Metamorphic Testing Applied on Baidu Apollo Autonomous Driving System

Submitted October 24, 2024, in partial fulfillment of
the conditions for the award of the degree **BSc Computer Science.**

Zhichao He
20032295

Supervised by Dr. Dave Towey

School of Computer Science, University of Nottingham Ningbo China

Acknowledgements

This project would not have been accomplished without the guidance of my supervisor, Dr. Dave Towey, and I extend my sincerest thanks to him for helping me at every stage of the project and supporting me throughout the difficult times. Additional recognition goes to Yifan Zhang, a PhD student from the University of Nottingham Ningbo China, who shared some experience with me about running Apollo software as well as dealing with specific bugs.

I would also like to express my gratitude to the researchers at the University of Wollongong, Dr. Zhiquan Zhou and Jiancheng Han, for giving me guidance of implementing automatic testing in Baidu Apollo.

Abstract

Nowadays, autonomous vehicles have become a popular technology which is being developed and maintained by several well-known companies, including Tesla, Baidu and Google. The software and hardware products of self-driving vehicles are gradually commercialised, and the public has opportunities to experience them. Nevertheless, recent accident reports of self-driving cars have driven people to think about safety issues and software vulnerabilities of autonomous driving systems. Therefore, numerous and reliable tests should be conducted to make sure autonomous vehicles work safely. However, testing autonomous vehicles faces an oracle problem, meaning that it is difficult to define a set of rules to determine whether the vehicle behaves appropriately in specific scenarios, because the driving conditions of the autonomous vehicle are complicated and various situations should be considered. Metamorphic testing is introduced to test autonomous driving systems, because metamorphic testing does not require to examine the input and output mapping of the software. It first generates a source test case and uses metamorphic relations to generated a similar follow-up test case. Then, both test cases are executed and their results are compared regarding the metamorphic relations. If the relation is violated, it indicates that there is a problems with the software. Fuzz testing is a testing approach which generates random inputs and sends them to the software to test its robustness. In this project, both metamorphic testing and fuzz testing are utilised to test one of the well-known autonomous driving systems, Baidu Apollo.

Contents

List of Figures

List of Abbreviations

AILab	Artificial Intelligence Laboratory
ALV	Autonomous Land Vehicle
API	Application Programming Interface
AV	Autonomous Vehicle
CPU	Central Processing Unit
FT	Fuzz Testing
GB	Gigabyte
GPU	Graphic Processing Unit
MFT	Metamorphic Fuzz Testing
MR	Metamorphic Relation
MT	Metamorphic Testing
Navlab	Navigation Laboratory
OpenCV	Open Source Computer Vision Library
OS	Operating System
ROS	Robot Operating System
SAGE	Scalable Automated Guided Execution
SUT	System Under Test
SUV	Sport Utility Vehicle
UNNC	University of Nottingham Ningbo China

Chapter 1

Introduction

In recent years, driverless vehicles have become a thriving technology, and *Autonomous Vehicles (AVs)* will probably change human lifestyle and even the traffic conditions in the city [1], which means AVs will replace human drivers in the future and the communications between them can alleviate traffic congestion. Nevertheless, the rapid development of AVs leads to the public's concern about road safety issues. This chapter introduces the background information of AVs and the relevant safety problems. Then the motivation, aims and objectives of the project are described. Lastly, the outline of the dissertation is presented.

1.1 Background

Many well-known companies have invested heavily in the development of driverless vehicles. For instance, Baidu Apollo [1], a pioneering AV system, has been in development since 2015. It is now providing an open, integrated and secure software platform for its partners in the automated driving field [2]. Apollo is an open-source project, maintained by many developers, and can be conveniently downloaded and installed by users. Automobile manufacturers and individual car owners can install Apollo software in the vehicles with supporting hardware to achieve autonomous driving [2].

AVs will play a primary role in future urban transportation design, as they can provide improved safety, greater productivity, higher accessibility, better road efficiency, and a positive impact on the environment [3]. In recent years, research on autonomous systems has made tremendous progress due to the advancement in computing power and the reduction in the cost of sensing and computing technologies, which lead to an increasingly mature level of technological readiness for AVs [3].

1.2 Motivation

An research on autonomous driving testing [4] which was conducted in a real world situation, aimed at demonstrating the AV's ability to manage complex driving scenarios. The AV under test performed even better than a human driver when dealing with emergencies. Consequently, the results showed that the AVs can reduce the impact of human factors

[1]Baidu Apollo Official Website. Available at: `https://apollo.auto/`

[2]Baidu Apollo GitHub Documentation. Available at: `https://github.com/ApolloAuto/apollo`

on the occurrence of traffic accidents, meaning that an AV will not have fatigued driving issues and missed operations as a conventional driver would have. Consequently, AVs have the ability to manage real and complex situations. However, AVs cannot guarantee absolute safety. A research about AV accidents [5] pointed out that a fleet with AVs have a significantly higher probability of having rear-end collisions than a fleet with only conventional vehicles. The reason behind this is that human drivers in conventional vehicles are not familiar with the behaviors of self-driving cars, because AVs obeys the traffic rules strictly and do not have aggressive driving that a human driver may have. Therefore, read-end accidents are more likely to happen when AVs and conventional vehicles drive with each other.

Specifically, an accident of a Tesla AV with Autopilot [3] took place in the United States in 2019, and the driver of the self-driving car was killed [6]. The report of the accident [6] claimed that the vehicle was in autonomous driving mode when crashed into the other vehicle on the highway, and there was no data of braking until the collision took place. Similar problems also exist in other AV brands including Google Waymo [7], Uber [8] and Navya automated bus driving system [9]. Therefore, AVs still need technical refinement for widespread application in the future.

One of the reasons for the high incidence of accidents with AVs is that AV testing is difficult to implement. In traditional testing, for example, Java unit test [10] defines a set of behaviours that the software should and should not have. On the contrary, it is too complicated to create an integrated system specification against which the autonomous vehicle's behaviour can be checked [11], as it essentially involves replicating the operational logic of a human driver. This is the drawbacks of traditional testing approaches, that they cannot test a software with various functionalities and complicated structures. The problem is known as the Oracle problem [12], which means there is no available mechanism that determines whether the output of a software is correct because it is difficult to create such kind of mechanism for AVs, considering the complexity of the code structure. Consequently, it is vital to test the current autonomous driving systems with two novel testing approaches, *Metamorphic Testing (MT)* [13] and *Fuzz Testing (FT)* [14]. These two methods are suitable for testing AVs because they avoid the drawback of traditional testing methods and alleviate the Oracle problem. Therefore, they can find vulnerabilities that cannot be found by conventional methods. More information about FT and MT are introduced in Chapter 2.

1.3 Aims and Objectives

This project is a partial replication of a successful research of AV testing [11] adopting *Metamorphic Fuzz Testing (MFT)* [15] which is a combination of MT and FT. Similar methodologies in this research [11] are likely to be used in this project implementation, but the test design and procedures are ultimately distinctive. The purpose of this project is to test the safety and reliability of self-driving systems using MT and FT. The testing is particularly at the software level, because real-world testing is impractical without

[3]Tesla Autopilot Website. Available at: `https://www.tesla.com/autopilot`

supporting hardware and physical vehicles [4]. The target software is Baidu Apollo and the testing method is MFT [11]. The simulation software used for testing is Dreamview [5]. Based on the test results, the reliability and robustness of the Baidu Apollo are analysed. The (quality) of the metamorphic relations [16] and fuzz function is evaluated [17].

The key objectives of the project are:

1. Investigate the functionalities of specific working modules of Baidu Apollo relevant to the testing.
2. Generate Metamorphic Relations (MRs) suitable for testing AVs.
3. Design and implement automatic testing scripts with the assistance of Dreamview GUI.
4. Run the test scripts and collect relevant data from driving simulations.
5. Analyse the results and assess the reliability of the Baidu Apollo system. Identify the possible causes of the problem and propose solutions accordingly.
6. Summarize and reflect on how the fuzz function and MRs are implemented and improved based on the testing results.

1.4 Dissertation Outline

This chapter has summarized a brief introduction to AVs as well as corresponding safety problems. It then presented the motivation of using MFT to test AV systems. Finally it outlined the aims and objectives of this project. Chapter 2 discusses detailed background information and related research of AVs, MT and FT. Chapter 3 describes the experimental design and test implementation of the project. Chapter 4 presents the results and analysis obtained from the experiments. Finally, Chapter 5 discusses the project management, summary and reflections.

[4]Baidu Apollo Installation Documentation. Available at: `https://github.com/ApolloAuto/apollo#installation`

[5]Dreamview Documentation. Available at: `https://github.com/ApolloAuto/apollo/blob/master/docs/specs/dreamview_usage_table.md`

Chapter 2

Background and Related Work

This chapter introduces detailed background information of AV, MT and FT. This chapter also includes related research into AV testing and corresponding methodologies and results.

2.1 AVs and Baidu Apollo

The first semi-automated car, shown in Figure 2.1, was developed in 1977 at Tsukuba Mechanical Engineering Laboratory (TMEL) in Japan [18], and required specially labelled streets that were detected by cameras on the vehicle and the path of the vehicle was computed by a computer. The vehicle eventually accelerated to a speed of 30 km/h supported by an elevated track that assisted it in following a prescribed route [18]. This event marked the beginning of autonomous driving.

Figure 2.1: First AV at TMEL

Later in the 1980s, a great breakthrough of AVs took place, as a variety of successful projects were developed. For example, Carnegie Mellon University's (CMU) Navigation Laboratory (Navlab) [19] built a series of computer-controlled vehicles that can drive themselves or assist the human driver in driving. Figure 2.2 shows a driverless van built by Navlab in the 80s.

Figure 2.2: Navlab Autonomous Van

Figure 2.3: Autonomous Land Vehicle

Autonomous Land Vehicles (ALVs) [20] are another AV project carried out by CMU. A report about ALV [21] stated that ALV is a mobile robot with vision and control systems that can drive the vehicle in a real world environment, as shown in Figure 2.3. The system has been tested in both indoor and outdoor environments. In the best run, the vehicle finished a twenty meter route at the speed of 2 cm/sec. The low speed is impractical in real environments and it was caused by the immaturity of image processing, which is optimised in later generations [21].

It is currently a booming period of AVs. With the development of deep learning, especially deep reinforcement learning [22], the performance of the perception and prediction algorithms used in autonomous driving systems are constantly improving. Baidu Apollo [23] and Google Waymo [1] are two fully-fledged driverless vehicle projects in the field, shown in Figure 2.4 and 2.5. The two AV brands both launched several versions of their autonomous driving product, and are experienced by many users. The feedback of their products are generally good, and will be commercialised in the near future. Apollo Although Uber has given up the self-driving development plan, and sold their business to Aurora Innovations [24], they have contributed a lot of experience to the development of AVs. Developers of autonomous driving systems are currently at the phase of testing and optimising the autonomous driving systems to ensure that they are safe enough for public use [3].

Figure 2.4: Baidu Apollo AV

Figure 2.5: Google Waymo AV

AV systems share a similar working mode, and they all have common functional com-

[1]Google Waymo Official Website. Available at: https://waymo.com/

ponents [3]. Generally, these elements include algorithms for scene recognition, route planning and vehicle control, as well as open source software stacks including Robot Operating System (ROS) [25] and Open Source Computer Vision Library (OpenCV) [26], and data sets for training neural networks. The report which introduces the AV functions [27] shows that, the scene recognition part of AV systems requires algorithms for localization, object-detection, and object-tracking. AV localization [28] uses cameras and sensors to acquire information surrounding the AV in all weather and traffic conditions, including precise positions and orientations of the AV. Object-detection involves the detection and perception of vehicles, pedestrians, and traffic signals. Related technologies include Deformable Part Models (DPM) [29] for object classification and 3D Lidar sensors [30] for point cloud generation. Object-tracking is achieved by comparing different frames of the video recorded by the cameras. By associating the results of detected objects with other frames on a time basis, the trajectories of moving objects can be predicted [27].

Because AV has a very complicated software architecture and utilises various technologies from a broad range of disciplines, developing AV needs numerous fully-fledged open source Application Programming Interfaces (APIs) and platforms, for example, ROS, OpenCV and CUDA. These are essential elements in AV development. ROS [2] provides useful libraries and toolkits to support robot engineers creating robot-related applications. AV is also a type of robot which is applied on vehicles. Therefore, ROS is a fundamental part of AV systems that achieve message publishing and subscribing and other essential functions. OpenCV [26] is a standard computer vision library that contains image processing algorithms for object recognition, motion detection and analysis. CUDA [3] is a framework for general-purpose computing on Graphic Processing Units (GPUs), and AV requires massive computation when executing certain algorithms. Consequently, CUDA can speed up the commputation and improve the performance of AVs.

Another significant part of an AV system is the datasets. Point cloud data, image data and map information take up a lot of space of the AV software package, and these datasets are used in autonomous driving simulation and map generation [31]. After downloading Baidu Apollo from their GitHub repository in a remarkably long time, it turned out approximately 90% size of the software are made up of data, and the code and documentations represent only a small percentage of the software.

Specifically, Baidu Apollo 3.0 [4] has an overall software architecture as shown in Figure 2.6 with corresponding software modules including perception, planning, prediction and control. Firstly, the perception module works out the information of the obstacles near the vehicle. Secondly, the prediction module used the information and specific OpenCV algorithms to predict the trajectories of the vehicles, pedestrians and other obstacles near the AV. After analysing the possible behaviours of the obstacles, the planning module generates an appropriate routing for the AV to drive. Finally, the control module processes the routing request and send instructions to the hardware system of the AV to physically drive it.

