehicle/Speed")
    #Get the json content
    speed = speed_req.json()

```

Listing 3.13: This script returns the speed parameter from the LGSVL-simulator.

State	**Description**
x	x coordinate
y	y coordinate
z	z coordinate
rain	Light, moderate or heavy
fog	Light, moderate or heavy
wetness	Light, moderate or heavy
timeofday	Morning, noon or evening
speed	The speed of the vehicle

Table 3.2: This table illustrates the states that can be returned from the restAPI

3.4 The configurations between Carla, OpenModelica and the restAPI

This section introduced the configuration that got designed after the methodology for initializing Carla, OpenModelica, and the restAPI were acquired. Figure 3.6, illustrates the diagram of the DT system that will be designed in this project. This configuration got the characteristics of a DS because there are no returned actions from the DT to the PT[12]. However, the configuration represents a prototype, and this configuration will introduce the opportunity to include all the methodologies acquired. This configuration will lead to gained experience in how OpenModelica and Carla can transmit data bidirectionally, additionally if the configuration can serve as a DS. It can be applied further research based on the methodologies used in this project to extend the features of the DS to develop it to a DT. Furthermore, since this configuration is currently a prototype, it will still be referred to as a Digital Twin throughout the project.

By following figure 3.6, it can be seen that Carla will simulate one step with the input from the restAPI. The measured output speed from Carla, will be used as the input speed to simulate the Drag Force Model. The Drag Force Model will deploy the drag force equation through the simulation, and return the updated speed back to the Carla environment. This leads to that it will be generated two steps in Carla, so the concept is to illustrate that the DT can be one step ahead of the PT. The restAPI will then deploy a new step, and the loop will repeat. The Simple Car Model is used to re-simulate the scenario to show the concept of simulating historical data in OM with the DT.

This is the configuration that will be designed in chapter 4, and there will be done experiments related to this configuration in chapter 5. The design of this configuration is the process of responding to RQ3.

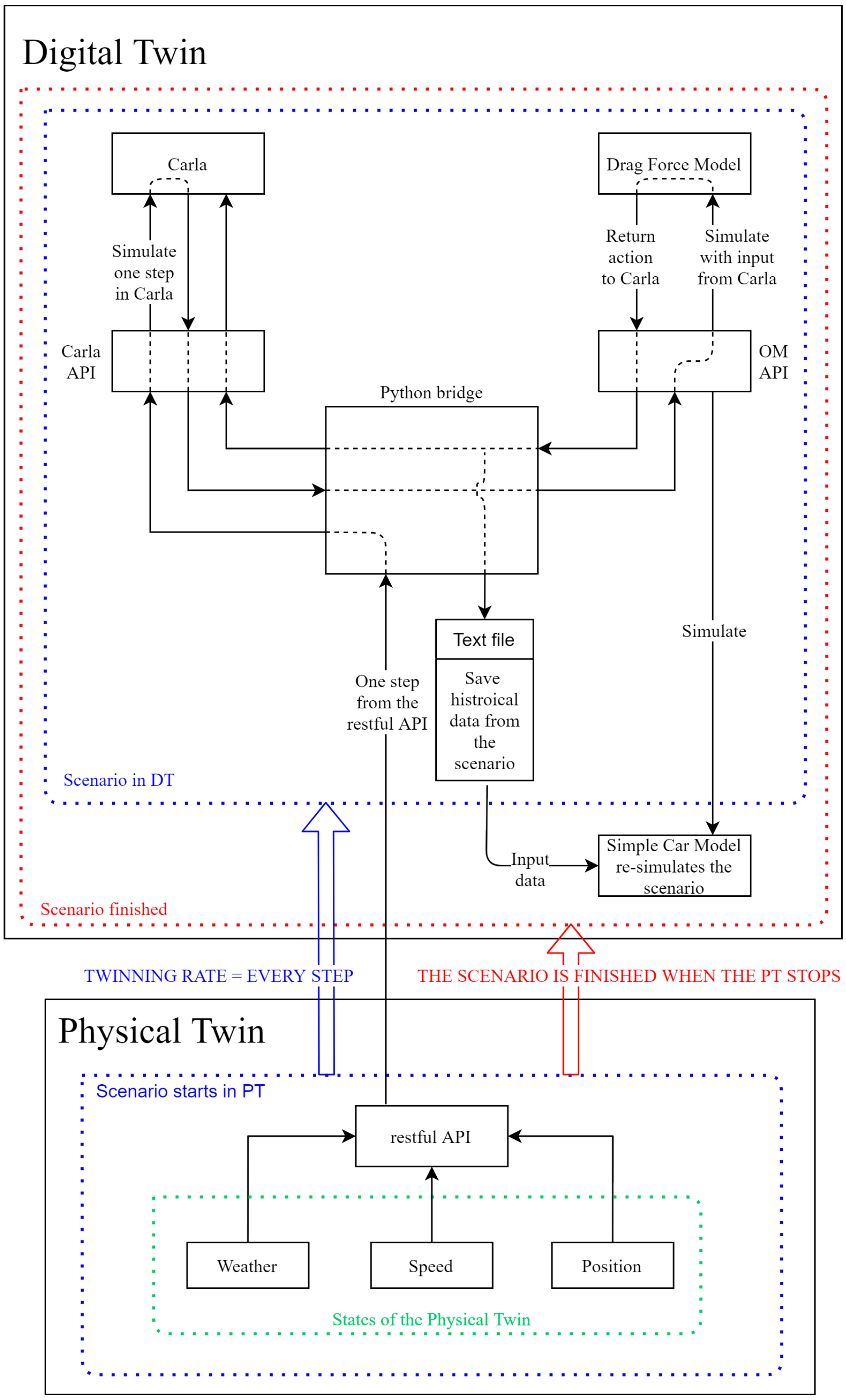

Digital Twin
Carla
Drag Force Model
Simulate one step in Carla
Return action to Carla
Simulate with input from Carla
Carla API
OM API
Python bridge
Text file
Save histroical data from the scenario
One step from the restful API
Simulate
Scenario in DT
Input data
Simple Car Model re-simulates the scenario
Scenario finished
TWINNING RATE = EVERY STEP
THE SCENARIO IS FINISHED WHEN THE PT STOPS
Physical Twin
Scenario starts in PT
restful API
Weather
Speed
Position
States of the Physical Twin

Figure 3.6: This figure shows the protoype of the DT of the vehicle by combining Carla, OM and the restAPI

Chapter 4

Design and Implementation

This chapter will combine all the acquired methodologies from chapter 3 to design the prototype of the DT. The configuration that is illustrated in figure 3.6 will be split into smaller connections to start the design. The chapter will begin the connection between OpenModelica and Carla where the historical data is saved from Carla and then used as input data to re-simulate the scenario in the Simple Car Model. Further, the design of the connection where the input to the Drag Force Model is updated for every step in Carla will be presented, so the tools can be simulated parallelly. Then the connection where the Drag Force Model will return an action back to the Carla environment will be designed, to show that the tools can transmit data bidirectionally. At last, the scripts that were designed in 4.1, 4.2 and 4.3 will be combined to design the configuration that that is illustrated in figure 3.6, to connect it to the restAPI.

4.1 Simple Car Model connected to Carla

The connection that will be designed in this section can be seen in figure 4.1. The figure illustrates that the speed parameter gets saved to a text file for each step in Carla. When the driving cycle in Carla is complete, the Simple Car Model will re-simulate the driving cycle. This connection aims to see if it is possible to generate the same driving cycle as in Carla with the Simple Car Model with the acquired methodologies.

The first step in this connection is to create a text file, so the speed parameter measured from Carla can be written to this text file throughout the simulation to save the historical data. For OM to understand the table in the text file, the table has to be on the form as illustrated in Figure 4.1[21]. The name of the table is *table1* and this name has to be defined to the *CombiTimeTable* block that feeds the input data to the *PropDriver* block designed by Ceraolo in [5].

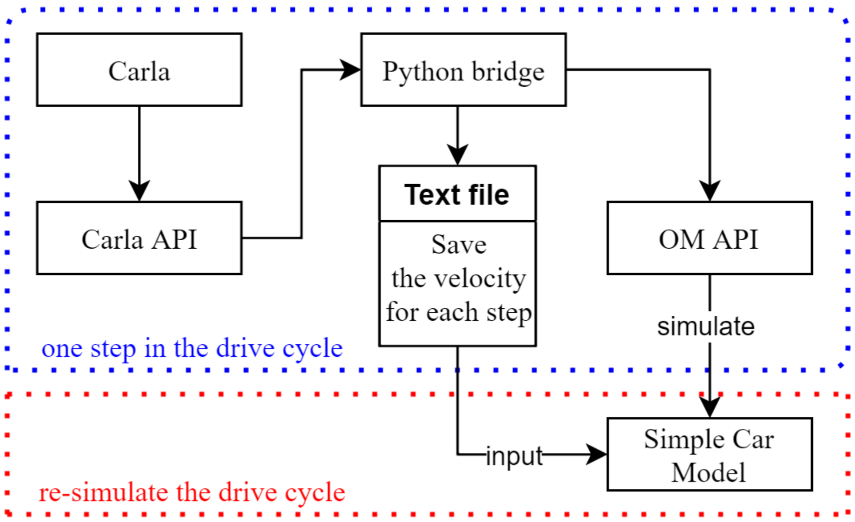
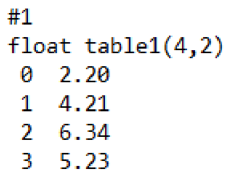

Carla
Carla API
Python bridge
Text file
Save the velocity for each step
OM API
simulate
one step in the drive cycle
input
Simple Car Model
re-simulate the drive cycle

Figure 4.1: This figure illustrates the connection between Carla and the Simple Car Model that are going to be designed in this section.