[2]ROS Online Documentation. Available at: `http://wiki.ros.org/Documentation`

[3]Nvidia CUDA Documentation. Available at: `https://developer.nvidia.com/cuda-zone`

[4]Baidu Apollo 3.0 Documentation. Available at: `https://github.com/ApolloAuto/apollo/tree/r3.0.0`

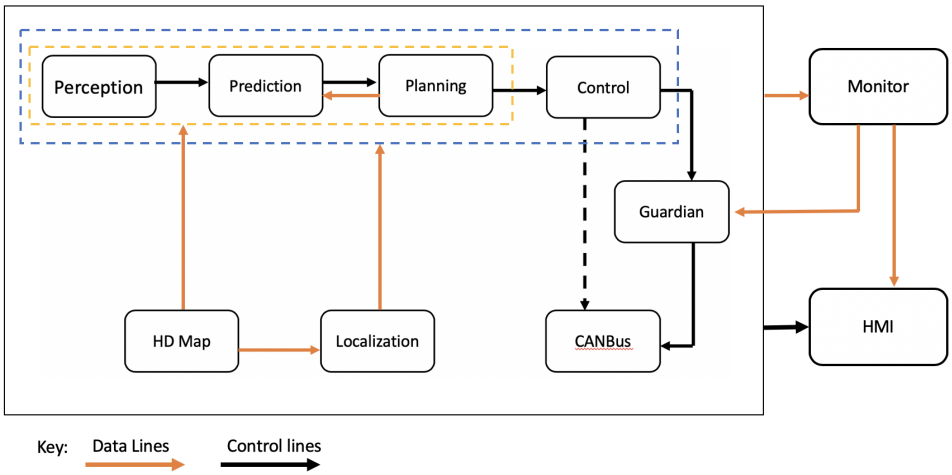

Figure 2.6: Baidu Apollo Software Architecture

2.2 Oracle Problem and Metamorphic Testing

In software testing, an oracle [12] is a mechanism that determines whether the output of the *System Under Test (SUT)* is correct or not. As a simple example, we can define an oracle for a quick sort algorithm. For a input unsorted array, the output array is correct if it is incremental. Otherwise, the output is incorrect. Traditionally, the oracle is generated by developers who have associated knowledge about the SUT [12], but it is impractical when the SUT becomes complicated. To solve this problem, test oracle automation [32] was introduced to replace human generated oracles. For certain software in real life, there is no suitable oracle for testing the correctness of the outputs for given inputs. This is the oracle problem and these software are referred to as untestable [33]. In this case, even test oracle automation is not applicable. Testing autonomous vehicles faces such an oracle problem because it is not deterministic whether the vehicle's behaviour and decisions are appropriate under specific conditions [11].

Metamorphic Testing (MT) was introduced to alleviate the oracle problem [11]. An essential component of MT is a set of *Metamorphic Relations (MRs)* [13], which are fundamental properties of the SUT in relation to multiple input and output pairs. When implementing MT [33], some program inputs (called source inputs) are first generated as source test cases, on the basis of which an MR can then be used to generate new inputs as follow-up test cases. In contrast to the traditional way of verifying the test results of each individual test case, MT compares the source and follow-up test cases and their outputs with the corresponding MRs. Continuing from the previous quick sort algorithm example, now use MT to check whether the algorithm performs correctly. Firstly, the source test case is applying the algorithm to an unsorted array A[3,2,4,1,5] and the output should be a sorted array [1,2,3,4,5]. Then, a follow-up test case is generated whose input is the same as the input array in the source test case, except that the new input array A'[2,5,1,4,3] has a different order but the same set of numbers. The outputs of source test case and follow-up test case should be the same, because the sorting result of the same set of numbers should be identical. Otherwise, this implementation of the quick sort algorithm is wrong because it violates the MR of this algorithm. In this example, the MR utilised is

that using a quick sort algorithm to sort two arrays with the same element set, but in a different order, the result should be the same, and this is a necessary property of a quick sort algorithm.

There are several formalised and symbolic definitions of MT-related concepts that were introduced in an ACM Computing Survey of MT [13], and are quoted to provide a better comprehension of MT and MR:

Definition 1 (Metamorphic Relation). Let f be a target function or algorithm. A ***metamorphic relation*** is a necessary property of f over a sequence of two or more inputs $\langle x_1, x_2, ..., x_n \rangle$, where $n \geq 2$, and their corresponding outputs $\langle f(x_1), f(x_2), ..., f(x_n) \rangle$. It can be expressed as a relation $R \subseteq X^n \times Y^n$, where $\subseteq$ denotes the subset relation, and X^n and Y^n are the Cartesian products of n input and n output spaces, respectively. Following standard informal practice, we may simply write $R(x_1, x_2, ..., x_n, f(x_1), f(x_2), ..., f(x_n))$ to indicate that $\langle x_1, x_2, ..., x_n, f(x_1), f(x_2), ..., f(x_n) \rangle \in R$.

Definition 2 (Source Input and Follow-up Input). Consider an MR $R(x_1,\ x_2, ..., x_n,\ f(x_1),\ f(x_2), ..., f(x_n))$. Suppose that each x_j $(j = k+1, k+2, ..., n)$ is constructed based on $\langle x_1,\ x_2, ..., x_k,\ f(x_1),\ f(x_2), ..., f(x_k) \rangle$ according to R. For any $i = 1, 2, ..., k$, we refer to x_i as a ***source input***. For any $j = k+1, k+2, ..., n$, we refer to x_j as a ***follow-up input***. In other words, for a given R, if all source inputs $x_i (i = 1, 2, ..., k)$ are specified, then follow-up inputs $x_j (j = k+1, k+2, ..., n)$ can be constructed based on the source inputs and, if necessary, their corresponding outputs.

Definition 3 (Metamorphic Testing). Let P be an implementation of a target algorithm f . For an MR R, suppose that we have $R(x_1,\ x_2, ..., x_n,\ f(x_1),\ f(x_2), ..., f(x_n))$. ***Metamorphic testing*** based on this MR for P involves the following steps:

1. Define R' by replacing f by P in R.
2. Given a sequence of source test cases $\langle x_1, x_2, ..., x_k \rangle$, execute them to obtain their respective outputs $\langle P(x_1), P(x_2), ..., P(x_k) \rangle$. Construct and execute a sequence of follow-up test cases $\langle x_{k+1},\ x_{k+2}, ..., x_n \rangle$ according to R' and obtain their respective outputs $\langle P(x_{k+1}),\ P(x_{k+2}), ..., P(x_n) \rangle$.
3. Examine the results with reference to R'. If R' is not satisfied, then this MR has revealed that P is faulty.

2.3 Fuzz Testing

Another technique that can be adopted in testing driverless vehicles is *Fuzz Testing (FT)* which rapidly generates random inputs to the software for identifying bugs and potential vulnerabilities [14]. FT is an efficient approach of test case generation. Compared to traditional test case generation performed by humans, FT can generate significantly more test cases. The amount and features of such randomly-generated test cases from FT are difficult for human to think of because human engineers normally generate a limit amount of sensible test cases based on their understanding of the software. They know which kind of test cases may cause failures of the software, but they are unlikely to create random test

cases. Nevertheless, in specific scenarios, it is these random tests cases that are more likely to reveal vulnerabilities in the SUT [34]. For instance, FT can be used to examine the input check of software. Most software has user input checks and corresponding prompts in order to maintain its usability and regulate the data formats in the database, and this is usually due to irrational and invalid input attempts by users. Software designers can hardly think of all possible input scenarios that a user may generate, and thus FT can be introduced to exhaustively generate a wider range of input situations to test the input check integrity of the software.

Another reason why FT is suitable for discovering bugs in software is that hackers use FT to find software vulnerabilities and launch zero-day attacks [35]. A software with high security has poor usability and software with high usability is less secure. Consequently, there is a compromise between security and usability when developers design the systems. Because FT generates a variety of random inputs, and hackers take advantage of the broad coverage of the test cases to reveal bugs of the system more easily. If the system is not securely test, there are some vulnerabilities that will not be identified by the software developers. Consequently, when one of these vulnerabilities is happened to be encountered, a hacker can use the random inputs generated by FT to crack the system.

As described in a research [14] which categorised different fuzzing methods, data generation by fuzzing can be implemented in two ways. The first one is to generate data completely randomly without knowing the code details of the program. This method is known as *Blackbox fuzzing* [14] which was the originally proposed fuzzing concept. Although blackbox fuzzing is easy to implement and is widely used by software test engineers [36], it has drawbacks on the code coverage, meaning that test cases generated by blackbox fuzzing can not execute all the lines of the code in the software. The reason is that blackbox fuzzing is absolutely random. An example of blackbox fuzzing was given by Godefroid [37]. The then branch of the conditional statement $if\ (x == 10)\ then$ has only one in 2^{23} chances of being executed if x is a randomly chosen 32-bit input value. This is the reason why blackbox fuzz testing usually has a poor code coverage.

Whitebox fuzzing [38] was introduced to alleviate the code coverage issue, because it generates test cases assuming a complete knowledge base of the application code and behaviours [14]. This approach symbolically executes the program dynamically and gathers information on inputs from conditional statements encountered along the path order of code execution. The collected constraints are then systematically negated and yielded new inputs that will enter different execution paths of the code. For example, the symbolic execution of the previous conditional statement branch on the input x = 0 generates the branch condition that is represented as $x \neq 10$. After this condition is negated, it becomes $x = 10$, which makes the program follow the *then* branch which is not executed beforet. Scalable Automated Guided Execution (SAGE) [38] is a tool based on x86 instruction-level tracing and emulation for whitebox fuzzing and it was used to test Microsoft Excel. It turns out whitebox fuzzing is highly effective and performs faster than blackbox fuzzing. Furthermore, *Graybox fuzzing* [39] is a compromise of the two methods aiming to utilize the advantages of both. It uses only a minimal knowledge set of target application.

For instance, when building the environment of a driving simulation, a fuzz algorithm generates obstacles that can suddenly appear and disappear in the simulation environ-

ment. These obstacles are randomly created and refreshed frequently to better test the robustness of the self-driving system, because a well-developed autonomous vehicle should rapidly react to emergencies. Fuzz testing can generate thousands of variations of a scenario within a short time, and thus, it is an efficient way of test case generation. [11].

Chapter 2 summarized detailed definitions of MT and MR, and a recent testing research on Apollo. Next chapter introduces the test design and implementation of the present project.

Chapter 3

Design and Implementation

This chapter covers the design of the automatic testing in this project. It then specifies the Apollo installation details and Dreamview Simulation Environment GUI.

Although AV is a commercialised industry, many companies developing AV choose make their AV platforms open source. This is because an open platform motivates the development and testing of the AV software, and there are various open source AV software available on GitHub. Baidu Apollo is the target of this testing because it is one of the leading AV platforms and is being contributed by numerous developers. It is thus a more sophisticated software and also more challenging to find vulnerabilities of it.

All the versions of Baidu Apollo are only supported on Ubuntu OS [1] which is a Linux-based as the installation of Apollo requires the nVidia driver for the utilisation of GPU calculating instead of CPU. I previously tried to install Apollo on my laptop using VMWare virtual machine (VM) [2]. Although my laptop is equipped with an nVidia GPU, it cannot be detected by the Ubuntu Operating System (OS) in the virtual machine. Even if the virtual machine can recognize the graphics card, the computing power of the GPU is not high enough. This is because the AV system is based on deep learning which are computation-expensive when executing computer vision programs, and advanced GPUs with extra parallel computing capabilities are in demand. Consequently, my laptop is not capable of running Baidu Apollo at optimal conditions. Fortunately, there are Ubuntu machines with high-performance GPUs in the Artificial Interlligence Laboratory (AILab) [3] of the School of Computer Science, so that I can utilise both the Ubuntu OS and the high-quality GPU. I installed Apollo on one machine at the AILab.

3.1 Baidu Apollo and Essential Software Installation

As described in Chapter 2, Baidu Apollo is a complicated software containing a variety of technologies and APIs which are used for data collection and processing. Therefore, it cannot be directly compiled in the command line and requires specific OS and supporting software.

[1] Ubuntu Official Website. Available at: `https://ubuntu.com/`

[2] VMWare Workstation Website. Available at: `https://www.vmware.com/products/workstation-pro.html`

[3] AILab was founded to support students major in computer science to do researches on artificial intelligence and machine learning. Because some of the machine learning programs require expensive computations, the AILab is equipped with workstations with high-performance GPUs

3.1.1 Docker Environment and NVIDIA Driver

Docker [4] is a tool designed to make it easier to create, deploy, and run applications by using containers. Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and deploy it as one package. Because Baidu Apollo is dependent on numerous libraries and datasets, it is encapsulated in a Docker image by developers to accelerate the building speed and avoid compilation errors. After installing Docker in the machine, the NVIDIA Driver and [5] the NVIDIA Container Toolkit [6] are installed as well.

3.1.2 Building Apollo on Ubuntu 18.04

After the Docker environment and NVIDIA Driver are configured appropriately, Baidu Apollo can be downloaded from its GitHub repository to the local machine. The code cloning process lasts for considerable amount of time as the repository is over four GB in size, and the data for training object detection algorithms occupy most of the space. After the whole code base is downloaded, the branch should be *checkout* to r3.0.0 to use the version 3.0 of the software. Then *bashdocker/scripts/dev_start.sh* is executed to pull the Baidu Apollo image from web, and *bashdocker/scripts/dev_into.sh* is executed to enter the docker environment afterwards. Two options can be chosen to build Apollo, CPU version and GPU version. Because GPU has better capacity of parallel computing, the performance of running the simulation is better when building a GPU version as long as the NVIDIA driver and NVIDIA docker are correctly installed and configured. The CPU version is built by running *bashapollo.shbuild*, while the GPU version is by running *bashapollo.shbuild_opt_gpu*. Then Dreamview GUI can be started by executing *bashscripts/bootstrap.sh* and accessing localhost:8888 in the web browser which is shown in figure 3.1.

[4]Docker Official Website. Avaliable at: `https://www.docker.com/`

[5]NVIDIA Driver Installation Documentation. Available at: `https://tech.amikelive.com/node-731/how-to-properly-install-nvidia-graphics-driver-on-ubuntu-16-04/`

[6]NVIDIA Container Donwload Website. Available at: `https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/install-guide.html#docker`

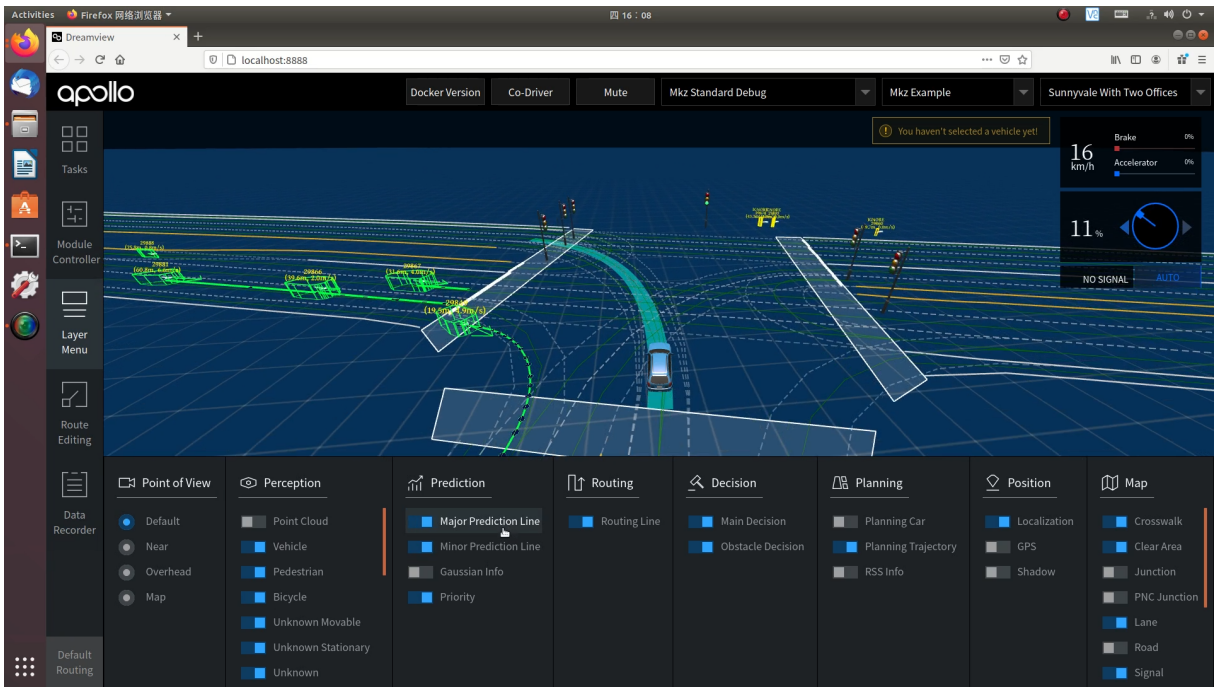

Figure 3.1: Dreamview GUI

The car model and map can be selected from the top right of the interface, and *SimControl* button and *Setup* button should be clicked to enable the core functional modules including Perception, Prediction, Planning and Routing. Finally, a route can be chosen in the *DefaultRouting* menu at the bottom left of figure 3.1, and then the driving simulation of the vehicle can be shown in the GUI and the corresponding information of the vehicle including speed and steer angle is visible at the top right region of the GUI.

3.2 Testing Methodology and Experimental Design

3.2.1 MFT Process

The testing approach of this project is Metamorphic Fuzz Testing (MFT), which combines two stages, the fuzz testing (FT) stage and the metamorphic testing (MT) stage (see in 3.2.2). FT is adopted to generate randomly positioned obstacles and published to autonomous driving simulation. The vehicle departs from the start point while a set of obstacles is generated using fuzz function, and this set of obstacles is regarded as a scenario. After a fixed period of time, a new scenario is generated with a different set of obstacles in new positions. A round of test contains certain number of scenarios, and if the vehicle collides with obstacles in the simulation, it will be labelled as *Candidate Failure*. The candidate failure is classified into two categories, the *Genuine Failure* and *Unavoidable Failure (False Positive)* [11], as it is not convinced whether the failure is caused by the bugs in the system, or by any inevitable reasons. Due to the randomness of the fuzz function, the obstacles generated are likely to close to the vehicle. In this case, the collision may be unavoidable if there is no enough time for the vehicle to decelerate to zero in a short distance. Therefore, this collision is a false positive case which is not

the fault of the system. Otherwise, if the object detection algorithm cannot recognise the obstacles, the collision can be incurred, which is found to be avoidable, and this belongs to the genuine failure. During the FT stage, all the collision cases are labelled as the candidate failure. In the MT stage, the candidate failure occurred in FT stage is distinguished into genuine failure and false positives, and the relation is displayed in Figure 3.2

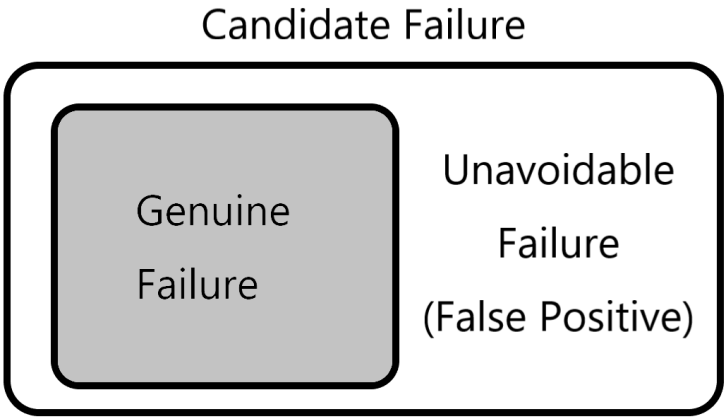

Figure 3.2: Candidate Failure Component

To find the bugs or vulnerabilities of the software, only the genuine failure is concerned rather than the unavoidable failure. Therefore, MT is adopted to differentiate genuine failure from unavoidable failure. For every candidate failure detected from the FT stage, a follow-up test case is produced base on the source test case and the MR. Then, the output of the source and follow-up test cases are compared according to the MR. Only if the source test case is genuine failure, the violation of MR will occur. Consequently, the false positives can be discarded and only target genuine failure is extracted from the candidate failure.

3.2.2 Experimental Design

The total number of testing rounds is set as 300 because only when the test set is adequate can the general conclusions be obtained from the experiment. In the FT stage, the run time for a single scenario is five seconds and thus the overall run time for 300 rounds is 150 minutes. In all rounds of FT, the car, map, traffic condition, starting and ending points are the same, but the scenarios which refers to the set of obstacles are different. In each round of test, a scenario containing hundreds of randomly generated obstacles will be created and published to the simulation scene. The scenario is refreshed to a new set of obstacles after five seconds. The only thing that changes is the positions of the obstacles.

In the MT stage, MR is adopted to identify genuine failure from the set of candidate failure detected in the fuzzing stage. For each collision recorded in the fuzzing stage, a follow-up scenario is created and run against MR. The MR used in this testing is the same as the one used in the MFT research on Baidu Apollo [11]:

- *MR1:* Suppose that in a driving scenario, S, a car collided with a static obstacle

O at location L. Construct a follow-up driving scenario, S', that is identical to S except that O is repositioned slightly further away from the previous position. Run the follow-up driving scenario. The car should not stop before L.

According to the MR1, the follow-up test case should be identical to the source test case, except that the position of the obstacle collided by the vehicle is moved away from the vehicle's direction for a certain distance. Genuine failure is determined by MR violations and the failure rates can be calculated. The whole testing process will be automatically executed.

3.3 Automatic Test Script Implementation

The test involves two stages. The first one is to use fuzz function to generate random obstacles in the scene, and the second one is using MT to differentiate the genuine failure from candidate failure. The version of Baidu Apollo was tested is 3.0. The test scripts was coded in Python and Shell scripts.

3.3.1 Automatic Test Script Architecture

The code is forked from the Baidu Apollo's repository to the individual GitHub account and a folder $auto - test$ is added to store the test scripts. The code is available at this GitHub repository: `https://github.com/NikoHhEe/apollo/tree/r3.0.0`. The structure of the test scripts is shown in Figure 3.3.

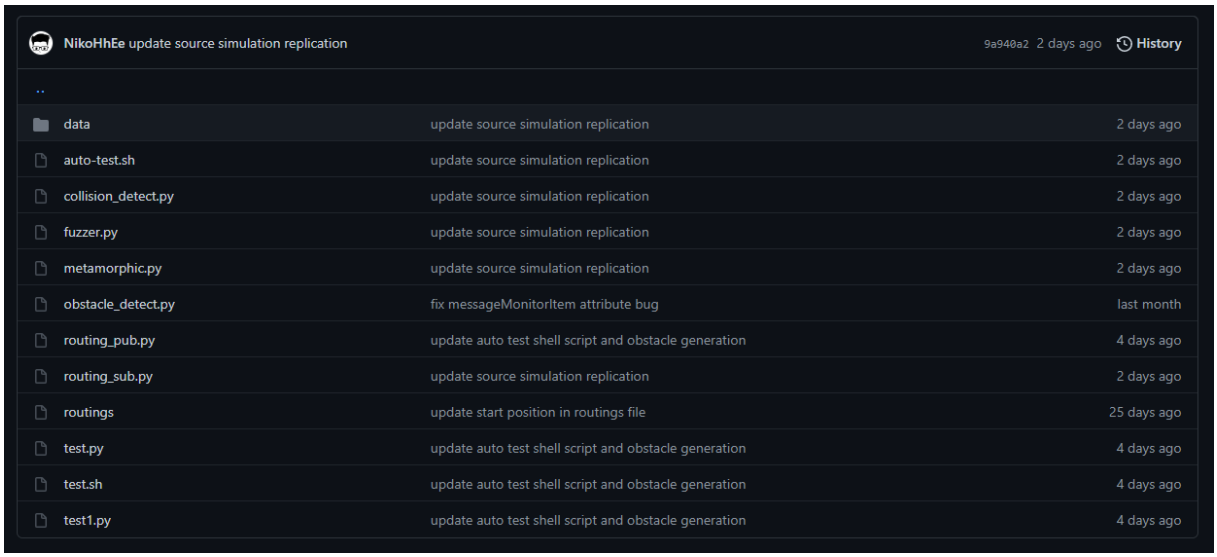

Figure 3.3: Test Script Structure

In the $auto - test$ folder, there is a *data* folder containing some csv files that store data used during the test. *routings.csv* contains the information of the different routes, including the coordinates of the start and end points of several routes. The main test

scripts can read this file and choose different routes for the vehicle to travel. The rest files in $auto-test$ folder are the implementation of the test, and the scripts are written in Python and Shell script. $fuzzer.py$ generally implements the fuzz function for the obstacle generation as well as records the information of the obstacles generated. $collision\_detect.py$ monitors the driving simulation and records the information of the obstacles that collide with the vehicle. $metamorphic.py$ restores the previous driving simulation and modifies it according to the MR and calculates the genuine failure rate of the simulation. $routing\_sub.py$ subscribes the routing information that is published to Apollo system, and $routing\_pub.py$ publishes user-defined routing requests to Apollo system to drive the vehicle in certain routings. These Python scripts implement different functions respectively and $auto-test.sh$ integrates the separated scripts and achieves the test automation.

3.3.2 ROS-related Concepts and rospy Usage

The implementation of the test scripts requires ROS [25] and the corresponding Python client library *rospy*. Before explaining the test scripts, certain background information of ROS is introduced.

ROS [25] is an open-source, meta-operating system for robots. It provides the operating system services, including hardware abstraction, low-level device control, implementation of commonly used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers. As ROS implements several features required by most robots, it can be used as a robot framework. The primary goal of ROS [25] is to support code reuse in robotics research and development. ROS is a distributed framework of processes as known as $Nodes$ and they can be easily shared and distributed. ROS mainly runs on Unix-based platforms now, and this is why Baidu Apollo is designed to be running on Ubuntu OS.

A computation graph [25] is a peer-to-peer network of ROS processes that work together to process data in the ROS system. The basic computation graph concepts of ROS includes Nodes, Masters, Parameter Servers, Messages, Services Services, Topics, and Bags, all of which perform data communications in different ways.

- Nodes: A node is an executable that uses ROS to communicate with other nodes. In Apollo, each individual module is a ROS node. For exampe, the perception module and the planning module.
- Messages: A message is a ROS data type used for subscribing or publishing information to a topic.
- Topics: Nodes can publish messages to a topic as well as subscribe to a topic to receive messages.

A node is an executable file within a ROS package. ROS nodes use a ROS client library to communicate with other nodes. Nodes can publish or subscribe messages to a $Topic$, and there are two ROS client libraries that can be used to interact with the data in ROS:

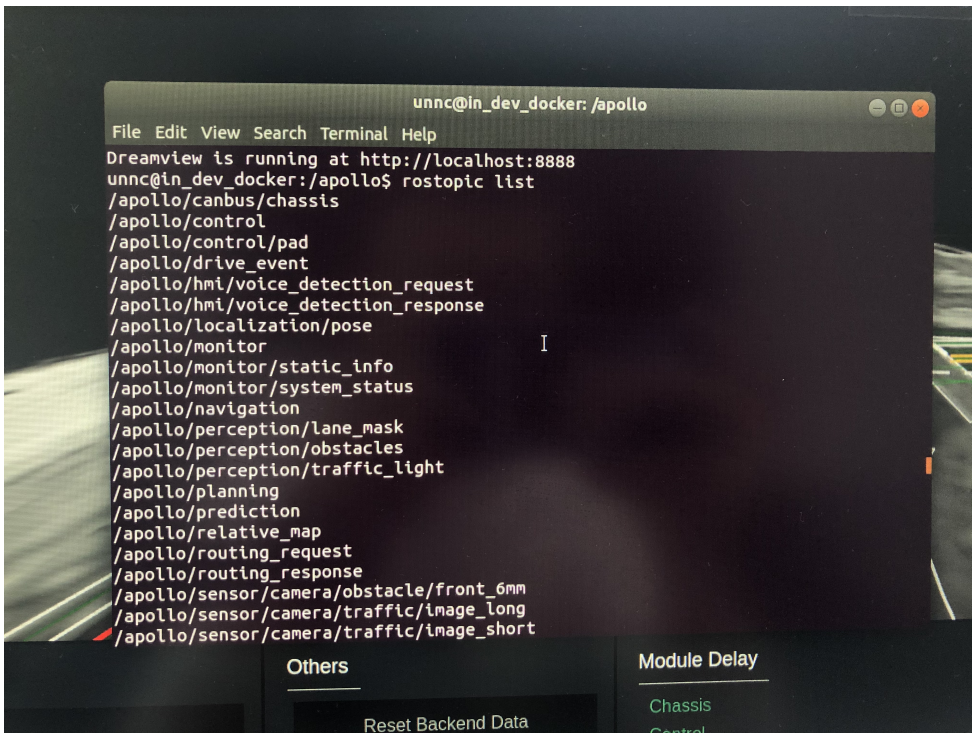

Figure 3.4: Executing *rostopic list*

- *rospy*: python client library.
- *roscpp*: C++ client library.