```
#1
float table1(4,2)
 0  2.20
 1  4.21
 2  6.34
 3  5.23
```

Figure 4.2: This figure illustrates the form that the table in the text file has to be on for the OM model to understand the data.

The path to the text file has to be in the resource files of the OM library, so the table can be portable as explained in 3.1.3. In Listing 4.1, the script that opens the text file to save the historical data is illustrated. Further, in line 19 in Listing 4.2 for the driving loop, the text file is written on the form shown in Figure 4.2.

```
    #Set path
    table_path = \
    'C://Program Files//OM1.16.1-64bit//lib//' \
    'omlibrary//DigitalTwinLibrary//Resources//Data//' \
    'Tables//data_from_Carla.txt'
    #Open file
    f = open(table_path, 'w+')
    #Write the name and size of the table
    f.write('#1'+'\n'+'float table1('+str(iterations)+',2)'+'\n
')
```

```
10
```

Listing 4.1: Open the text file that holds the table for the driving cycle.

The next step is to start the driving loop in Carla, as illustrated in Listing 4.2. Line 13 tells the Carla simulator to perform a tick, in line 15 the loop is paused until the tick is received from the Carla simulator. In comparison, if the loop is not paused the loop may continue while the tick is still loading, and that can cause wrong readings throughout the loop. In line 18, the velocity vector of the vehicle gets returned, and in line 18-25 the velocity vector is being converted to a one-dimensional speed parameter by deploying Pythagoras. The one-dimensional speed parameter will be written to the text file for each iteration, as illustrated in line 28. To stop the Simple Car Model from looping through the driving cycle several times, the stop time of the simulation in OM is set to the same number as the iterations in line 33. Finally, the Simple Car Model get simulated in line 35.

```
8       #Import numpy library
9       import numpy as np
10      #Starts the driving loop
11      for _ in range(iterations):
12          #Carla API: One tick in carla
13          world.tick()
14          #WCarla API: Wait for the tick
15          world.wait_for_tick()
16
17          #Carla API: Get velocity from Carla
18          velocity_3d = vehicle.get_velocity()
19          #Convert to numpy array
20          velocity = np.array([velocity_3d.x,
21                               velocity_3d.y,
22                               velocity_3d.z])
23          #Pytagoras from Listing 3.10
24          speed = np.sqrt(np.square(velocity[0]) +
25                          np.square(velocity[1]) +
26                          np.square(velocity[2]))
27          #Write to file
28          f.write('  '+str(_)+'   '+(str(round(speed,2)))+'\n')
29
30      #Close the text file
31      f.close()
32      #OM API: Set the stop time in OM to the same number of
       iterations
33      mod.setSimulationOptions("stopTime="+str(iterations))
34      #OM API: Simulate the Simple Car Model
35      mod.simulate()
36
```

Listing 4.2: Simulate Carla and the Simple Car Model.

When the OM simulation is complete, the measured driving cycle for the Simple Car Model is returned as a numpy array from the velocity sensor, *velSens* block as illustrated in Listing 4.3 line 2. Where '0' is the position in the numpy array that holds the measured driving cycle. The length of the returned array is longer then the number of iterations in the for loop, because OM returns results where the number of iterations are split into smaller intervals. To plot the measured driving cycle from OM with the same Y-axis as the driving cycle in Carla, the measured velocity array got shrunken as illustrated in line 4-8.

```
1     #OM API: Return the measured driving cycle for the Simple
     Car Model
2     measured_vel = mod.getSolutions("velSens.v")[0]
3     #Shrink the array:
4     #Divide the length by the iterations to find the step
5     div = round((len(measured_vel))/iterations)
6     #Shrink the array with np slicing [start:stop:step]
7     shrink_array = measured_vel[0:(len(sensor_array)):div]
8     speed_om = shrink_array[0:iterations]
9
```

Listing 4.3: Shrink the returned array from OM

4.2 Drag Force Model connected to Carla

In this section, the Drag Force Model will get connected to Carla. The connection is illustrated in figure 4.3, which shows that for every step in Carla, the velocity will be used as input to simulate the Drag Force Model.

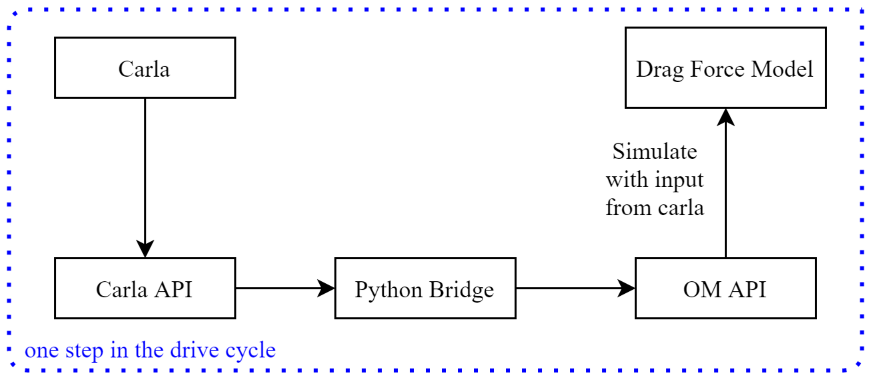


Figure 4.3: This figure illustrates the connection between Carla and the Drag Force Model that are going to be designed in this section.

The goal for this connection is to simulate the Drag Force Model with every tick in Carla, to see if the two tools can run parallelly. Listing 4.4 illustrates the approach of how this was done. In line 3 *time* gets imported, so it is possible to use the *time.time()* function in the driving cycle to be able to measure the time of the tick in the Carla environment. The simulation time of the Drag Force Model in OM, should be simulated for the same time as the tick in Carla. So, the *time.time()* function is stopped in line 13, and the difference between the start and the stop time is used as the simulation time in OM in line 17. The next step is to get the parameters that are going to be used as inputs to the Drag Force Model, because the *mass* block in the Drag Force Model needs the acceleration and the speed from Carla to represent the same forward force. In line 20-37, the Carla.3Dvectors which represents the velocity and the acceleration gets returned, and Pythagoras gets deployed to transform the vectors to 1D parameters for the simulation of the Drag Force Model. In line 40 and 41 the one-dimensional acceleration and speed parameters are set to the *constant* blocks connected to the *mass* block as explained in 3.1.3 (Method 2). Additionally, the Drag Force Model gets simulated in line 43 and the measured velocity is getting returned from the *velSens* block in line 45. This array holds the measures for the whole driving cycle that is generated by the Drag Force Model. Accordingly, to get the last measured output speed, the last position in the array has to be accessed as illustrated in line 47.

```
1    #Import numpy- and time library
2    import numpy as np
3    import time
4    #The driving loop starts
5    for _ in range(iterations):
6        #Start the counter
7        start = time.time()
8        #Carla API: One tick in Carla
9        world.tick()
10       #Carla API: Wait for the tick
11       world.wait_for_tick()
12       #Stop the counter
13       stop = time.time()
14       #Calculate the time to load the tick in Carla
15       duration = stop - start
16       #OM API: Set the same simulation time for the Drag
     Force Model
17       mod.setSimulationOptions("stopTime="+str(duration))
18
19       #Carla API: Get acceleration from Carla
20       acceleration_3d = vehicle.get_acceleration()
21       #Convert to numpy array
22       a3d_to_a = np.array([acceleration_3d.x,
23                            acceleration_3d.y,
24                            acceleration_3d.z])
25       #Phytagoras from Listing 3.10
26       a = round(np.sqrt(np.square(a3d_to_a[0]) +
27                         np.square(a3d_to_a[1]) +
28                         np.square(a3d_to_a[2])),2)
29       #Carla API: Get velocity from Carla
30       velocity_3d = vehicle.get_velocity()
```

```
        #Same approach as line 21-28 to convert to 1D
        velocity = np.array([velocity_3d.x,
                             velocity_3d.y,
                             velocity_3d.z])
        v = round(np.sqrt(np.square(velocity[0]) +
                          np.square(velocity[1]) +
                          np.square(velocity[2])),2)

        #OM API: Set the 1D speed and accelration parameters
        mod.setParameters("start_vel.k={}".format(v))
        mod.setParameters("start_acc.k={}".format(a))
        #OM API: Simualte the Drag Force Model
        mod.simulate()
        #OM API: Return measured speed
        vel_sensor = mod.getSolutions("velSens.v")[0]
        #Get the last item in the array
        OM_vel = vel_sensor[len(vel_sensor)-1]

```

Listing 4.4: Simulate Carla and the Drag Force Model

When the connection for the Drag Force Model and Carla is established, the next step is to access the *dragF* block, which is designed by Ceraolo in [5]. In addition to changing the different parameters that affect the vehicle's moving resistance. The script that accesses the different parameters are illustrated in Listing 4.5. There will be experimented with different values of the rolling friction parameter for the Drag Force Model with this connection in section 5.2.2, to see if it causes any difference to the measured output speed. The values of the rolling friction that will be used in this project is based on Ejsmont et al. findings in [4] as addressed in 2.4.