Node communicates with each other using a ROS Topic, and the several Topic-related commands are shown:

- *rostopic list*: returns a list of all topics currently subscribed to and published.
- *rostopic echo [topic]*: shows the data (messages) published on a topic.

Communication on topics happens by sending ROS messages between nodes. *msg* stands for messages in ROS, and *msg* files are simple text files that describe the fields of a ROS message. They are used to create source code for messages in different languages (Python or C++). The following are several message-related commands:

- *rostopic type [topic]*: returns the message type of any topic being published.
- *rosmsg show [message]*: shows the details of the message.
- *rostopic pub*: publishes data on to a topic currently advertised.

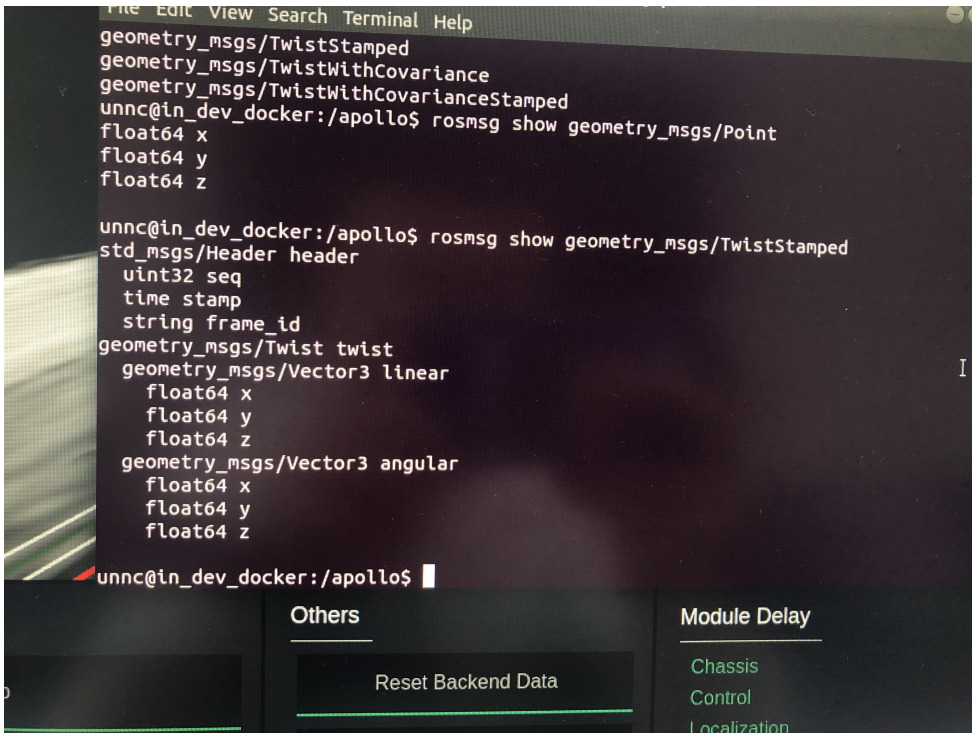

Figure 3.5: Executing *rosmsg show [message]*

3.3.3 Stage One: Fuzzing

Fuzzing is a concept or pattern of generating random inputs, so there is no fixed code and the implementation of fuzzing is dependent on the intention and the aim to achieve. In the scenario of the generating obstacles, the blackbox fuzzing is used which generates obstacles in the scene absolutely randomly without considering the code coverage. The purpose of the fuzzing in this test is to generate extreme scenarios and an obstacle, for example, immediately appears in front of the vehicle with a high speed. Therefore, the randomness satisfies the intention of the fuzzing and there is no need to consider the code coverage.

Fuzz Function: fuzzer.py

The primary script that implements the fuzzing is *fuzzer.py*. Except for some common Python libraries, *numpy*, particular Apollo-related libraries, can also be imported in the test script, including *rospy* and *PerceptionObstacles*. *rospy* provides the functions to interact with the ROS topics and ROS messages in Baidu Apollo, and the information of the obstacles and collisions can be extracted. *PerceptionObstacles* provides the interface defined in perception module of Apollo code [7], and artificial obstacles can be generated and published to ROS topics using the functions in this interface.

The main function of *fuzzer.py* is shown in Listing 3.2, and it simply calls *talker*() and

[7]Baidu Apollo Perception Module. Available at: `https://github.com/NikoHhEe/apollo/tree/r3.0.0/modules/perception`

throws *ROSInterruptException* if exceptions of this kind are detected.

```
if __name__ == '__main__':
    try:
        talker()

    except rospy.ROSInterruptException:
```

Listing 3.1: *fuzzer.py* main function

Then the code structure of *talker*() is explained, and the reason why it is named *talker* is that the function publishes messages containing obstacle information to the ROS topic */apollo/perception/obstacles*, and it is a role of a talker that sends information out.

```
def talker():
    pub_obstacle = rospy.Publisher('/apollo/perception/obstacles',
    PerceptionObstacles, queue_size=10)
    pub_routing = rospy.Publisher('/apollo/routing_request', routing_pb2
    .RoutingRequest, queue_size=10)

    rospy.init_node('talker', anonymous=True)
```

Listing 3.2: *fuzzer.py* publisher initialisation

First, a ROS publisher for the obstacles is initialised and the target ROS topic */apollo /perception/obstacles* is specified. Then, another ROS publisher for the routing request is initialised and the corresponding ROS topic is */apollo/routing*$_r$*equest*. The node is named as *talker* and the refresh rate is defined to be 0.2Hz, meaning that the obstacles are regenerated every five seconds and each scenario lasts for five seconds as well.

Next, the routing request is generated from the data in *waypoints.csv* which stores the position information of the waypoints on the routing. After essential information is assigned, the request is published to ROS topic */apollo/routing*$_r$*equest*. This function simulates the routing selecting by manually clicking action in Dreamview, and the manual action now can be executed by scripts.

```
    routing_data = genfromtxt('/apollo/auto-test/data/waypoints.csv',
    delimiter=',',dtype=None)
    len_waypoints = len(routing_data)

    for point in range(len_waypoints):
        curr_point = routing_data[point]
        msg = msg_routing_request.waypoint.add()
        msg.id = curr_point[0]
        msg.s = curr_point[1]
        msg.pose.x = curr_point[2]
        msg.pose.y = curr_point[3]

    # print(msg_routing_request)
    # wait for 2 seconds to let the message published successfully
    # if time is too short, the message may be ommited by the system
    time.sleep(2.0)
    pub_routing.publish(msg_routing_request)
```

Listing 3.3: *fuzzer.py* routing request publisher

Then the route data is read from */apollo/auto−test/routings.csv* and a specific route is selected by the ID index, and in this case, route five is chosen. The coordinates of start and end points of the route are then extracted, which are used to define the region where obstacles are generated.

```
    # read data from routings.csv
    routings = genfromtxt('/apollo/auto-test/data/routings.csv', dtype=
    int, delimiter=',')
    # select the routing by id
    routing_id = 5 # 'Sv sunrise loop'
    # routing = routings[routings[:,0] == routing_id]
    routing = routings[routings[:,0] == routing_id][0]
    # extract the start and end coordinates from the list
    start_x = routing[1.1]
    start_y = routing[2]
    end_x = routing[3]
    end_y = routing[4]
```

Listing 3.4: *fuzzer.py* route initialisation

The map provided by Baidu Apollo 3.0 is Sunnyvale big loop, which is taken from the Silicon Valley of California. All the routes provided in Sunnyvale big loop are shown in Figure 3.6, and the major routes are labelled with colored lines.

Figure 3.6: Sunnyvale Map

To simplify the test, routes used for testing are all straight, because it makes obstacle generation easier and the driving tracks are basically straight. Based on the start and

end position of the route, a rectangular region can be defined for each route, as shown in Figure 3.7. This region area covers all possible locations for vehicles to reach. Because the route is the diagonal of the rectangular region, all obstacles will be generated in this region. A drawback of this obstacle generation approach is that majority of the region is wasted, for the region area is much larger than that of the road where the vehicle can appear. Consequently, obstacles generated in the rest of the region are meaningless as there is no chances for them to collide with the vehicle. When the number of obstacles in the scene grows up, the simulation is stuck because of the huge amount of obstacles to render, and this is optimised by rotating and compressing the generation region.

```
    # define the boundary of region where obstacles are generated
    # using the start and end points in this case
    bound_left = min(end_x, start_x)
    bound_right = max(end_x, start_x)
    bound_up = max(end_y, start_y)
    bound_down = min(end_y, start_y)

    center_x = 0.5 * (start_x + end_x)
    center_y = 0.5 * (start_y + end_y)

    # get the tan value of angle of the road according to the horizontal
    line
    tan_value = abs(float(end_y-start_y) / float(end_x-start_x))
```

Listing 3.5: *fuzzer.py* obstacle generation region initialisation

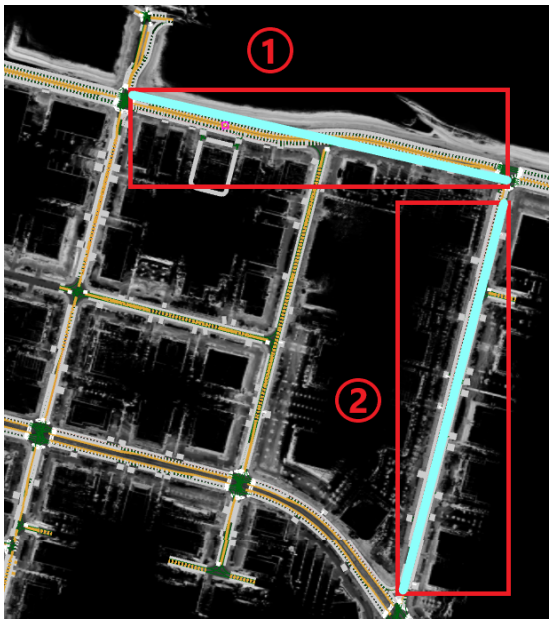

Figure 3.7: Obstacle Generation Region

Figure 3.8: Obstacle Generation Region Optimised

The way to optimise the region is to firstly rotate the region on the axis of the road centre to make the rectangular area along the same direction as the road, as shown in Figure 3.8. Then the width of the region is compressed to decrease the area that is beyond the edge of

the road. The rotation of the region is achieved by transformation matrix [8]. For a point (x, y), after it is rotated by an angle θ clockwise about the origin, its new coordinates (x', y') are represented as $x' = x\cos\theta + y\sin\theta$ and $y' = -x\sin\theta + y\cos\theta$. Written in matrix form, and this becomes:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Figure 3.9: Rotation Matrix

It is worth noting that the terminology *Transformation* represents all types of movements, including *Translation*, *Rotation* and *Scaling*. Consequently, translation and rotation are two forms of transformation. Before applying the transformation, the general direction of the road should be determined. If the route is generally horizontal, for instance, route 1 in Figure 3.7, the region be rotated clockwise at the angle between the road and the horizontal. Another scenario is where the general direction of the road is vertical, for example, road 2 in Figure 3.7, then it should be rotated clockwise at the angle between the road and the vertical. After the road direction is determined, the rotation angle is calculated using trigonometry. If the routing is generally horizontal, then the region should be compressed vertically and *compress* is marked as 1. Otherwise, the compression should be horizontal and *compress* is marked as 0.

```
    # determine whether the region is general horizontal or vertical
    if abs(start_x-end_x) > abs(start_y-end_y):
      compress = 1
      angle = arctan(tan_value)
        # negate the angle if counter-clockwise
        if (start_x-end_x)*(start_y-end_y) < 0:
            angle = -angle
    else:
      compress = 0
        angle = arctan(1/tan_value)
        # negate the angle if counter-clockwise
        if (start_x-end_x)*(start_y-end_y) > 0:
            angle = -angle
```

Listing 3.6: *fuzzer.py* region transformation

In practice, region rotation can be achieved by applying the transformation matrix to all the points in the region, and the points after rotation will be generated in the new region, which looks like the region is being rotated. A function is defined to apply transformations to the points. The inputs are the x and y coordinates of a point to be transformed and the rotation center point, and a flag *compress_x_y* indicating the direction of region compression. Inside the function, the point is first translated to the origin point before applying the rotation. Because rotation is at the centre of the origin, and if the rotation is performed at other positions, the point will not be rotated to the desired location. The point is translated back to the original point after rotation to reverse the translation effect

[8]Transformation Matrix introduction. Available at: `https://www.khanacademy.org/math/linear-algebra/matrix-transformations`

in the beginning. Eventually, the coordinates of the transformed point are obtained and output.

```
# calculate the tansformed coordinates of x and y
# according to the angle and distance between center to origin
def transform(x, y, angle, x_center, y_center, compress_x_y):
    # first translate to the origin
    x = x - x_center
    y = y - y_center

    # compress the width of the region
    if (compress_x_y == 0):
        x = 0.2 * x
    if (compress_x_y == 1):
        y = 0.2 * y

    # apply rotation
    x = x * cos(angle) - y * sin(angle)
    y = x * sin(angle) + y * cos(angle)

    # translate back to the previous position
    x_transform = x + x_center
    y_transform = y + y_center

    return x_transform, y_transform
```

Listing 3.7: *fuzzer.py* transformation matrix implementation

Then the area of the region is computed, and the density of obstacles is defined. Based on these two parameters, the number of obstacles to be generated in one scenario is then calculated. A prefix can be added to obstacle ID to make the obstacle ID unique among all scenarios.

```
    # calculate the area of the region
    area_region = (bound_right - bound_left) * (bound_up - bound_down)
    # define the obstacle density (0 - 1)
    obstacle_density = 0.003

    # define number of obstacles
    n_obstacles = int(obstacle_density * area_region)
```

Listing 3.8: *fuzzer.py* area and density

Next, a *for loop* is started and the obstacles are generated and published to ROS topic */apollo/perception/obstacles* until the scenario ID exceeds the limit. A new object *msg_obstacles* is created, and it is a list of obstacles where new obstacles can be inserted. The inner *for loop* generates specific number of obstacles for one single scenario, and the obstacle amount is computed before the loop. For each individual scenario, a fixed amount of obstacles are generated and added to the obstacle list, and the attributes of the obstacle are defined respectively. The obstacle ID is defined by adding the loop index and the prefix to ensure that every obstacle ID is unique. The x and y coordinates are randomly generated within the region range and then applied to the transformation function to realise rotation. The z coordinate is 0 by default. The dimensions of the obstacle are randomly selected from one meter to four meters, and the heading direction is fixed to 1.0 in radius. The type of the obstacle is randomly defined, and possible choices include pedestrian, bicycle, vehicle and other types. The key information of obstacles

should be stored in *obstacle.csv* after generation. At the end of the inner *for loop*, the obstacle list containing all the obstacles in one scenario is published to the ROS topic */apollo/perception/obstacles* in the message format. After that, the simulation lasts for certain period and then the next scenario will start and a new list of obstacles will be generated (see code in Appendix A).

Collision Detection: collision-detect.py

collision − detect.py monitors the simulation and records any collision for further analysis. A new library *MonitorMessage* is imported which provides the interface of the simulation monitoring messages, including the collision obstacle ID and other information. The main function directly calls *listener*() which subscribes messages from ROS topic */apollo/monitor* and extracts information of the collisions. *rospy.Subscriber*() calls a function *monitorCallback* which is the primary implementation of collision detection.

```
def listener():
    rospy.init_node('collision_detect', anonymous=True)
    rospy.Subscriber('/apollo/monitor', MonitorMessage, monitorCallback)
    rospy.spin()

if __name__ == '__main__':
    listener()
```

Listing 3.9: *collision − detect.py* main function

monitorCallback() has a input variable *monitorMessage* which contains certain meta information of the messages in the monitor, including the header information of the monitor and a list of monitor messages. The message list is defined as the *item* attribute of *monitorMessage* object and the first message in the list can be obtained by indexing the item list by zero. The message list is a collection of message items which is similar to *PerceptionObstacles* in *fuzzer.py*. The monitor message item contains certain meta information of the message, including message source and the message string. The string is the *msg* attribute of *monitorMessageItem*, and it is the target information to be obtained. After extracting the string of the message, the content can be analysed. If there is collision detected by Apollo, the message string will be *Found collision with obstacle: obstacle_id*. Then, by checking whether there is a substring *Found collision with obstacle:* in the message, the obstacle that collided with the vehicle can be found and its ID can be saved to file *collision.csv*.

```
def monitorCallback(monitorMessage):
    # get the monitor message item
    monitorMessageItem = monitorMessage.item[0]
    # extract message from the item
    msg = monitorMessageItem.msg
    # expected message when collision is detected
    collisionMessage = 'Found collision with obstacle: '

    # check if the message contains collision information
    if (collisionMessage in msg):
        # extract the obstacle id from the message
        # need to consider when id has '_0' suffixes which cannot be
   convert to int directly
```

```
        if ('_' in msg):
            # find the index of of '_' in the msg
            position = msg.find('_')
            # drop the chars include and after '_' to get a proper int
            msg = msg[:position]
        obstacle_id = int(msg.replace(collisionMessage, ''))
        obstacle = [obstacle_id]
        #print(obstacle_id)

        # add new obstacle data into the csv file
        with open('/apollo/auto-test/data/collision.csv', 'a') as csv:
            np.savetxt(csv, obstacle, fmt='%d', delimiter=',')

        # need to remove redundant ids
```

Listing 3.10: *collision − detect.py* monitorCallback

One potential problem of this script is that *collision.csv* stores a single obstacle multiple times because of the monitor information refreshes frequently. When collision takes place, the vehicle will stop and the state of collision remains for certain period. Hence, the monitor will continue outputting the same collision message until the obstacle is removed and new ones are generated at different locations. Several approaches are attempted to fix this problem, including changing the refreshing rate of the monitor. Nonetheless, the problem cannot be perfectly solved because changing refresh rate will lose part of the collision information. A temporary solution to this issue is to keep the redundancy of the obstacle ID in *collision.csv* and remove the repetition in the MT stage of the experiment, which is in script *metamorphic.py*. In the future, the data redundancy should be resolved at the root cause, which means the data saving part should guarantee the uniqueness of the obstacle id.

3.3.4 Stage Two: MT

After running *fuzzer.py* and *collosion − detect.py*, there are two files stored in folder *data*. The first one is *obstacles.csv*, consisting of the the information of all the obstacles generated in every scenario of one test. The second one is *collision.csv* which stores all the obstacle ID that collided with the vehicle in the simulation. The information in this two files can be used to reproduce the source test scenario and the replicated scenario is then modified to be follow-up test cases with respect to MR.

MR Identification

In this testing project only one MR is used, and is taken from the MFT research carried out by Jiacheng Han and Dr. Zhiquan Zhou [11].

- *MR1:* Suppose that in a driving scenario, S, a car collided with a static obstacle O at location L. Construct a follow-up driving scenario, S', that is identical to S except that O is repositioned slightly further away from the previous position. Run the follow-up driving scenario. The car should not stop before L

The motivation of identifying this MR is from intuition, because if the ego car cannot stop before location L and hence a collision with O at L in scenario S is unavoidable, then it is impossible to stop before L. Otherwise, if the car could stop before L in scenario S', then it should have been able to stop before L in scenario S in the first place. This would mean that the collision in scenario S is not unavoidable, in other words, the collision is a genuine failure of the SUT. $MR1$ gives a sufficient but not necessary condition for the identification of genuine failure.

Follow-up Test Case Generation: metamorphic.py

Generating follow-up test cases involves two steps. Firstly, the same set of obstacles appeared in the source test case should be reproduced in the follow-up test case according to the positions and scenario ID in the source test case, except for those modified regarding the MR. Secondly, it should be ensured that the vehicle driving situation is absolutely identical to the original one. To reconstruct the obstacles, the data of obstacles should be read from *obstacles.csv*, including the obstacle id, scenario id, obstacle position and dimensions as well as the type of the obstacle. *collision.csv* is then read and the ID of obstacles that collided with the vehicle can be obtained. With the information of the complete obstacle set, the obstacles can be regenerated at their original positions. For the obstacles collided with the vehicle, their new position is calculated by modifying the original position according to MR. Specifically, $MR1$ generates a follow-up scenario that the position of the obstacle is moved away for certain a distance from the original position in the source scenario.

To restore the vehicle driving state, the only applicable approach is to let the vehicle start moving from the starting point while the obstacles in the first scenario are generated. In this case, the timeline of the vehicle driving can correspond to the timeline of obstacle generation, which means the states of the vehicle when approaching the obstacle, will be identical in both source and follow-up test cases. Therefore, it is extremely important to ensure that the time slot between the vehicle starts and obstacle generation is exactly the same in the source and follow-up test case generations. Otherwise, a time difference of less than one second can lead to a huge difference in the simulations.

Full code of *metamorphic.py* is available in Appendix A. Originally, the vehicle was started by clicking the routing choices in the $DefaultRouting$ menu, but it is not practical to click it every time a new test scenario is generated. In order to choose a routing and start the vehicle automatically, a function was designed to generate the routing request messages and publish them to the ROS topic "/apollo/routing_request". Before publishing a routing request, certain information about the routing should be obtained including the coordinates of the routing waypoints, which refer to the $Nodes$ that the vehicle pass through. In this test, the routing selected is the road 2 in Figure 3.7. A listener script *routing_sub* was written to extract the information of the waypoints from the ROS messages subscribed from the ROS topic "/apollo/routing_request". The script will receive the messages after the routing choice is clicked in Dreamview, then the information is manually stored in file *waypoints.csv*. When publishing the routing request in $routing_pub.py$, file *waypoints.csv* is read and its data is extracted to generate new routing requests, and the request is then published to ROS topic "/apollo/routing_request". After that, the vehicle will start

and navigate through the desired waypoints of the routing. Later, considering that the time synchronization issue in the source and follow-up cases, the routing publish function was embedded in both *fuzzer.py* and *metamorphic.py*. Previously, this is an individual script encoded in *routing_pub.py*, and called in the main test script $auto - test.sh$ before the *fuzzere.py* or *metamorphic.py* are executed. Nevertheless, this approach will result in different time slots between the routing publishing and obstacle generation in source and follow-up test cases, because the code for obstacle generation in *fuzzer.py* and *metamorphic.py* is located at different lines of the corresponding scripts, and the CPU time taken to arrive at the obstacle generation code is slightly different. Therefore, to keep the synchronization, the routing publishing code is inserted right before the obstacle generation code in both *fuzzer.py* and *metamorphic.py*.

Theoretically, the method should generate identical simulation corresponding to the source simulation scenario, and can then be modified to be follow-up simulation. To examine whether the replication of the source simulation is completely identical, the replicated scenario is ran and the collisions are checked. If all the collisions in the source simulation can be reproduced in the replicated scenario, it demonstrates that the replicated simulation is identical to the source one. Numerous simulation pairs are carried out, and the collision results of them are compared respectively. According to the experiment results, the probability of completely replication is 55%, which means the script can not completely replicate the all the simulations in the experiment. This is a serious problem for implementing MT, because the follow-up case generation is based on the replication of source test case. If the source simulation cannot be replicated, then the follow-up case which is modified with MR cannot be generated validly. The experiment cannot continue without solving this problem, and thus a considerable amount of effort went into solving this problem, including inserting $routing_pub.py$ into *fuzzer.py* and *metamorphic.py* to alleviate the time synchronization problems, and checking the code logic in *fuzzer.py* and *metamorphic.py* to ensure the time between routing message publishing and obstacle generation remains the same. Unfortunately, after several attempts, the problem still remains. Therefore, the final objective of the experiment was not accomplished which is calculating the genuine failure rate of Baidu Apollo system using MR1.

Chapter 4

Results and Analysis

This chapter presents the testing results of the experiment. Observations through the experiment are discussed and the difficulties in the experiment are analysed.

4.1 Testing Results

As mentioned in Chapter 3, the experiment was not successfully carried out because the replication of the source test case was unsuccessful. Consequently, the follow-up simulations cannot be generated and the MT part of the experiment cannot be implemented. One of the primary aims of the experiment is computing the genuine failure rate of Baidu Apollo AV system, but the value cannot be calculated due to the lack of valid experimental data. Although the most desired results are not obtained, there are still certain aspects which can be discussed and analysed.

4.2 Fuzz Function Evaluation

The fuzz function is implemented to generate effective test cases to the simulation environment in Apollo system, and it is one of the most important functionalities in the simulation experiment, because it influence the fluency of the collisions and the vehicle driving states.

Specifically, the first key parameter of the fuzz function is the refresh frequency of the obstacles. The final choice of the refresh rate is five seconds, meaning that a scenario of obstacles is cleared every five seconds and then a new set of obstacles for a new scenario is generated. If the refresh rate is too large, for example, less than one second, the obstacles will be reconstructed too frequently. The drawback is that the vehicle will not be able to drive at a normal speed, because the obstacles are more likely to collide with the vehicle when they are reconstructed too frequently. In this case, the simulation cannot reflect the best performance of the vehicle, because this kind of situation is unlikely to happen in real life. Although FT aims at testing the robustness of the system, fuzz function with high frequency can not carry out appropriate results. On the other side, the refresh frequency can not be too small. If one scenario lasts for too long, the time will be wasted during the testing. In each scenario of the testing, there are generally two outcomes of the vehicle, the vehicle either collide with the obstacle or stop before hitting it. In both cases, the vehicle is eventually static and waits for the next scenario to be generated and continues its driving. If the scenario duration is longer, the time between the vehicle remaining

stationary and the vehicle moving on is longer as well. This period is meaningless and should be shortened to an applicable range. After adequate tests on different refreshing rates, five seconds per scenario was chosen as the most appropriate value, because it both guarantees a relatively normal driving simulation for the vehicle, and decreases the time spent waiting for the next scenario.

Another factor of the fuzz function that affects the simulation is the density of the obstacles generated in each scenario. Density in the fuzz function represents the amount of obstacles in unit area, which is one square meter. If the density is too small, the amount of obstacles generated is small, and the probability of collision between the vehicle and obstacles is low as well. Then the expected results cannot be obtained because the experiment depends on the data generated by collisions. If there are not enough collision data, the analysis of the results will be unconvincing. Nevertheless, higher density is not applicable either, because if the obstacles are fully-filled in the map, the vehicle will not be able to drive normally in the simulation. This is similar to the problem of high refresh frequency. To choose an appropriate density, sufficient tests are conducted to find a suitable density value.

Another feature of the fuzz function worth discussing is that the obstacle generation region is optimised using transformation matrix. As discussed in Section 3.3.3: Fuzz Function, the region of obstacle generation was rotated and compressed to fit into the selected routing used for testing. The intention of this optimisation is to avoid useless obstacles that are generated beyond the road area, and this can also reduce the work load of storing obstacle information. When the amount of obstacles is larger, the time spent on obstacle information storing will increase and thus has an influence on the time synchronisation between the source and follow-up test case. Because there is no data saving process in *metamorphic.py*, if $fuzzer.py$ takes noticeable time to store obstacle data, the simulation time between the two cases will vary significantly, and eventually results in the difference of source simulation and the replicated one.