```
    #Variables for the drag force equation
    m = 1600              #Mass
    rho = 0.001225        #Air density
    fc = 0.01             #Rolling friction
    Cx = 0.23             #Drag coefficient
    S = 2.22              #Vehicle cross area

    #OM API: Initialize parameters
    mod.setParameters("mass.m={}".format(m))
    mod.setParameters("dragF.S={}".format(S))
    mod.setParameters("dragF.fc={}".format(fc))
    mod.setParameters("dragF.Cx={}".format(Cx))
    mod.setParameters("dragForce.m={}".format(m))
    mod.setParameters("dragF.rho={}".format(rho))
    #OM Carla: Initialize parameters
    physics_control.mass = m
    physics_control.drag_coefficient = Cx
    physics_control = vehicle.get_physics_control()

```

Listing 4.5: Initialize the drag force parameters

4.3 the Drag Force Model returns an action to Carla

The goal of this connection is to return an action back to the Carla environment from the Drag Force Model, to see if it possible to exchange bidirectionally data between the tools. The connection is illustrated in figure 4.4, which is an extended version of the connection from section 4.2.

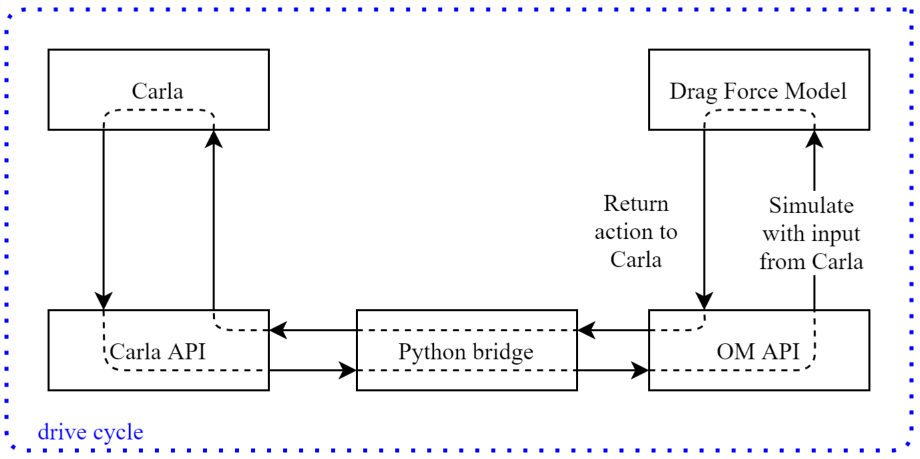


Figure 4.4: This figure illustrates the connection where the Drag Force Model are going to return an action back to Carla, that are going to be designed in this section.

In section 4.1 and 4.2, the two different OM models got connected to determine if it was possible to generate the same driving cycle by simulating the historical data with the Simple Car Model and simulating Carla and OM parallelly with the Drag Force Model. The next step is to configure OM to return an action back to the Carla environment. The framework for this configuration is based on that the Drag Force Model will update the velocity in the Carla environment, after subtracting the *dragF* block which is designed by Ceraolo in [5].

Since this script is an extended version of the connection in 4.2, the script will be added to Listing 4.4. In listing 4.4 line 47, the measured speed from the Simple Car Model is returned to the variable *OM_vel*. This variable represents the speed in one dimension, and as explained in section 3.2.3, the *set_velocity()* function requires a *Carla.3Dvector* as input. Accordingly, before the speed can be returned to Carla, it has to be done a geometric transformation from 1D to 3D. The method for applying this transformation together with how to use the *set_velocity()* function is illustrated in the script in Listing 3.11 in 3.2.3, which will be the extension of Listing 4.4 to design the connection in Figure 4.4.

4.4 The DT connected to the restAPI

In this section, the designing phase of the DT system is presented. The connections from 4.1 and 4.3 will be combined to serve as the DT for the restAPI. The configuration which will be designed is illustrated in Figure 3.6. The connection from 4.1 will introduce the feature of re-simulating scenarios with OM, in addition the connection from 4.3 will give the opportunity to bidirectionally exchange data between Carla and OM in the DT.

The Drag Force Model implements the resistance to movement of the vehicle in the DE, and as addressed in 2.4 the rolling friction is affected by the wetness of the road[4]. To introduce the scenario of adding wetness to the road, the weather in Carla and the restAPI will be set to ligth-, moderate- or heavy rain. In addition, the rolling friction parameter for the *dragF* block, which is designed by Ceraolo in [5], will be changed based on the weather that is set for the environment in Carla and the restAPI. Listing 4.6 illustrates the initializing phase for the weather for the restAPI, Carla and the rolling friction parameter. In line 1 *requests* gets imported, so the script can access the restAPI. Further, in line 2, the scene is loaded for the LGSV-simulator to be able to get and set states to the LGSVL environment. In line 4-8 the weather is set manually with True or False , and the weather settings for the tools changes based on these choices as illustrated in line 10-22. Line 16-22 illustrates how the different weather parameters is set if *light* from line 6 is set to True. The method of setting the rain to *moderate* and *heavy* is the same as in line 16-22, so they are not included in the script in listing 4.7.

```
1   import requests
2   requests.post("http://119.45.188.204:5000/LGSVL/LoadScene?
   scene=SanFrancisco&road_num=1")
3
4   #Set weather manually
5   no_rain = False      #fc = 0
6   light = True         #fc = 0.01
7   moderate = False     #fc = 0.02
8   heavy = False        #fc = 0.03
9
10  if no_rain == True:
11      requests.post("http://119.45.188.204:5000/LGSVL/Control
   /Weather/Nice")
12      weather = Carla.WeatherParameters.Default #:Carla API
13      world.set_weather(weather)                #:Carla API
14      fc = 0.00  #Rolling friction
15
16  if light == True:
17      requests.post("http://119.45.188.204:5000/LGSVL/Control
   /Weather/Rain?rain_level=Light")
18      requests.post("http://119.45.188.204:5000/LGSVL/Control
   /Weather/Fog?fog_level=Light")
```

```python
        requests.post("http://119.45.188.204:5000/LGSVL/Control/Weather/Wetness?wetness_level=Light")
        weather = Carla.WeatherParameters.SoftRainSunset #: Carla API
        world.set_weather(weather)                       #: Carla API
        fc = 0.01  # Rolling friction

```

Listing 4.6: Initializing the weather settings

The next step in this configuration is to turn off the autopilot for Carla and use the returned speed parameter from the restAPI to control the velocity in Carla. In line 1 in Listing 4.8, the LGSVL-simulator is set to run for t=1, so the states can be loaded in the LGSVL environment. Further, in line 2 the speed parameter is returned by the restAPI and in line 3 the JSON content is returned from the request object. The speed returned from the restAPI is represented as a one dimensional speed parameter, so the geometric transformation from Listing 3.11 in 3.2.3 is applied to convert the speed parameter to a Carla.Vector3D. Furthermore, to set the velocity vector to Carla the *set_velocity()* function is used as illustrated in line 7 in Listing 4.7. This script will represent the connection between the PT and the DT, so Listing 4.7 has to be implemented inside the for-loop that represents the driving cycle in Listing 4.4. The PT will feed the DT with the forward speed, so it is the first lines of code that should be iterated in the driving cycle. For this reason, Listing 4.7 should be implemented before the time.time() function in line 7 in Listing 4.4.

```python
1     requests.post("http://119.45.188.204:5000/LGSVL/Run?t=1")
2     speed_req = requests.post("http://119.45.188.204:5000/LGSVL
      /Status/EGOVehicle/Speed")
3     speed_restAPI = speed_reg.json()
4
5     #Apply the 1D to 3D geometric transformation from section
      3.2.3
6
7     vehicle.set_velocity(speed_restAPI)
8
```

Listing 4.7: Set the input speed to the DT with the state returned by the restAPI

The last step that has to be implemented in this configuration is to add the features where the Simple Car Model is re-simulating the scenario, the method to do this is to add Listing 4.1 to open the text file, and line 22 from listing 4.2 where the text file is being written to for every iterations inside the for loop, and when the for loop ends the text file should be closed and the Simple Car Model should be simulated as shown in line 24-27 in listing 4.2.

Chapter 5

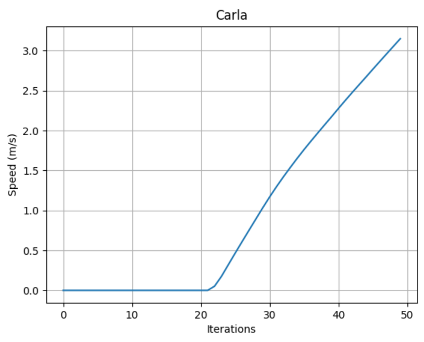
Experiments and Results

In this chapter the experiments and the results for the different connections that where designed in chapter 4 will be introduced. In section 5.1, the experiments from the design in 4.1, where the Simple Car Model is connected to Carla will be presented. Further, in section 4.2, the experiments from the design in 4.2, where the Drag Force Model is connected to Carla will be presented. In section 5.3, the connection from 4.3 where the Drag Force Model returns an action to Carla will be presented. At last, in section 5.4, the experiments and results from when the DT is connected to the restAPI in section 4.4, will be presented.

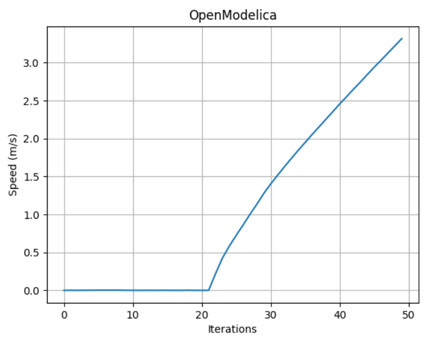
5.1 Simulate the Simple Car Model with input from Carla

In this section there will be presented one image that that represents the driving cycle in Carla, and one image that represents the driving cycle for the Simple Car Model. The goal for the experiment is that the Simple Car Model should copy Carla's driving cycle, to see if the Simple Car Model and the method of working with historical data can be used to re-simulate scenarios. There will be presented results with 50, 500 and 1000 iterations for this experiment.

Figure 5.1 illustrates the results of 50 iterations. The results shows that the Simple Car Model and Carla represents the same driving cycle. They both start to throttle at 22 iterations, and the top speed after 50 iterations are 3.5 m/s.



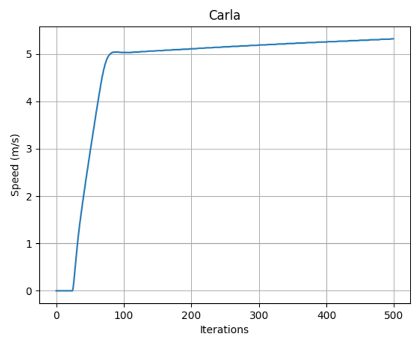
(a) Carla



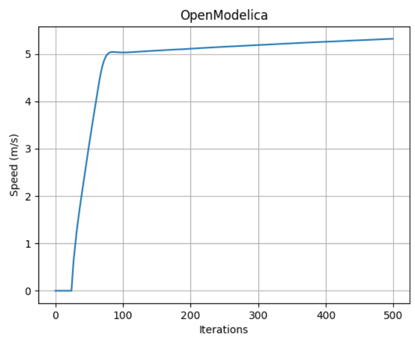
(b) Simple Car Model

Figure 5.1: 50 iterations with the Simple Car Model

Figure 5.2 illustrates the results of 500 iterations. The results shows that the Simple Car Model and Carla represents the same driving cycle. They both start to throttle after 20 iterations, and from 100 iterations and out there is an steady accelerations up to a top speed of 5.5 m/s.



(a) Carla

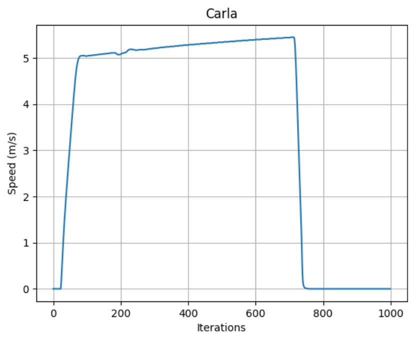


(b) Simple Car Model

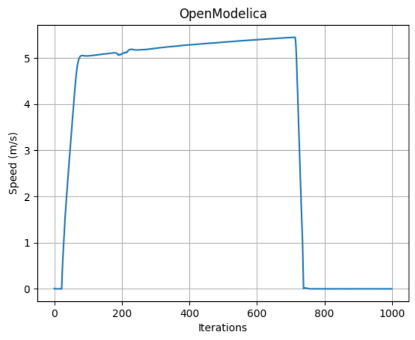
Figure 5.2: 500 iterations with the Simple Car Model

Figure 5.3 illustrates the results of 1000 iterations. The results shows that the Simple Car Model and Carla represents the same driving cycle. In this figure, it can be seen that the small spikes after 200 iterations in Carla's driving cycle are reflected in the Simple Car Model.

The results from the three experiments over 50, 500 and 1000 iterations presents that the method of working with historical data in OpenModelica can be a good feature to add to the DT.



(a) Carla



(b) Simple Car Model

Figure 5.3: 1000 iterations with the Simple Car Model

5.2 Simulate the Drag Force Model with input from Carla

The main goal of this experiment is to see if it is possible to simulate the Drag Force Model and the vehicle in Carla parallelly by sending input data to the Drag Force Model for every tick in Carla. In section 5.2.1 is it presented an experiment where the rolling friction is set to 0 to make sure that Carla and the Drag Force Model got the same driving cycle while simulated parallelly. Further, in section 5.2.2, there will be presented experiments to determine if the rolling friction parameter will affect the driving cycle of the mass that represents the vehicle in the Drag Force Model to decide if this is a feature that can be added to Carla. Three experiments will be presented where the rolling friction is set to 0.01, 0.02, and 0.03 which are based on Ejsmont et al. findings in [4]. For the experiments in this section, the driving cycle for both Carla and the Drag Force Model will be plotted in the same figure. In addition to analyzing any changes that may occur in the driving cycle.

5.2.1 Drag Force Model with no rolling friction

When the rolling friction is set to 0, the expected results are that the Drag Force Model and Carla will represent the same driving cycle. Whereas if that is the result, it will be possible to explore if the rolling friction parameter may cause differences to the driving cycle. There will be presented results over 50, 500, and 1000 iterations in this experiment, and the results will now be presented.

1. **50 iterations:** The result for this experiment can be seen in figure 5.4, and the red line that represents the OM model is on-line with the blue line that represent Carla's driving cycle. This is a good result because it shows that the method of simulating Carla and the Drag Force Model parallelly generates the same output.

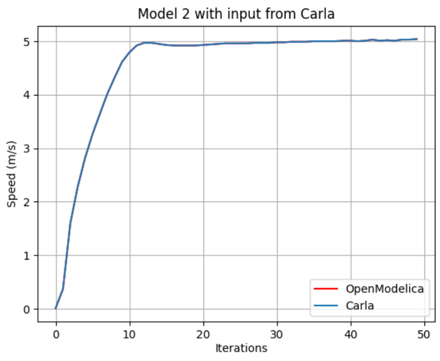
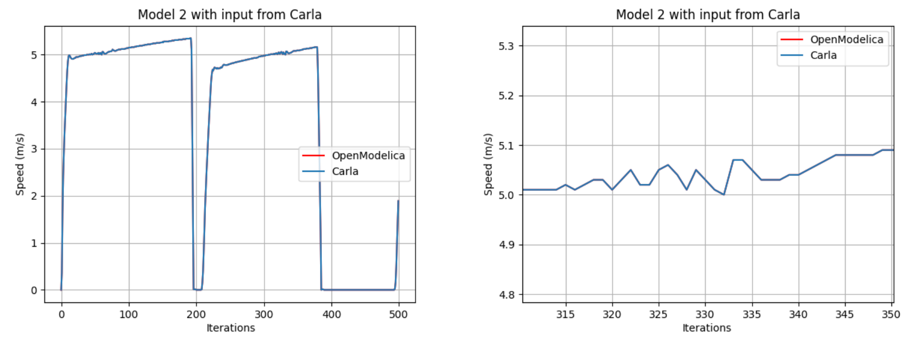


Figure 5.4: 50 iterations with the Drag Force Model

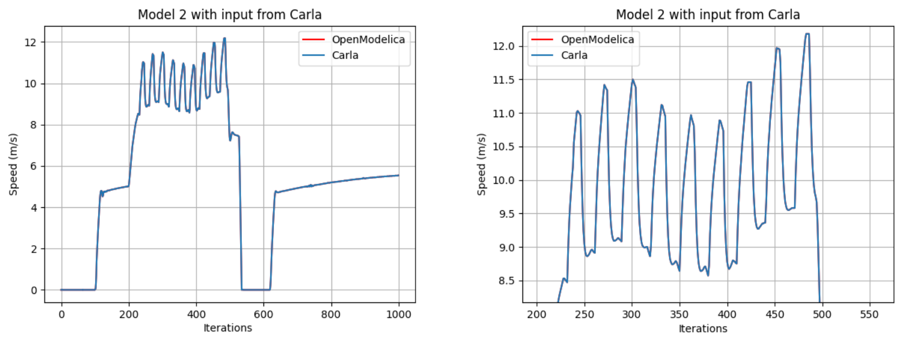
2. **500 iterations:** The results for 500 iterations can be seen in figure 5.5, and the results shows that Carla and the Drag Force Model represents the same Drive Cycle. Figure 5.5(b) illustrates a zoomed in representation of an area where the acceleration is changing fast, and the results are as expected.



(a) Full cycle (b) Zoomed in

Figure 5.5: 500 iterations with the Drag Force Model

3. **1000 iterations:** The results for 1000 iterations are also accepted because the driving pattern is perfectly copied as shown in figure 5.6. There is also a zoomed in presentation of the driving cycle from 200-500 iterations in figure 5.6 (b) to make sure that the the Drag Force Model holds the cycle even though there are rapid changes in speed over short period of time.



(a) Full cycle (b) Zoomed in

Figure 5.6: 1000 iterations with the Drag Force Model

5.2.2 Changing the rolling friction in the Drag Force Model

The next step is to experiment with the rolling friction to see if it causes any difference to the driving cycle in OM. The results for these experiments will be introduced in this subsection, together with graphs and calculations that shows how much the speed is changed dependent on the rolling friction. There will only be introduced experiments with 1000 iterations in this thesis to show the difference.

To make sure that the difference in speed between the two tools is 0 when the rolling friction is set to 0, the sum of the forward speed for the whole driving cycle got summed for both Carla and the Drag Force Model. The difference got calculated by subtracting the sum of Carla by the sum of the Drag Force Model, as illustrated in Listing 5.1. The difference were calculated over 5 iterations as illustrated in table 5.1, and the average difference in speed is 0.00154 m/s. This is a good base for the further experiments, to be able to see if the rolling friction causes any difference in the speed.