4.3 Reasons for Scenario Replication Failure

As mentioned in Section 3.3.4: Follow-up Test Case Generation, the failure of replicating the simulation after running the source simulation is caused by the time synchronization issue among the source and follow-up test case generations. Theoretically, the method to replicate a driving simulation in Dreamview is to generate the same set of obstacles and start the vehicle at the same time. To achieve this, the time slot between sending the routing request and starting the vehicle should be constant in both source and follow-up tests. Although the routing publishing function is placed right before the obstacle generation, the time between the two operations is different in the two simulations in practice. Several round of experiments are conducted to find out the reason why replicating the previous simulations, and the experiment is based on the current code. As mentioned in Section 3.3.4, there are 55% of the replications that are identical to the their source simulation, for these successful replicated cases, an additional replication based on the source test case was made respectively. A disappointing result shows that not all the successful replicated simulations can have a third complete replication, and only 33% of them can replicate the original simulation three times. This indicates the simulation is

highly sensitive to time factor, even if a small difference of time will result in completely different driving scenarios. Many operations of the simulation influences the time, including the obstacle data saving and reading. Additionally, the time between different scenarios is also uncertain. Although each scenario is defined to last for five seconds, there are noticeable CPU time spend on executing other lines of code. Therefore, the current test script cannot guarantee the time consistency among different test cases, and thus is incapable to replicate source simulations or generate follow-up test cases.

4.4 Problem Found in the System

Although the experiment was not completely carried out, some observations on the performance of Baidu Apollo was made during numerous rounds of simulations. Firstly, the collision only happens when there are obstacles generated close to the vehicle. Otherwise, if the obstacle is far away from the vehicle, there is unlikely to be a collision. Sometimes, the vehicle is at a low speed which is under 10 km/h, and it will hit an obstacle generated close to it. Thus, the speed of the vehicle does not affect whether the vehicle can avoid the collision with a nearby obstacle. From a long time observation, a common feature of the collisions was found, which is the stop decision wall is generated inside the vehicle. The situation is shown in Figure 4.1.

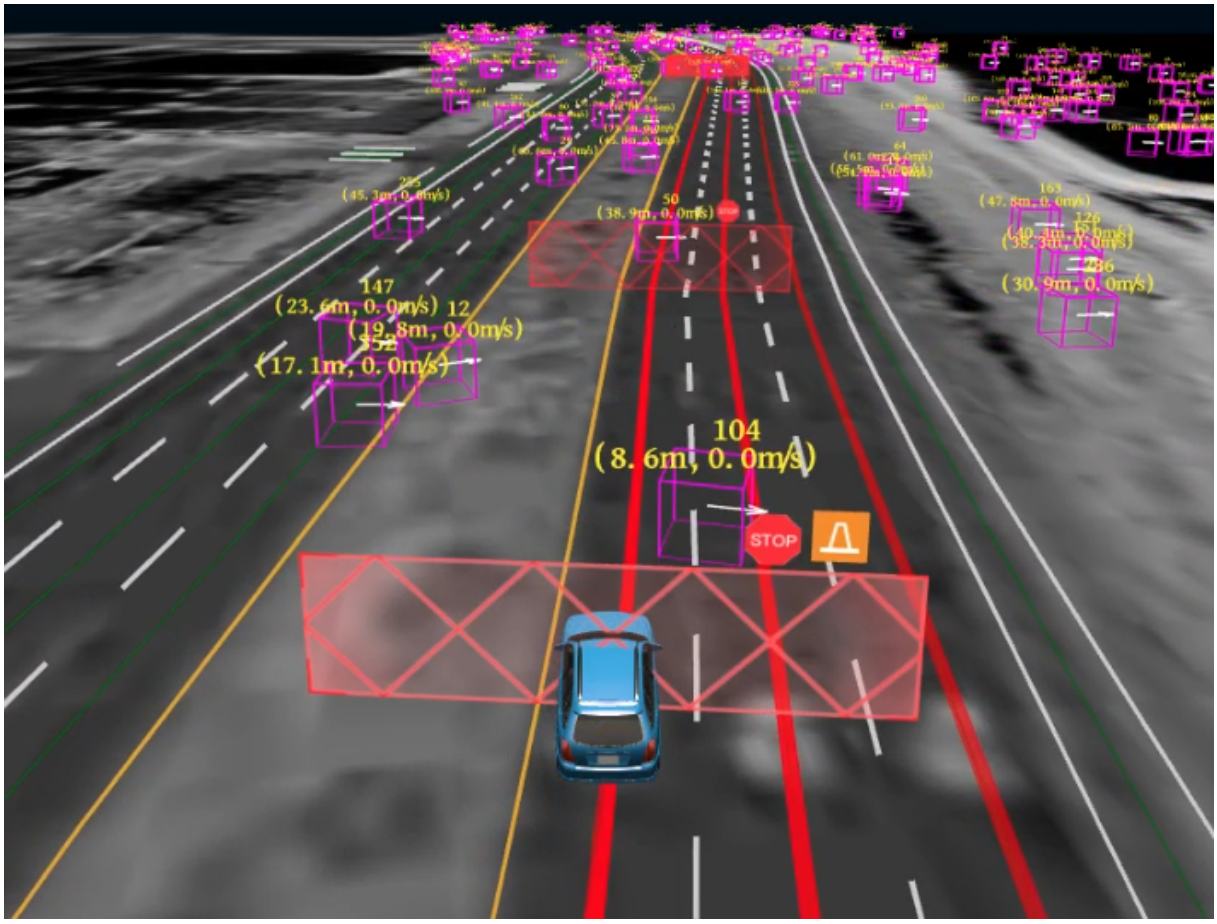

Figure 4.1: Stop Decision Fence

The stop decision fence is generated in front of an obstacle when the vehicle is approaching it. The fence informs the vehicle to stop at that point which is safe and will not collide with obstacles. However, when the obstacle is immediately generated near the vehicle, the fence will be created inside the vehicle. In this case, the vehicle may lose control and keep driving until it hits the obstacle, even if the speed is low.

Chapter 5

Summary and Reflections

This chapter includes the project management details and personal reflections on the project. It then presents the takeaways from this project and plans of future work about the features that are not achieved in this project.

5.1 Project management

The project started with literature reviews on topics related to MT and AV. With the background information, the MT testing plan were carried out. Then the major work was on implementing automatic testing scripts to test Baidu Apollo for certain number of rounds. Writing automatic test scripts took a lot of time because it requires the understanding of Baidu Apollo software architecture, ROS concepts embedded in Apollo and the work flow of obtaining data from Dreamview simulation. Because Baidu Apollo is a sophisticated software system that combines various technologies, getting familiar with Apollo and interacting with its data is challenging.

The first semester of the academic year was mainly devoted to literature reviews on MT, FT and AV. The relevant background information was collected as the basis for writing the dissertation. Then, a landmark achievement of this project was installing Baidu Apollo successfully and a demo simulation was ran on Dreamview, which laid foundations for the future testing. Installing Apollo was challenging as it can only run Ubuntu, so I finally utilised the machines in AILab at UNNC. After building Apollo, the test plan and experimental design were initially determined and the rest of the time was mainly on writing the interim report.

Figure 5.1 shows the original project timeline conducted in the middle of the academic year. The work was mostly completed according to the plan in the first half of the academic year, and a general MT plan was designed. One thing that did not go as planned was that the testing script was not completed and thus this part of work is rearranged to the second half of the year.

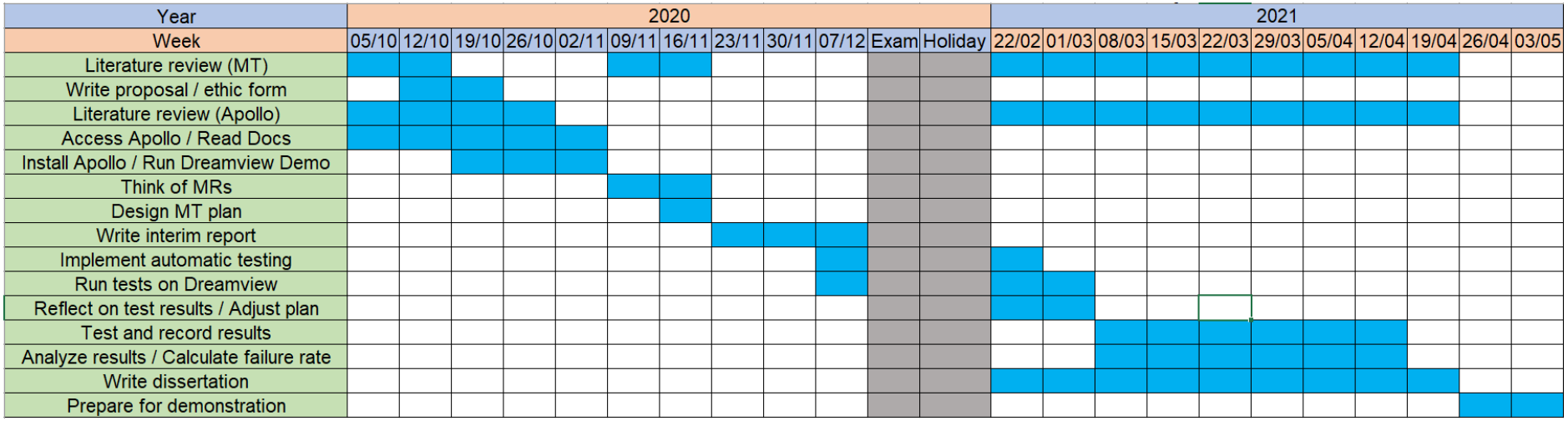

Figure 5.1: Original Project Timeline

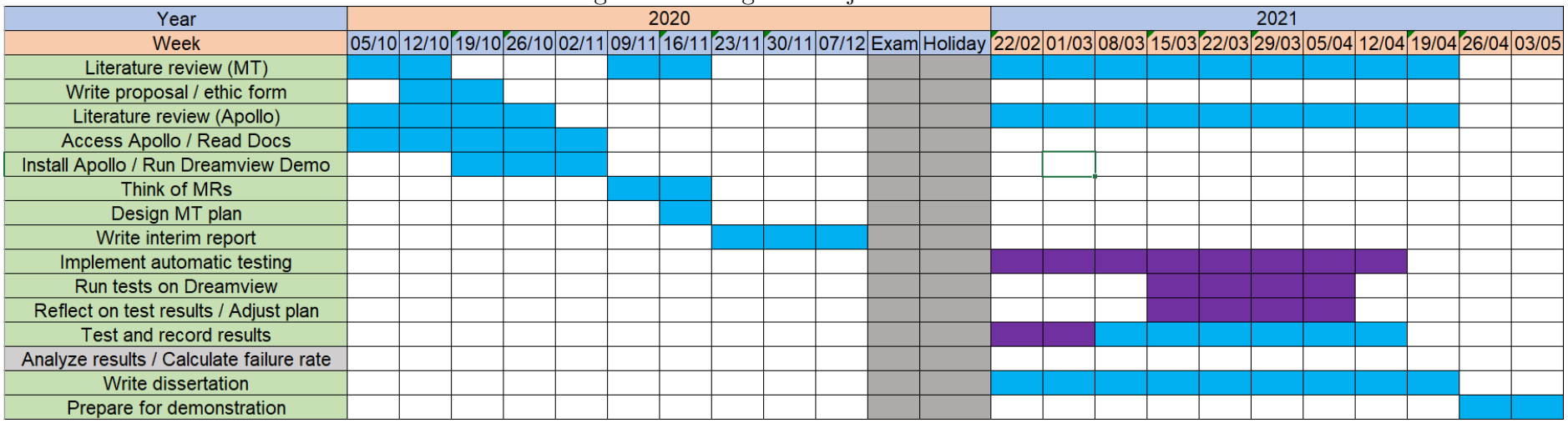

Figure 5.2: Updated Project Timeline

Figure 5.2 is the new project timeline updated by the end of this project, certain modifications were made due to the change of my working plan and progress. One significant change is the work time of automatic testing script development was extended due to the difficulties encountered during the script writing. The corresponding work of testing plan adjustment and result analysis changed as well. Another major change was that the section of calculating genuine failure rate was cancelled as the testing script could not generate valid follow-up cases that can be used to implement the metamorphic stage of the testing. The dark purple color indicates the differences of work timeline between the original one and the updated one.

During the winter vocation, there was not much progress on the project. Because running Apollo requires an Ubuntu system, I cannot run simulations on Apollo or examine the test scripts on my Windows laptop. I tried using virtual machine to simulate the Ubuntu OS in the beginning of the project, but I failed because the GPU of my laptop is not capable of running these simulations. Therefore, the test script development progress was interrupted. And the main focus was on MR identification and more literature reviews.

In the second semester, the major work was getting familiar with Baidu Apollo code structure and studying ROS-related commands to implement the automatic test scripts. With the Ubuntu machine, the test scripts was updated continuesly, and the understanding of Apollo software was better. A lot of useful functions that interact with Apollo were written, and numerous data were generated and fed into Baidu Apollo system, and the data received from it were also analysed. Many simulations were carried out and the test scripts were optimised to improve the efficiency and accuracy of testing. Simultaneously, the final dissertation was started, and new results obtained as well as problems encountered were included in the dissertation.

5.2 Project Summary and Reflections

The overall project outcomes is acceptable, despite the most important part of the MT of Baidu Apollo was not carried out due to the difficulties of replicating simulations. After doing this project, I gained a lot of knowledge about MT and FT. These are effective testing techniques that can be applied to complicated software other than AV, for example, search engine, video games and other security-critical software. In my future studies and work, when I design tests for the software I develop or maintain, MT and FT can be adopted to test them and find vulnerabilities that are difficult to find by unit tests. As a pure computer scientist student, I had little background of writing a formal dissertation. During this project, I obtained a lot of experience on literature review and academic writing. The test scripts for Apollo were implemented and ran on Ubuntu OS, and thus gained much experience about Linux system management, Docker usage and ROS framework. After writing the test script to implement the data interaction with Baidu Apollo, I have a better understanding of the AV system and the functionality of Baidu Apollo.