```python
    import numpy as np
    sum_om = np.sum(om_plot)
    sum_Carla = np.sum(Carla_plot)
    diff_speed = sum_Carla - sum_om
```

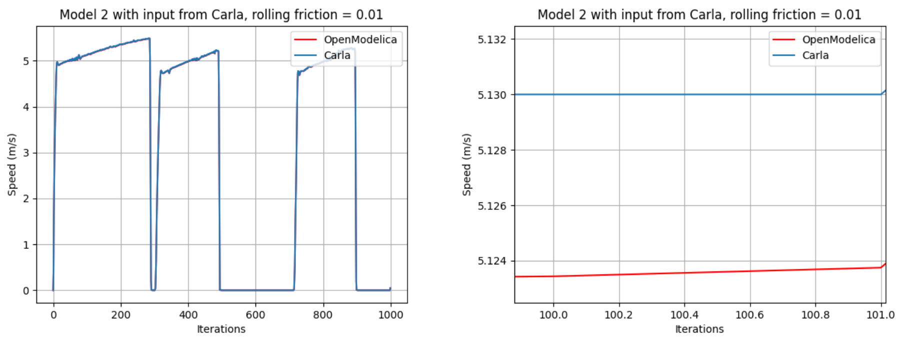
Listing 5.1: Python example

Rolling friction = 0	
Simulation	**Difference** (m/s)
1	0.0015
2	0.0018
3	0.0013
4	0.0015
5	0.0016
Average	0.00154

Table 5.1: Speed difference for Carla and the Drag Force Model

Rolling friction = 0.01

In this experiment the rolling friction parameter will be set to 0.01. The change in the speed is very small, so it is difficult to see it in figure 5.8 (a). It will be easier to see the change when the rolling friction increases, but by taking a close look it is possible to see a slight red line that represent the simulation in the Drag Force Model, under the steady accelerations from 10-280, 300-480 and 720-900 iterations in figure 5.8 (a). Figure 5.9 (b) is zoomed in at the area around 100 iterations, so the reader can clearly see the difference.



(a) The whole driving cycle (b) Zoomed in at 100 iterations

Figure 5.7: 1000 iterations with the Drag Force Model while rf = 0.01

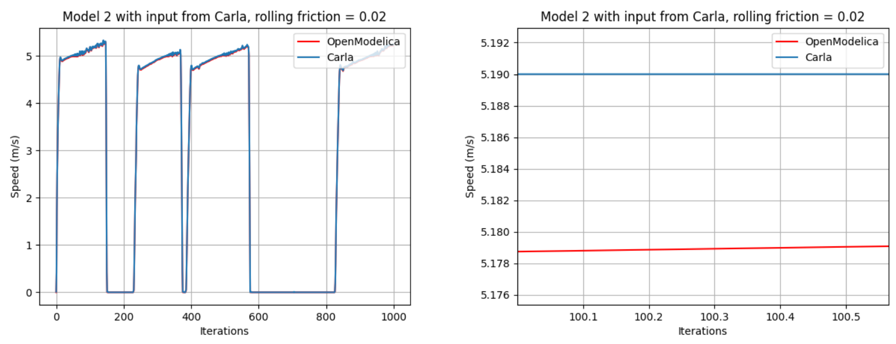
In table 5.2 the difference between the speed of Carla and the Drag Force Model is calculated, and as illustrated the average difference in speed is 2.63 m/s over 1000 iterations, that equals a difference of 9.468 km/h. So with the added rolling friction of 0.01, it is possible to see that the the Drag Force Model generates a slower forward speed then the vehicle in Carla. The results from these experiments are as expected, because by changing the rolling friction, there will be more force that is working against the mass that represents the vehicle in the Drag Force Model. The change in speed is very small, but it is still an interesting feature that can be added to the DE, specially if the prototype are going to be further developed and possibly be used in collision scenarios.

Rolling friction = 0.01	
Simulation	**Difference** (m/s)
1	2.61
2	2.58
3	2.68
4	2.79
5	2.51
Average	2.63

Table 5.2: Speed difference when the rolling friction is set to 0.01.

Rolling friction = 0.02

In this experiment the rolling friction parameter in the Drag Force Model is set to be 0.02. It is possible to see a slightly bigger distance between the blue and the red line in figure 5.10 (a), then from the experiment in figure 5.8. This is as expected because when the rolling friction parameter increases, the forces that works against the vehicle in the Drag Force Model will also increase. In figure 5.10 (b), there is possible to see a zoomed in presentation of the graph at around 100 iterations.



(a) The whole driving cycle (b) Zoomed in at 100 iterations

Figure 5.8: 1000 iterations with the Drag Force Model while rf = 0.02

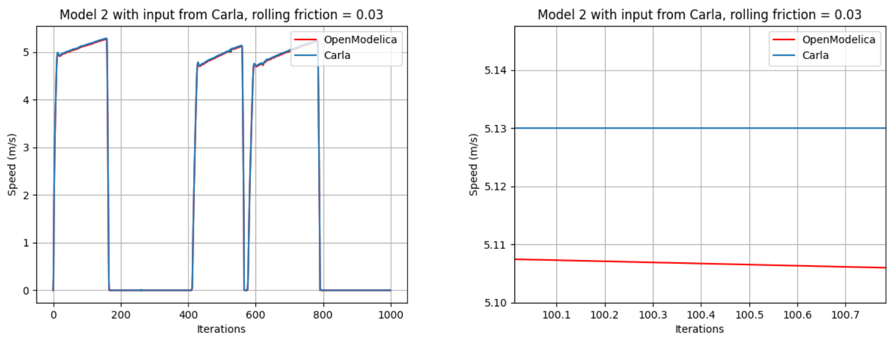
The average difference between the speed is illustrated in table 5.3, and this table shows that the average speed difference over all the iterations in this experiment is 4.30 m/s. That is an increase from the experiment where the rolling fiction where set to 0.01, so this is a good result. The difference of 4.30 m/s equals 15.48 km/h.

Rolling friction = 0.02	
Simulation	**Difference** (m/s)
1	4.31
2	4.25
3	4.38
4	4.42
5	4.18
Average	4.30

Table 5.3: Speed difference while the rolling friction is set to 0.02.

Rolling friction = 0.03

The rolling friction will in this experiment be set to be 0.03. In figure 5.12 (a) the red line that represents the Drag Force Model is more clear then in the other experiments, and that is as expected when the rolling friction increases. In figure 5.12 (b),the reader can see the difference more clearly, there is also possible to compare the Y-axis of this figure with the other (b) figures in this section to see the difference.



(a) The whole driving cycle (b) Zoomed in at 100 iterations

Figure 5.9: My flowers.

The difference between the speed is illustrated in table 5.4, and it is possible to see that the average difference in speed is almost doubled compared to the experiment where the rolling friction is set to be 0.02. This is a good finding, because it shows that the DragF block that Ceraolo designed in [5] can be integrated to the DT, to represent the drag force equation.

Rolling friction = 0.03	
Simulation	**Difference** (m/s)
1	8.74
2	8.61
3	8.59
4	8.66
5	8.72
Average	8.66

Table 5.4: Speed difference while the rolling friction is set to 0.03.

5.3 Return an action to Carla with the Drag Force Model

From the experiments in 5.2.2, it is possible to see that the speed of the vehicle in the Drag Force Model decreases when the rolling friction increases. The goal of this experiment is to return the speed that is generated in the Drag Force Model back to the vehicle in Carla, to see if it is possible to reduce the speed difference between the tools when the rolling friction increases. If the difference between the tools decreases, it shows that OpenModelica can return an action to Carla.

The results for the experiments in this section are partly accepted, because the speed difference decreases so it shows that the Drag Force Model can return an action, but for the results to be accepted the difference in speed has to be 0. The reason for why this experiment did not work out as planned is because Carla got autopilot turned on, so the vehicle in Carla will automatically throttle or brake even though the Drag Force Model sets a new input speed.

Rolling friction = 0.01

When the Drag Force Model returns an action to Carla, the speed difference between the two tools is reduced from 2.6 ms to 1.93 when the rolling friction is 0.01.

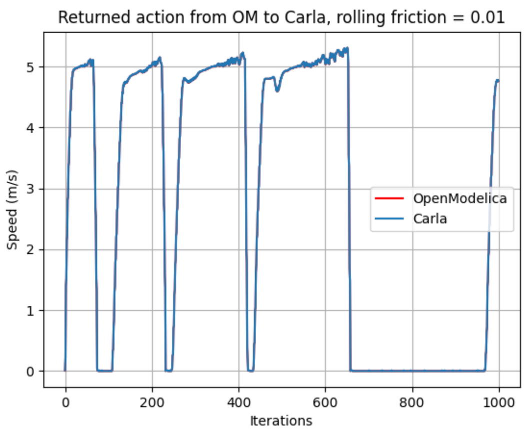


Figure 5.10: 1000 iterations when the Drag Force model returns an action

Rolling friction = 0.01	
Simulation	**Difference** (m/s)
1	1.93
2	1.88
3	1.95
4	1.87
5	1.91
Average	1.90

Table 5.5: Drag Force Model returns an action, rolling friction is set to 0.01

Rolling friction = 0.02

When the Drag Force Model returns an action to Carla, the speed difference between the two tools is reduced to 3.99 m/s from 4.3 m/s when the rolling friction is 0.02.

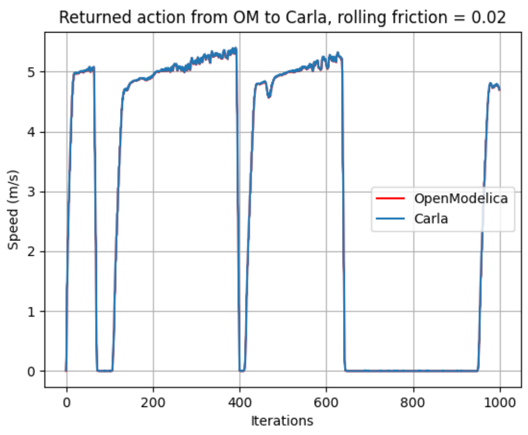


Figure 5.11: 1000 iterations when the Drag Force model returns an action

Rolling friction = 0.02	
Simulation	**Difference** (m/s)
1	3.99
2	3.87
3	3.88
4	3.92
5	3.94
Average	3.92

Table 5.6: Drag Force Model returns an action, rolling friction is set to 0.02

Rolling friction = 0.03

When the Drag Force Model returns an action to Carla, the speed difference between the two tools is reduced to 5.55 m/s from 8.6 m/s when the rolling friction is 0.03.

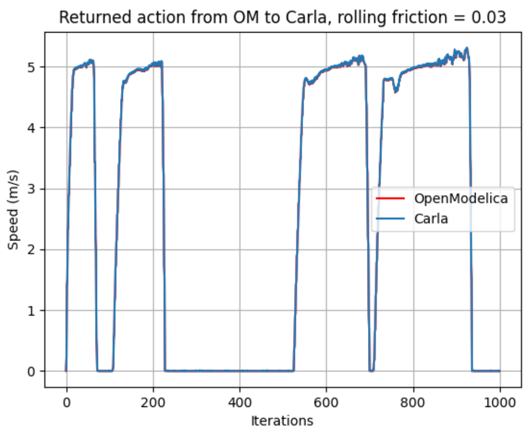


Figure 5.12: 1000 iterations when the Drag Force model returns an action

Rolling friction = 0.03	
Simulation	**Difference** (m/s)
1	5.55
2	5.63
3	5.62
4	5.51
5	5.46
Average	5.554

Table 5.7: Drag Force Model returns an action, rolling friction is set to 0.03

5.3.1 The DT connected to the restAPI

The experiments that were done while the DT is connected to the restAPI are going to be presented in this section. The only input parameter from the restAPI is the forward speed. The vehicle in Carla will then crash after a couple of seconds, because there are no steering implemented. This will lead to that the experiments in this section will only include 10 iterations. There will not be done any experiments with the different values for the drag force in these experiments, because the simulation time is to short to see any big differences. The most important aspect of the experiments in this section, is to get the Drag Force Model and the vehicle in Carla to copy the same forward speed as the restAPI over the driving cycle of 10 iterations. If the DT generates the same driving cycle, it shows that the prototype can be connected to a PT, and that would be a good result, if this project were to be further developed.

The first experiment can be seen in figure 5.13, where the green line represents the driving cycle for the restAPI and the orange line represents the driving cycle in Carla. The rolling friction is set to 0 and this leads to that Carla and the Drag Force Model got the same driving cycle, so the blue line is hidden by the orange line. The results that are generated in figure 5.13 are not accepted, because the driving cycle for the DT is different then the one that is generated by the PT.

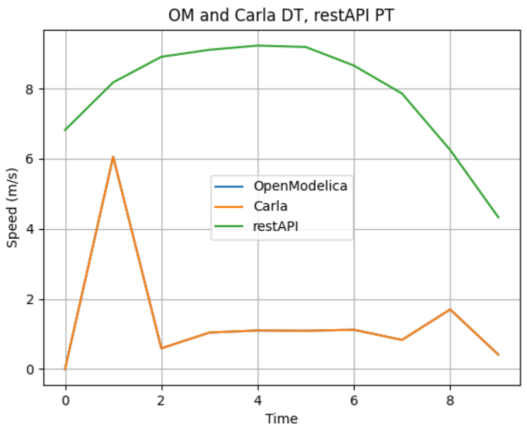


Figure 5.13: 10 iterations with the DT connected to the PT

As illustrated in figure 5.13, Carla is starting at a speed of 0, while the restAPI is starting at a speed of 7. The vehicle in Carla tries to accelerate to get up to the input speed from the restAPI, but it crashes after one iteration. The vehicle in Carla should automatically have the same starting speed as the input from the restAPI, because from this figure it looks like the DT is one step behind the restAPI. While reading more about the set functions in the Python API reference, it got figured out that the velocity in Carla will be updated two ticks after the velocity is set[20]. So in addition to make sure that the speed in Carla were set to the same speed as the input from the restAPI. There were implemented two new ticks in Carla right after the input speed from the restAPI were set, as illustrated in the pseudo-code in Listing 5.2. The speed in Carla will then be measured right after the two world ticks, and the measured velocity from Carla will be used as the input to the Drag Force Model.