Apart from these valuable takeaways from doing this project, there are also certain lessons I have learned from it. Firstly, I didn't reasonably estimate the workload of test scripts development, and the experiment was not carried out as expected. After I successfully

wrote the test script of generating obstacles for the first time, I thought the test scripts will not be difficult to implement. However, when I get to the advanced features of Baidu Apollo, for example, the code for publishing routing requests had bugs that the routing could not be published successfully. It took me about one day to debug the scripts, and it was because the waypoint id of the routing should be exactly the same as the routing information generated after clicking the routing choice in Dreamview. There were many similar problems in the scripts, so I spent a lot of time dealing with these issues.

5.3 Future Work

In the future, I will keep on investigating Baidu Apollo and its code implementation, because I want to figure out the reason why the time synchronization can not achieved. This requires deeper understanding of Baidu Apollo working mode, and the functionality of different modules and their connections. Because Baidu Apollo is an open source project, I will possibly contribute my successful testing work to this project which is beneficial to Apollo group and the open source community.

The concepts and implementation of MT and FT deserve further research because they can be utilised in testing complicated software in my future work. Nowadays, the amount of software published is increasing rapidly every year, and corresponding testing work should also be conducted to ensure the software quality and user experience. If I become a software engineer or a testing engineer in the future, MT and FT-related approaches can be adopted to implement the tests and these approaches probably will identify software bugs that are difficult for traditional testing methods to discover.

References

[1] Fábio Duarte and Carlo Ratti. The Impact of Autonomous Vehicles on Cities: A Review. *Journal of Urban Technology*, 25(4):3–18, 2018.

[2] Lin Zhang, Robert Merrifield, Anton Deguet, and Guang-Zhong Yang. Powering the world's robots - 10 years of ROS. *Science Robotics*, 2:1–5, October 2017.

[3] Scott Drew Pendleton, Hans Andersen, Xinxin Du, Xiaotong Shen, Malika Meghjani, You Hong Eng, Daniela Rus, and Marcelo H. Ang. Perception, planning, control, and coordination for autonomous vehicles. *Machines*, 5(1):1–54, 2017.

[4] Alberto Broggi, Pietro Cerri, Stefano Debattisti, Maria Chiara Laghi, Paolo Medici, Daniele Molinari, Matteo Panciroli, and Antonio Prioletti. PROUD-Public Road Urban Driverless-Car Test. *IEEE Transactions on Intelligent Transportation Systems*, 16(6):3508–3519, 2015.

[5] Dorde Petrovic, Radomir Mijailović, and Dalibor Pešić. Traffic Accidents with Autonomous Vehicles: Type of Collisions, Manoeuvres and Errors of Conventional Vehicles' Drivers. *Transportation Research Procedia*, 45(2019):161–168, 2020.

[6] Linker Sheldon and Linker Shannah. A Recap of the Life and Death Matters of the Current State of Artificial Intelligence Programming. `http://linker.com/sol/RecapLifeNDeathMattersCurrentStateAI.pdf`, 2019.

[7] Fabian Pütz, Finbarr Murphy, and Martin Mullins. Driving to a future without accidents? Connected automated vehicles' impact on accident frequency and motor insurance risk. *Environment Systems and Decisions*, 39(4):383–395, 2019.

[8] Bo Yang, Xuelin Cao, Xiangfang Li, Chau Yuen, and Lijun Qian. Lessons learned from accident of autonomous vehicle testing: An edge learning-aided offloading framework. *IEEE Wireless Communications Letters*, 9(8):1182–1186, 2020.

[9] Yangcheng Luo. Autonomous vehicles and road transportation safety. 2018.

[10] Petr Stefan, Vojtech Horky, Lubomir Bulej, and Petr Tuma. Unit testing performance in Java projects: Are we there yet? *ICPE 2017 - Proceedings of the 2017 ACM/SPEC International Conference on Performance Engineering*, pages 401–412, 2017.

[11] Jia Cheng Han and Zhi Quan Zhou. Metamorphic Fuzz Testing of Autonomous Vehicles. *Proceedings - 2020 IEEE/ACM 42nd International Conference on Software Engineering Workshops, ICSEW 2020*, pages 380–385, May 2020.

[12] Earl T Barr, Mark Harman, Phil Mcminn, Muzammil Shahbaz, and Shin Yoo. The Oracle Problem in Software Testing : A Survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

[13] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. Metamorphic Testing: Challanges and Oppotunities. *ACM Computing Surveys*, 51(1):1–27, 2018.

[14] Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. Finding software vulnerabilities by smart fuzzing. *Proceedings - 4th IEEE International Conference on Software Testing, Verification, and Validation, ICST 2011*, pages 427–430, 2011.

[15] Hong Zhu. JFuzz: A Tool for Automated Java Unit Testing Based on Data Mutation and Metamorphic Testing Methods. *Proceedings - 2nd International Conference on Trustworthy Systems and Their Applications, TSA 2015*, pages 8–15, 2015.

[16] T.Y. Chen, D.H. Huang, T.H. Tse, and Z.Q. Zhou. Case studies on the selection of useful relations in metamorphic testing. *Proceedings of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering (JIISIC 2004)*, pages 569–583, 2004.

[17] Johannes Mayer and Ralph Guderleit. An empirical study on the selection of good metamorphic relations. *Proceedings - International Computer Software and Applications Conference*, 1:475–484, 2006.

[18] Marc Weber. Where to? A History of Autonomous Vehicles. `https://computerhistory.org/blog/where-to-a-history-of-autonomous-vehicles/?key=where-to-a-history-of-autonomous-vehicles`, 2014.

[19] CMU. The Carnegie Mellon University Autonomous Land Vehicle Project (NAVLAB). `https://www.cs.cmu.edu/afs/cs/project/alv/www/index.html`, 2020.

[20] Kanade Takeo, Thorpe Chuck, and Whittaker William. Autonomous land vehicle project at CMU. In *Proceedings of the 1986 ACM Fourteenth Annual Conference on Computer Science*, CSC '86, pages 71–80, New York, NY, USA, 1986. Association for Computing Machinery.

[21] Richard Wallace, Anthony Stentz, Charles Thorpe, Hans Moravec, William Whittaker, and Takeo Kanade. First Results in Robot Road-Following. In *IJCAI*, pages 1089–1095. Citeseer, 1985.

[22] Abdur R. Fayjie, Sabir Hossain, Doukhi Oualid, and Deok Jin Lee. Driverless Car: Autonomous Driving Using Deep Reinforcement Learning in Urban Environment. *2018 15th International Conference on Ubiquitous Robots, UR 2018*, pages 896–901, 2018.

[23] Baidu. Baidu apollo autonomous driving github repository. `https://github.com/ApolloAuto/apollo/blob/master/README.md`, 2020.

[24] Andrew J. Hawkins. Uber's fraught and deadly pursuit of self-driving cars is over. `https://www.theverge.com/2020/12/7/22158745/uber-selling-autonomous-vehicle-business-aurora-innovation`, 2020.

[25] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. ROS: an open-source Robot Operating System. *ICRA workshop on open source software*, 3:5, 2009.

[26] Ivan Culjak, David Abram, Tomislav Pribanic, Hrvoje Dzapo, and Mario Cifrek. A brief introduction to OpenCV. *MIPRO 2012 - 35th International Convention on Information and Communication Technology, Electronics and Microelectronics - Proceedings*, pages 1725–1730, 2012.

[27] Shinpei Kato, Eijiro Takeuchi, Yoshio Ishiguro, Yoshiki Ninomiya, Kazuya Takeda, and Tsuyoshi Hamada. An open approach to autonomous vehicles. *IEEE Micro*, 35(6):60–68, 2015.

[28] Tyler G.R. Reid, Sarah E. Houts, Robert Cammarata, Graham Mills, Siddharth Agarwal, Ankit Vora, and Gaurav Pandey. Localization requirements for autonomous vehicles. *arXiv preprint arXiv:1906.01061*, June 2019.

[29] Pedro F. Felzenszwalb, Ross B. Girshick, David McAllester, and Deva Ramanan. Object detection with discriminatively trained part-based models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(9):1627–1645, 2010.

[30] Chenxi Tu, Eijiro Takeuchi, Alexander Carballo, and Kazuya Takeda. Point cloud compression for 3d lidar sensor using recurrent neural network with residual blocks. In *Proceedings - IEEE International Conference on Robotics and Automation*, pages 3274–3280. Institute of Electrical and Electronics Engineers Inc., May 2019.

[31] Braden Hurl, Krzysztof Czarnecki, and Steven Waslander. Precise synthetic image and LiDAR (PreSIL) dataset for autonomous vehicle perception. In *IEEE Intelligent Vehicles Symposium, Proceedings*, pages 2522–2529. Institute of Electrical and Electronics Engineers Inc., June 2019.

[32] Seyed Reza Shahamiri, Wan Mohd Nasir Wan Kadir, Suhaimi Ibrahim, and Siti Zaiton Mohd Hashim. An automated framework for software test oracle. *Information and Software Technology*, 53(7):774–788, July 2011.

[33] S. Segura, D. Towey, Z. Q. Zhou, and T. Y. Chen. Metamorphic testing: Testing the untestable. *IEEE Software*, 37(3):46–53, 2020.

[34] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. *arXiv*, pages 2123–2138, 2018.

[35] Daniel S. Fowler, Jeremy Bryans, Siraj Ahmed Shaikh, and Paul Wooderson. Fuzz Testing for Automotive Cyber-Security. In *Proceedings - 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops, DSN-W 2018*, pages 239–246. Institute of Electrical and Electronics Engineers Inc., 2018.

[36] Brian S Pak. Hybrid Fuzz Testing: Discovering Software Bugs via Fuzzing and Symbolic Execution. *School of Computer Science Carnegie Mellon University*, page 80, 2012.

[37] Patrice Godefroid. Random testing for security: blackbox vs. whitebox fuzzing. *International Conference on Automated Software Engineering (ASE 2007)*, page 1, 2007.

[38] Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE: Whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.

[39] Jie Liang, Mingzhe Wang, Yuanliang Chen, Yu Jiang, and Renwei Zhang. Fuzz testing in practice: Obstacles and solutions. *25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 562–566, 2018.

Appendix A

Source Code Used for Testing

A.1 fuzzer.py

```
import random, time
import numpy as np
from numpy import arctan, cos, sin, genfromtxt
import rospy
from routing_pub import talker as rtalker
from pb_msgs.msg import PerceptionObstacles
from modules.routing.proto import routing_pb2


# calculate the tansformed coordinates of x and y
# according to the angle and distance between center to origin
def transform(x, y, angle, x_center, y_center, compress_x_y):
    # first translate to the origin
    x = x - x_center
    y = y - y_center

    # compress the width of the region
    if (compress_x_y == 0):
        x = 0.2 * x
    if (compress_x_y == 1):
        y = 0.2 * y

    # apply rotation
    x = x * cos(angle) - y * sin(angle)
    y = x * sin(angle) + y * cos(angle)

    # translate back to the previous position
    x_transform = x + x_center
    y_transform = y + y_center

    return x_transform, y_transform


def talker():
    pub_obstacle = rospy.Publisher('/apollo/perception/obstacles',
    PerceptionObstacles, queue_size=10)
    pub_routing = rospy.Publisher('/apollo/routing_request', routing_pb2
    .RoutingRequest, queue_size=10)

    rospy.init_node('talker', anonymous=True)

    sequence_num = 0