```
#1. Get input from the restAPI.
#2. Set the velocity in Carla with the input from the resful
    API.
#3. Add two world ticks to update the speed in Carla.
world.tick()
world.wait_for_tick()
world.tick()
world.wait_for_tick()
#4. Measure the velocity in Carla
#5. Set the velocity for the Drag Force Model with the measured
     velocity from Carla.
```

Listing 5.2: Adding two ticks to Carla to load the frame where the speed is updated.

When the two new world ticks where added to the Carla environment, there where done a new experiment. The result for this experiment is illustrated in Figure 5.14. The driving cycle for the PT and the DT are looking similar, so it helped by adding the two world ticks. Even though the cycle looks similar, these results are still not accepted. The speed for the PT and the DT needs to be the same to be able to represent the same scenarios.

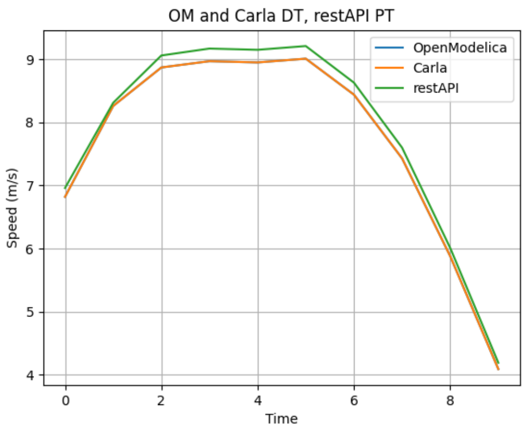


Figure 5.14: 10 iterations with the DT connected to the PT

Another experiment was done where the input speed from the restAPI got divided by 5 to see if it would cause any difference if the Digital Twin were working with smaller numbers for each iteration. This experiment is illustrated in figure 5.15, and the result for this experiment looks better. This could be random because the acceleration between the iterations should not matter in the Carla simulator. In addition, it is illustarted that the green and orange lines split after eight iterations, so the results are not accepted. It will be difficult to further tune the connection without any implemented methods of steering the vehicle in Carla. However, the experiments show that the DT can be simulated with input data from the restAPI, which is one important achievement for further developing the prototype.

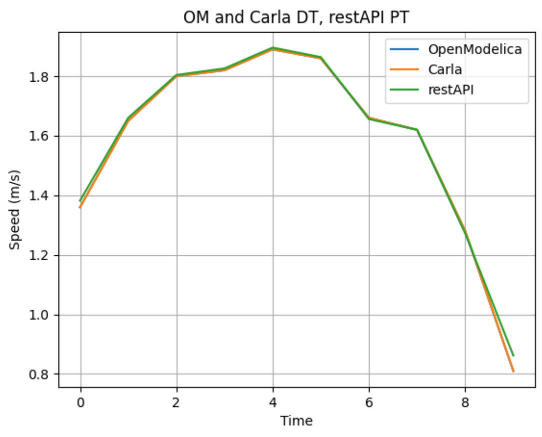


Figure 5.15: 10 iterations with the DT connected to the PT

Chapter 6

Discussion

The discussion will start by introducing the features explored with Open-Modelica together with the methodologies of how the tool can be connected to a Digital Twin. Further, the method of connecting Carla to the DT will be discussed, and at last the discussion related to the configuration that were developed in this project will be presented.

In OpenModelica, two different methods of working with input data were acquired, one by working with historical data from the text files and one where the parameters get changed directly. These two methods were a good base for connecting OM to the DT because it presents two possibilities on how the different models can be connected. the Simple Car Model was mainly implemented to test the method of working with historical data to see if it could generate the same driving cycle. The Simple Car Model was able to re-simulate the scenario, but it does not return any valuable information to the Digital Twin to process. The limitation of the mechanical blocks is that they are designed for creating Digital Models. It is an excellent method if the goal is to design a new system, but the DT should be able to monitor the mechanical parts. If the DT can monitor these parts, the system can gain valuable information related to predictions of further repairs or recommendations for replacing the different parts. However, to re-design the mechanical OpenModelica block to return this kind of information, requires more knowledge in the Modelica coding language and research which were out of reach for this project. The method where the parameters get directly changed with the Python API was a good approach because this is an essential feature so the parameters in the OM models can be updated in real-time. The method of connecting the constant blocks to the initialization parameters worked, so this method can be used with different parameters that cannot be accessed through the API. The Drag Force Model was proved to be a useful model in this project because the model applies the drag force equation to the digital environment. The main feature of this model is the *dragF* block that was designed by Massimo Ceraolo in [5], and the method of integrating this block to a new OM model worked nicely, as presented in the experiments. Without prior knowledge in OM, the methods acquired to connect OpenModelica into the prototype were good choices. However, there should have been spent more time on researching the Modelica coding language to possibly design some methods of steering the vehicle in Carla by using OpenModelica. Anyhow it is not guaranteed that the vehicle should be controlled by OpenModelica, since Carla already got a integrated PID controller that possibly can be used, with further research.

The method of initializing Carla to the Digital Twin together with the GET and SET functions acquired from the API were good choices, because it was able to simulate and test the different OM models as planned. Early in the project, Carla was used as a tool to generate data so the different models in OM could get tested. To generate this data from Carla, it got decided to turn on autopilot as the driving algorithm. The reason for this is because Carla can generate the same data as would be returned from a physical vehicle, which represents speed, acceleration and forward force. Further, because there were not acquired no other methods than using autopilot and the function vehicle.set_velocity() to control the vehicle in Carla. The project could not fully explore the method of adding the drag force equation to the Carla environment. If the autopilot got turned off, and the vehicle.set_velocity() function got deployed, the vehicle in Carla would drive straight and crash after a couple of seconds. The same problem was encountered when the restAPI got connected as the PT. So there should have been spent more time on researching a method for steering the vehicle in Carla as addressed earlier in this chapter. However, the restAPI and the Carla environment were not initialized in different maps. So by implementing another method of steering, the vehicle would still crash while it is connected to the restAPI. Therefore, there should also have been researched a method of implementing a blank world to Carla without any obstacles to configure the tools properly.

The configuration that is developed in this project has an interesting feature, the reason for this is because with when the restAPI deploys a step. The step will be simulated in Carla and the Drag Force Model, further the Drag Force Model will return on action to Carla. Which lead to that there will be deployed two steps in the Carla environment. This may be a configuration that possibly can be used to have the DT one step ahead of the PT, which is an interesting method of working with the tools. However, there will be required further research on how the DT can generate an action that should be returned to the PT based on the generated steps in the digital environment. In addition, because there were not implemented any states in the digital environment that could be returned to the physical twin, the prototype DT designed in this project has the characteristics of a Digital

Shadow [12]. While reflecting on the configuration developed after what has been learned in this project, there will be presented a new configuration that possibly be designed to convert the connection between OpenModelica and Carla from a DS to a DT. The configurations is illustrated in figure 6.1 on the next page. The main concept of this configuration, is to change the methodology of re-simulating scenarios with OpenModelica, by rather re-simulating the scenarios in Carla. This could be a interesting configuration to further research because Carla is mainly developed for training and validating autonomous driving systems. The idea is then to implement a DT-To-PT configuration where Carla can simulate real-time "what if?" analysis and return an action to the PT based on the scenario in Carla, which generated the outcome with the highest reward or accuracy. If this feature were researched and implemented, the configuration in the DT could return an action back to the PT. Which would represent a bidirectionally data-exchange between the DT and the PT, which represents the characteristic of a DT[12][14][13][2].

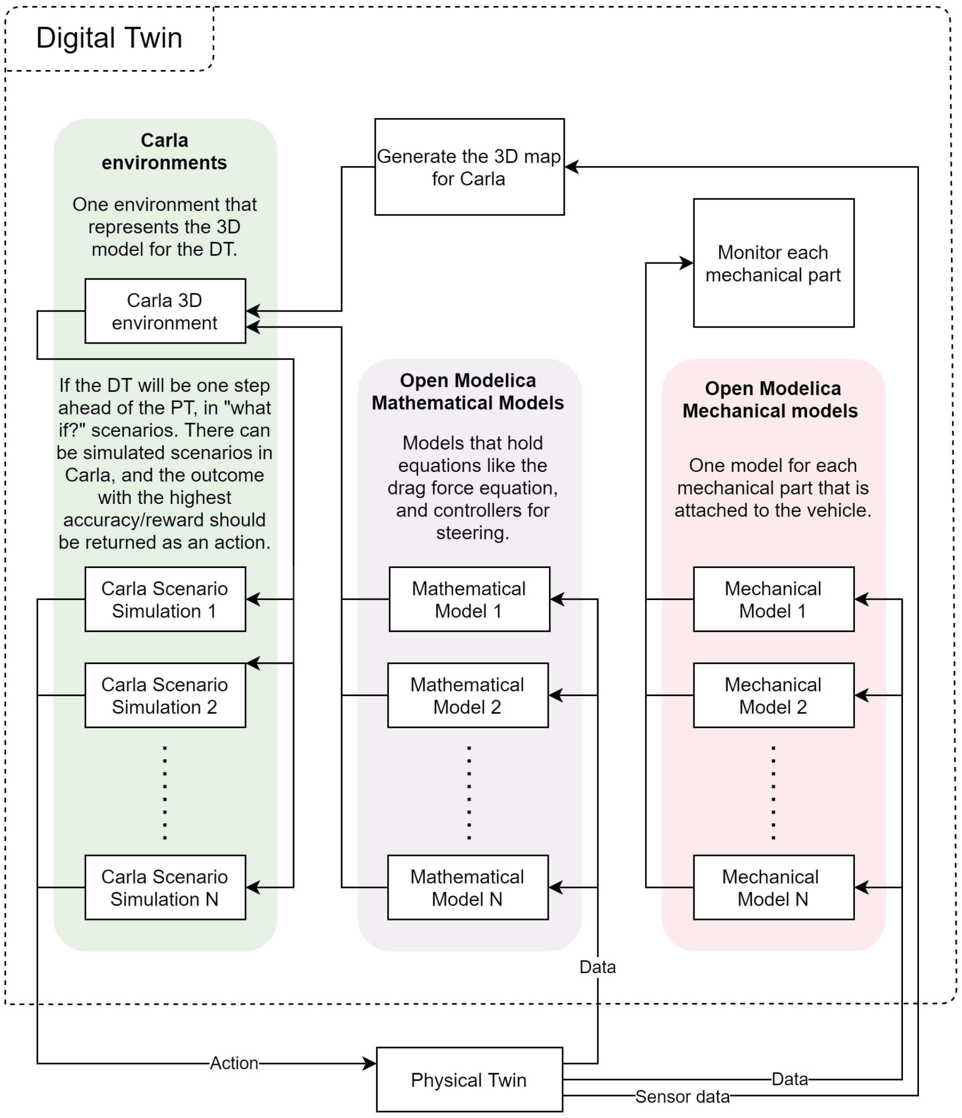

Digital Twin
Carla environments
One environment that represents the 3D model for the DT.
Carla 3D environment
If the DT will be one step ahead of the PT, in "what if?" scenarios. There can be simulated scenarios in Carla, and the outcome with the highest accuracy/reward should be returned as an action.