```

```
    msg_routing_request = routing_pb2.RoutingRequest()
    msg_routing_request.header.timestamp_sec = rospy.get_time()
    msg_routing_request.header.module_name = 'routing_request'
    msg_routing_request.header.sequence_num = sequence_num
    sequence_num = sequence_num + 1

    # load data from data/waypoints.csv
    routing_data = genfromtxt('/apollo/auto-test/data/waypoints.csv',
    delimiter=',',dtype=None)
    len_waypoints = len(routing_data)

    for point in range(len_waypoints):
        curr_point = routing_data[point]
        msg = msg_routing_request.waypoint.add()
        msg.id = curr_point[0]
        msg.s = curr_point[1]
        msg.pose.x = curr_point[2]
        msg.pose.y = curr_point[3]

    # print(msg_routing_request)
    # wait for 2 seconds to let the message published successfully
    # if time is too short, the message may be ommited by the system
    time.sleep(2.0)
    pub_routing.publish(msg_routing_request)

    # define the frequency of refresh (Hz)
    rate = rospy.Rate(0.2)

    # read data from routings.csv
    routings = genfromtxt('/apollo/auto-test/data/routings.csv', dtype=
    int, delimiter=',')
    # select the routing by id
    routing_id = 5 # 'Sv sunrise loop'
    # routing = routings[routings[:,0] == routing_id]
    routing = routings[routings[:,0] == routing_id][0]
    # extract the start and end coordinates from the list
    start_x = routing[1.1]
    start_y = routing[2]
    end_x = routing[3]
    end_y = routing[4]


    # define the boundary of region where obstacles are generated
    # using the start and end points in this case
    bound_left = min(end_x, start_x)
    bound_right = max(end_x, start_x)
    bound_up = max(end_y, start_y)
    bound_down = min(end_y, start_y)

    center_x = 0.5 * (start_x + end_x)
    center_y = 0.5 * (start_y + end_y)

    # get the tan value of angle of the road according to the horizontal
    line
    tan_value = abs(float(end_y-start_y) / float(end_x-start_x))
    # get the angle
    # road_angle = arctan(tan_value)
    # print(road_angle)
```

```
97
98      # determine whether the region is general horizontal or vertical
99      if abs(start_x-end_x) > abs(start_y-end_y):
100       compress = 1
101       angle = arctan(tan_value)
102         # negate the angle if counter-clockwise
103         if (start_x-end_x)*(start_y-end_y) < 0:
104             angle = -angle
105     else:
106       compress = 0
107         angle = arctan(1/tan_value)
108         # negate the angle if counter-clockwise
109         if (start_x-end_x)*(start_y-end_y) > 0:
110             angle = -angle
111
112     # calculate the area of the region
113     area_region = (bound_right - bound_left) * (bound_up - bound_down)
114     # define the obstacle density (0 - 1)
115     obstacle_density = 0.003
116
117     # define number of obstacles
118     n_obstacles = int(obstacle_density * area_region)
119
120     # set scenario id to 1
121     # scenario_id = 1
122     # add a large number to obstacle id to make them unique among all
    simulation scenarios
123     id_prefix = 10000
124     # denfine number of scenarios
125     scenario_num = 6
126
127     for scenario_id in range(1, scenario_num+1):
128         # create PerceptionObstacles object
129         msg_obstacles = PerceptionObstacles()
130         # print('Scenario/ ID: %d' % scenario_id)
131         for obs in range(n_obstacles):
132             msg = msg_obstacles.perception_obstacle.add()
133             msg.id = obs + scenario_id * id_prefix
134             # randomly assign x and y coordinates to the obstacle
135             x = random.uniform(bound_left, bound_right)
136             y = random.uniform(bound_down, bound_up)
137
138             # apply transformation to fit the region to the routing
139             x_, y_ = transform(x, y, angle, center_x, center_y, compress
    )
140
141             msg.position.x = x_
142             msg.position.y = y_
143             msg.position.z = 0
144
145             # assign random theta to the obstacle
146             # theta represents the heading direction of the obstacle
147             msg.theta = 1.0
148
149             # custom obstacle dimentions
150             msg.length = random.uniform(1, 4)
151             msg.width = random.uniform(1, 4)
152             msg.height = random.uniform(1, 4)
```

```
153
154             # define the type of the obstacle (default 10: general
    obstacle)
155             msg.type = random.randrange(0, 5)
156
157             # gather the obstacle information in array
158             # id, scenario_id, position_x, position_y, position_z,
    direction, length, width, height, type
159             obstacle_data = np.array([[msg.id, scenario_id, msg.position
    .x, msg.position.y,
160                                 msg.position.z, msg.theta, msg.length, msg.
    width, msg.height, msg.type]])
161
162             # the reason why obstacle_data is a 2D array is that using a
     1D array will insert a column of data
163             # to the file rather than a row. Thus, this is a hack to fix
     the bug.
164
165             # add new obstacle data into the csv file
166             with open('/apollo/auto-test/data/obstacles.csv', 'a') as
    csv:
167                 np.savetxt(csv, obstacle_data, fmt = ['%d','%d','%.4f','
    %.4f','%.4f','%.4f','%.4f','%.4f','%.4f','%d'], delimiter=',')
168
169         # increment the scenario counter
170         scenario_id = scenario_id + 1
171
172         # publish the obstacles to ROS topic '/apollo/perception/
    obstacles'
173         pub_obstacle.publish(msg_obstacles)
174         rate.sleep()
175
176     # # publish an empty set of obstacles to reset the scenario for the
    next test case
177     # msg_obstacles = PerceptionObstacles()
178     # pub_obstacle.publish(msg_obstacles)
179
180
181
182 if __name__ == '__main__':
183     try:
184         talker()
185
186     except rospy.ROSInterruptException:
187         pass
```

Listing A.1: *fuzzer.py*

A.2 collision-detect.py

```
1 #!/usr/bin/env python
2 import numpy as np
3 import rospy
4 from pb_msgs.msg import MonitorMessage
5 import sys, os, time
6 # from modules.common.monitor_log.proto.monitor_log_pb2 import
    MonitorMessage
7
```

```
def monitorCallback(monitorMessage):
    # get the monitor message item
    monitorMessageItem = monitorMessage.item[0]
    # extract message from the item
    msg = monitorMessageItem.msg
    # expected message when collision is detected
    collisionMessage = 'Found collision with obstacle: '

    # check if the message contains collision information
    if (collisionMessage in msg):
        # extract the obstacle id from the message
        # need to consider when id has '_0' suffixes which cannot be convert to int directly
        if ('_' in msg):
            # find the index of of '_' in the msg
            position = msg.find('_')
            # drop the chars include and after '_' to get a proper int
            msg = msg[:position]
        obstacle_id = int(msg.replace(collisionMessage, ''))
        obstacle = [obstacle_id]

        #print(obstacle_id)

        # add new obstacle data into the csv file, path is determined at the start of main
        with open(file_dest, 'a') as csv:
            np.savetxt(csv, obstacle, fmt='%d', delimiter=',')

        # print("Collision detect.")

        # terminate fuzzer or metamorphic script

        # exit once collision is detected
        # os._exit(1)

def listener():
    rospy.init_node('collision_detect', anonymous=True)
    rospy.Subscriber('/apollo/monitor', MonitorMessage, monitorCallback)
    rospy.spin()

if __name__ == '__main__':
    # check the argument vector to obtain the destination of the output file
    if (sys.argv[1] == 'src'):
        file_dest = '/apollo/auto-test/data/collision.csv'
    elif (sys.argv[1] == 'follow'):
        file_dest = '/apollo/auto-test/data/collision_new.csv'
    else:
        print("Invalid arguments for collision_detect.py")
        sys.exit()

    time.sleep(3)
    listener()
```

Listing A.2: *collision – detect.py*

A.3 metamorphic.py

```
import numpy as np
from numpy import genfromtxt, arctan, cos, sin
import os, sys, time
from routing_pub import talker as rtalker
import rospy
from pb_msgs.msg import PerceptionObstacles
from modules.routing.proto import routing_pb2


# load data from data/obstacles.csv and data/collision.csv
obstacles_data = genfromtxt('/apollo/auto-test/data/obstacles.csv',
    delimiter=',')

# check whether collision.csv exists, it should not exist if no
    collision is detected
if (not os.path.exists('/apollo/auto-test/data/collision.csv')):
    print('No collision detected in the simulation!')
    sys.exit()

collision_data = genfromtxt('/apollo/auto-test/data/collision.csv',
    delimiter=',')

# remove duplicated data in collision_data
collision_data = np.unique(collision_data)
collision_id = collision_data[0]
print(collision_id)

# # find the information of all the collision obstacles in obstacle_data
# collision_obstacle_data = np.zeros([collision_size, 10])
# for c in range(collision_size):
#     obstacle_info = obstacles_data[obstacles_data[:,0] ==
    collision_data[c]]
#     collision_obstacle_data[c] = obstacle_info[0]



# print collision rate
# get the number of scenarios generated
scenario_number = int(obstacles_data[-1][1])
obstacle_number = int(obstacles_data.shape[0])
# check whether Obstacle number is divisable by scenario number
if (obstacle_number % scenario_number != 0):
    print("Error: Obstacle number is not divisable by scenario number!
    Something goes wrong!")
    sys.exit()


obstacle_per_scenario = obstacle_number / scenario_number

# collision_analysis = np.array([[obstacles_data.shape[0],
    scenario_number, collision_size, general_rate]])
# with open('/apollo/auto-test/data/collision_analysis.csv', 'a') as csv
    :
#     np.savetxt(csv, collision_analysis, fmt=['%d','%d','%d','%.6f'],
    delimiter=',')

# regenerate the follow-up scenarios based on the obstacle data and MR
# MR: Suppose that in a driving scenario,S, a car collided with a static
```

```
    obstacle O at location L,
#     Construct a follow-up driving scenario,S', that is identical to S
    except that O is repositioned
#     slightly further away from the previous position. Run the follow-
    up driving scenario. The car
#     should not stop before L.

# generate the obstacles based on obstacle.csv
pub_obstacle = rospy.Publisher('/apollo/perception/obstacles',
    PerceptionObstacles, queue_size=10)
pub_routing = rospy.Publisher('/apollo/routing_request', routing_pb2.
    RoutingRequest, queue_size=10)
rospy.init_node('talker', anonymous=True)
# define the frequency of refresh (Hz)
rate = rospy.Rate(0.2)

# send routing request
sequence_num = 0

msg_routing_request = routing_pb2.RoutingRequest()
msg_routing_request.header.timestamp_sec = rospy.get_time()
msg_routing_request.header.module_name = 'routing_request'
msg_routing_request.header.sequence_num = sequence_num
sequence_num = sequence_num + 1

# load data from data/waypoints.csv
routing_data = genfromtxt('/apollo/auto-test/data/waypoints.csv',
    delimiter=',',dtype=None)
len_waypoints = len(routing_data)

for point in range(len_waypoints):
    curr_point = routing_data[point]
    msg = msg_routing_request.waypoint.add()
    msg.id = curr_point[0]
    msg.s = curr_point[1]
    msg.pose.x = curr_point[2]
    msg.pose.y = curr_point[3]

# print(msg_routing_request)
# wait for 2 seconds to let the message published successfully
# if time is too short, the message may be ommited by the system
time.sleep(2.0)
pub_routing.publish(msg_routing_request)

road_angle = 1.33015289662

# generate the same number of scenarios
for scenario in range(1, scenario_number+1):
    # print('Scenario ID: %d' % scenario)
    # create PerceptionObstacles object
    msg_obstacles = PerceptionObstacles()
    # obtain the data for the scenario of the loop
    obs_info = obstacles_data[obstacles_data[:,1] == scenario]
    for obs in range(obstacle_per_scenario):
        obs_info_one = obs_info[obs]
        msg = msg_obstacles.perception_obstacle.add()
        msg.id = int(obs_info_one[0])
        msg.position.x = obs_info_one[2]
```

```
        msg.position.y = obs_info_one[3]
        msg.position.z = obs_info_one[4]
        # todo move obstacle further away
        # if (int(obs_info_one[0]) == collision_id):
        #     msg.position.x = msg.position.x - 4.0 * cos(road_angle)
        #     msg.position.y = msg.position.y - 4.0 * sin(road_angle)

        msg.theta = obs_info_one[5]
        msg.length = obs_info_one[6]
        msg.width = obs_info_one[7]
        msg.height = obs_info_one[8]
        msg.type = int(obs_info_one[9])

    pub_obstacle.publish(msg_obstacles)
    rate.sleep()

# # publish an empty set of obstacles to reset the scenario for the next test case
# msg_obstacles = PerceptionObstacles()
# pub_obstacle.publish(msg_obstacles)

if (not os.path.exists('/apollo/auto-test/data/collision_new.csv')):
    print('No collision detected in the follow-up simulation!')
    sys.exit()

collision_data_new = genfromtxt('/apollo/auto-test/data/collision_new.csv', delimiter=',')

# remove duplicated data in collision_data
collision_data_new = np.unique(collision_data_new)

print('collosions 1: ', collision_data)
print('collosions 2: ', collision_data_new)

collision_cmp = np.array([collision_data, collision_data_new])
# add new obstacle data into the csv file, path is determined at the start of main
with open('/apollo/auto-test/data/collision_compare.csv', 'a') as csv:
    np.savetxt(csv, collision_cmp, fmt='%d', delimiter=',')

if (np.array_equal(collision_data, collision_data_new)):
    print('Identical repruduction!')

print('============================================================')
```

Listing A.3: *metamorphic.py*

A.4 routing-sub.py

```
import rospy
import numpy as np

from pb_msgs.msg import RoutingResponse, RoutingRequest

def routingRequestCallback(RoutingRequest):
    print(RoutingRequest)
    print("============================================================")
```

```
9
10 def listener_request():
11     rospy.init_node('routing_request', anonymous=True)
12     rospy.Subscriber('/apollo/routing_request', RoutingRequest,
   routingRequestCallback)
13     rospy.spin()
14
15 if __name__ == '__main__':
16     listener_request()
17     # listener_respond()
```

Listing A.4: *routing_sub.py*

A.5 routing-pub.py

```
1  import rospy
2  import time
3  from numpy import genfromtxt
4  from pb_msgs.msg import RoutingResponse, RoutingRequest
5  from modules.routing.proto import routing_pb2
6
7  def talker():
8      pub = rospy.Publisher('/apollo/routing_request', routing_pb2.
   RoutingRequest, queue_size=1)
9      # rospy.init_node('routing_request', anonymous=True)
10
11     sequence_num = 0
12
13     msg_routing_request = routing_pb2.RoutingRequest()
14     msg_routing_request.header.timestamp_sec = rospy.get_time()
15     msg_routing_request.header.module_name = 'routing_request'
16     msg_routing_request.header.sequence_num = sequence_num
17     sequence_num = sequence_num + 1
18
19     # load data from data/waypoints.csv
20     routing_data = genfromtxt('/apollo/auto-test/data/waypoints.csv',
   delimiter=',',dtype=None)
21     len_waypoints = len(routing_data)
22
23     for point in range(len_waypoints):
24         curr_point = routing_data[point]
25         msg = msg_routing_request.waypoint.add()
26         msg.id = curr_point[0]
27         msg.s = curr_point[1]
28         msg.pose.x = curr_point[2]
29         msg.pose.y = curr_point[3]
30
31     # print(msg_routing_request)
32     # wait for 2 seconds to let the message published successfully
33     # if time is too short, the message may be ommited by the system
34     time.sleep(2.0)
35
36     pub.publish(msg_routing_request)
37
38 if __name__ == '__main__':
39     try:
40         talker()
41     except rospy.ROSInterruptException:
```

```
        pass
```

Listing A.5: $routing_sub.py$

A.6 auto-test.sh

```
# trap "kill 0" EXIT # kill all background process on exit

# run collision_detect.py and fuzzer.py in parallel
round=50
curr_round=1

while [ $curr_round -le $round ]
do
    echo "
    ================================================================"
    echo "Test round: $curr_round"
    echo "Source test case generating..."
    python auto-test/collision_detect.py src &
    python auto-test/fuzzer.py

    # wait -n
    pkill -P $$

    echo "Follow-up test case generating..."
    python auto-test/collision_detect.py follow &
    python auto-test/metamorphic.py

    # wait -n
    pkill -P $$

    # remove the old csv files
    rm -f /apollo/auto-test/data/obstacles.csv /apollo/auto-test/data/
    collision.csv /apollo/auto-test/data/collision_new.csv
    curr_round=$(( curr_round+1 ))
done
```

Listing A.6: $auto - test.sh$