Carla Scenario Simulation 1
Carla Scenario Simulation 2
Carla Scenario Simulation N
Generate the 3D map for Carla
Open Modelica Mathematical Models
Models that hold equations like the drag force equation, and controllers for steering.
Mathematical Model 1
Mathematical Model 2
Mathematical Model N
Monitor each mechanical part
Open Modelica Mechanical models
One model for each mechanical part that is attached to the vehicle.
Mechanical Model 1
Mechanical Model 2
Mechanical Model N
Data
Action
Physical Twin
Data
Sensor data

Figure 6.1: This figure is an example of an configuration that can be further researched.

6.1 The research questions

The first research question was to find methodologies to connect open Modelica to the digital twin by adding mechanical or mathematical models. There were acquired two methods of connecting OpenModelica to the digital twin, one where historical data can be simulated and one where the data can be updated in real-time. Both methodologies worked well to connect the digital twin tool, as shown in the experiments. The mathematical model that got connected added a feature to the digital environment, but the mechanical model did not return any valuable information to the Digital Twin. There will be needed to do further research related to the mechanical blocks in OpenModelica to find methods of making them useful for the Digital Twin.

The second research question was to determine how Carla and OpenModelica could be configured to serve as a Digital Twin. The idea behind the configuration that was designed in this project was to use OpenModelica to add a step to Carla, so the vehicle in the digital environment could be one step ahead of the physical twin, which possibly could lay a base for further research regarding real-time ”what-if?” scenarios. The mechanical model connected to the Digital Twin was used to re-simulate the scenario but did not return any valuable information for this configuration. Further, from the knowledge gained in this project, another configuration was presented, which could be further researched. The idea for the new configurations is that Carla can be used to re-simulate scenarios since the architecture for training and validating autonomous driving systems is already implemented in Carla.

Chapter 7

Conclusion

This project aimed to determine how Carla and OpenModelica can be connected and configured to serve as a Digital Twin. In OpenModelica, two methods of working with data outside the OpenModelica environment got used so the tool could get connected to the Digital Twin. The first method represents a way of working with historical data to re-stimulate scenarios, and the second method got used to update the parameters in OpenModelica in real-time. Both methods worked well to connect OpenModelica to a Digital Twin from the experiments done in this project. The mechanical model connected to the Digital Twin was able to re-simulate the scenario. Further, the mathematical model connected to the Digital Twin proved to add a feature from the physical environment to the digital environment. The Carla simulator got connected to the Digital Twin as the visual simulation tool, but the simulator also got used to test the different OpenModelica models early in the project. Simulation tools like Carla and the LGSV-simulator were shown to be good tools for generating states for prototyping a Digital Twin before connecting the system to a Physical Twin. The experiments illustrated that the Digital Twin prototype developed in this project could connect to a Physical Twin. However, it will require more tuning and research to make the configurations optimal. The conclusion for this project states that the methodologies gathered to connect Carla and OpenModelica as a Digital Twin lays a good base for further research related to the development of a Digital Twin.

Bibliography

[1] Solomon W Golomb. 'Mathematical models: Uses and limitations'. In: *IEEE Transactions on Reliability* 20.3 (1971), pp. 130–131.

[2] Michael Grieves. 'Digital twin: manufacturing excellence through virtual factory replication'. In: *White paper* 1 (2014), pp. 1–7.

[3] Jacob D Hochhalter, William P Leser, John A Newman, Edward H Glaessgen, Vipul K Gupta, Vesselin Yamakov, Stephen R Cornell, Scott A Willard and Gerd Heber. *Coupling Damage-Sensing Particles to the Digitial Twin Concept.* National Aeronautics and Space Administration, Langley Research Center, 2014.

[4] Jerzy Ejsmont, Leif Sjögren, Beata Świeczko-Żurek and Grzegorz Ronowski. 'Influence of road wetness on tire-pavement rolling resistance'. In: *Journal of Civil Engineering and Architecture* 9.11 (2015), pp. 1302–1310.

[5] Massimo Ceraolo. *Simplified Modelling of Electric and Hybrid Vehicles.* Sept. 2017. URL: http://omwebbook.openmodelica.org/SMEHV (visited on 07/05/2021).

[6] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez and Vladlen Koltun. 'CARLA: An Open Urban Driving Simulator'. In: *Proceedings of the 1st Annual Conference on Robot Learning.* 2017, pp. 1–16.

[7] Michael Grieves and John Vickers. 'Digital twin: Mitigating unpredictable, undesirable emergent behavior in complex systems'. In: *Transdisciplinary perspectives on complex systems.* Springer, 2017, pp. 85–113.

[8] Peter Fritzson, Bernhard Bachmann, Kannan Moudgalya, Francesco Casella, Bernt Lie, Jiri Kofranek, Massimo Ceraolo, Christoph Nytsch Geusen and Luigi Vanfretti. *Introduction to Modelica with Examples in Modeling, Technology and Applications.* Sept. 2018. URL: http://omwebbook.openmodelica.org/ (visited on 28/04/2021).

[9] Alexander Koumis. *Gitub - Clarification regarding player measurements[Discussion post]*. Mar. 2019. URL: https://github.com/carla-simulator/carla/issues/355#issuecomment-477472667 (visited on 25/03/2021).

[10] Aniruddh Mohan, Shashank Sripad, Parth Vaishnav and Venkatasubramanian Viswanathan. 'Automation is no barrier to light vehicle electrification'. In: *arXiv preprint arXiv:1908.08920* (2019).

[11] Adrian Pop. *Github - OpenModelica Examples - OMExamples/FMUResourceExample/*. Nov. 2020. URL: https://github.com/adrpo/OMExamples/tree/master/FMUResourceExample (visited on 08/04/2021).

[12] Aidan Fuller, Zhong Fan, Charles Day and Chris Barlow. 'Digital twin: Enabling technologies, challenges and open research'. In: *IEEE Access* 8 (2020), pp. 108952–108971.

[13] David Jones, Chris Snider, Aydin Nassehi, Jason Yon and Ben Hicks. 'Characterising the Digital Twin: A systematic literature review'. In: *CIRP Journal of Manufacturing Science and Technology* 29 (2020), pp. 36–52.

[14] Jakob Trauer, Sebastian Schweigert-Recksiek, Carsten Engel, Karsten Spreitzer and Markus Zimmermann. 'What is a Digital Twin?–Definitions and Insights from an Industrial Case Study in Technical Product Development'. In: *Proceedings of the Design Society: DESIGN Conference*. Vol. 1. Cambridge University Press. 2020, pp. 757–766.

[15] Concetta Semeraro, Mario Lezoche, Hervé Panetto and Michele Dassisti. 'Digital Twin Paradigm: A Systematic Literature Review'. In: *Computers in Industry* 130 (2021), p. 103469.

[16] The European Space Agency. *Digital Twin Earth*. URL: https://www.esa.int/ESA_Multimedia/Images/2020/09/Digital_Twin_Earth.

[17] CARLA Simulator. *1st. World and client*. URL: https://carla.readthedocs.io/en/latest/core_world/ (visited on 25/03/2021).

[18] CARLA Simulator. *2nd. Actors and blueprints*. URL: https://carla.readthedocs.io/en/latest/core_actors/ (visited on 22/03/2021).

[19] CARLA Simulator. *Core concepts*. URL: https://carla.readthedocs.io/en/0.9.11/core_concepts/#first-steps (visited on 08/05/2021).

[20] CARLA Simulator. *Python API reference*. URL: https://carla.readthedocs.io/en/latest/python_api/ (visited on 11/05/2021).

[21] Modelica Documentation. *Modelica.Blocks.Sources.CombiTimeTable*. URL: https://build.openmodelica.org/Documentation/Modelica.Blocks.Sources.CombiTimeTable.html (visited on 05/04/2021).

[22] Néstor Subirón. *carla/LICENSE - Github*. URL: https://github.com/carla-simulator/carla/blob/master/LICENSE (visited on 26/05/2021).

[23] OMPython. *OpenModelica Python Interface*. URL: https://www.openmodelica.org/doc/OpenModelicaUsersGuide/latest/ompython.html (visited on 11/05/2021).

[24] Tao Yue, Paolo Arcaini and Shaukat Ali. 'Understanding Digital Twins for Cyber-Physical Systems: A Conceptual Model'. In: *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation* (Will be published in 2021).