UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

GABRIEL STEFANIAK NIEMIEC

Design and Implementation of a Fault Injection Prototype on an Autonomous Vehicle Simulator

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Engineering

Advisor: Prof. Dr. Taisy Silva Weber

Porto Alegre
May 2021

“If you recognize that self-driving cars are going to prevent car accidents, AI will be responsible for reducing one of the leading causes of death in the world.”

— MARK ZUCKERBERG

AGRADECIMENTOS

I thank the Universidade Federal do Rio Grande do Sul for the opportunity of graduating in one of the most esteemed universities in Brazil.

I thank my adviser Dr. Taisy Weber for the counsel and support during the development of this work, without which none of this could have happened.

I thank Dr. Paolo Rech for all the insights he gave me during our correspondence.

Lastly, I thank my family for all the love and support that got me through these stressful times.

ABSTRACT

Autonomous vehicles are the future of transportation. However, progress is slow and large-scale deployment is still far off. Autonomous vehicle simulators can be leveraged to shorten the time-to-market, but they lack the tools to simulate vehicular faults. I developed a fault injection prototype for the CARLA autonomous vehicle simulator capable of simulating any fault to any sensor. It consists of a Python module and a version of CARLA with two very minor alterations.

Keywords: Autonomous, vehicles, simulator, fault, injection, CARLA.

Design e Implementação de um Protótipo de Injeção de Falhas em um Simulador de Veículos Autônomos

RESUMO

Veículos autônomos são o futuro do transporte. Todavia, o progresso é lento e a comercialização em grande escala ainda está longe. Simuladores de veículos autônomos podem impulsionar o desenvolvimento para encurtar o tempo de comercialização, mas eles não possuem as ferramentas para simular falhas veiculares. Eu desenvolvi um protótipo de injeção de falhas para o simulador de veículos autônomos CARLA capaz de simular qualquer faalha a qualquer sensor. Ele consiste em um módulo Python e uma versão do CARLA com duas alterações bem pequenas.

Palavras-chave: autônomo, veículo, simulador, injeção, falha, CARLA.

LIST OF ABBREVIATIONS AND ACRONYMS

AV Autonomous Vehicle

AVS Autonomous Vehicle Simulator

ROS Robot Operating System

SAE Society of Automotive Engineers

LIST OF FIGURES

CONTENTS

1 INTRODUCTION

There is still a lot of work to be done before fully autonomous vehicles become a part of our day-to-day lives. Development and testing are slow and expensive without the use of simulators. And even with simulators, there are scenarios the simulators can't recreate, such as vehicular faults. These range from faulty sensors to steering issues and they need to be simulated for better development and effective validation.

My goal was to implement fault injection in an open-source autonomous vehicle simulator. Upon analysis, CARLA was chosen as the best-suited candidate. I would focus first on sensor faults, trying to get the most basic prototype functional and then build on top of that. Ideally, the prototype would evolve to support different faults and injecting them on other systems, while also keeping it easy to include on a project.

The actual result was a small library that allows a user to tamper with a sensor's data in any way possible, running on a slightly modified version of CARLA. Time constraints and unfamiliarity with the technology kept me from achieving more, but it is a solid proof of concept.

2 STATE OF THE ART

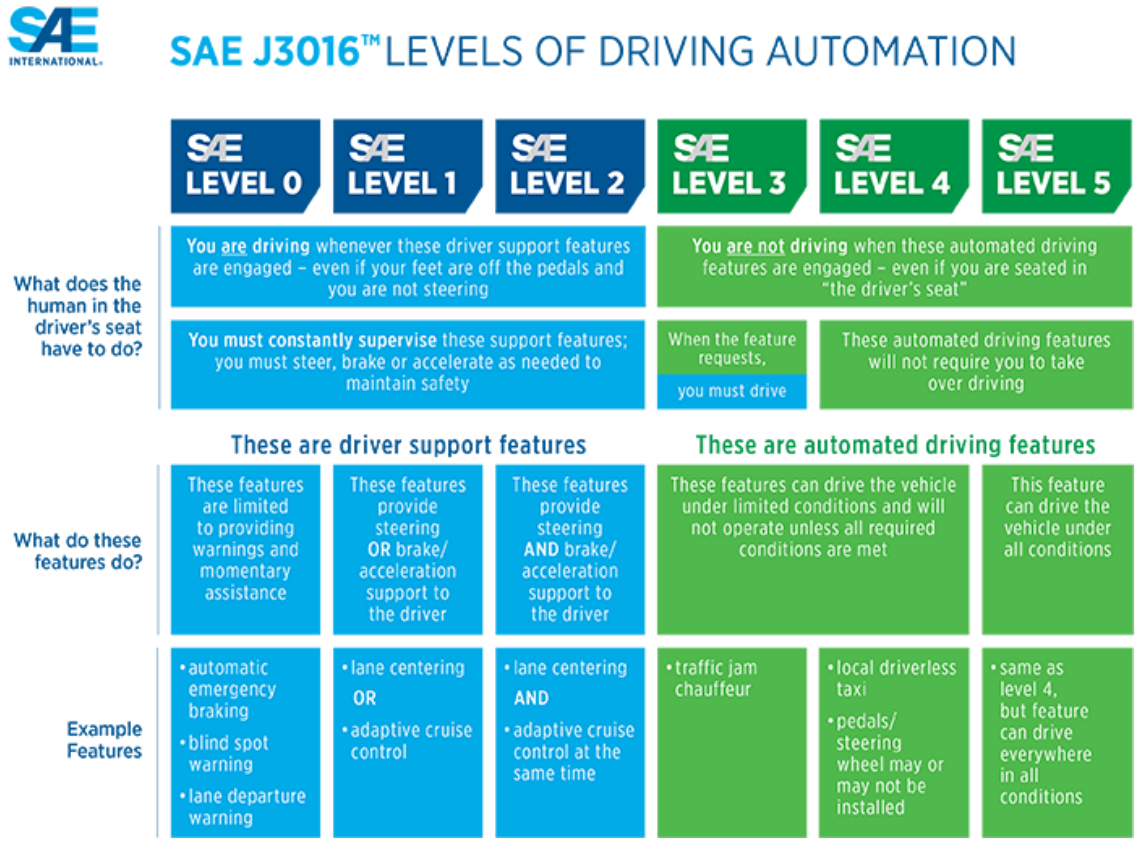
Before discussing the implementation, a few topics will be covered to prepare the reader on the state of the art of autonomous vehicles and how fault tolerance is related to the subject. Ultimately, I hope to better explain why work on this feature is important to the area.

2.1 Theoretic fundamentals

Starting with the main concept, autonomous vehicles are machines capable of locomotion without a driver. Also known as self-driving or driver-less vehicles, they represent the future of urban transportation.

To better describe this technology, the Society of Automotive Engineers proposed a level-based distinction to the nature of a vehicle's automation features 2.1. This grading ranges from level 0, where no automation is present, to level 5, where the vehicle is always driver-less.

Figure 2.1: SAE's levels of driving automation



Source: SAE's website

Level 5 automation still presents many challenges before a full release (Koopman and Wagner (2017)). For instance, according to a study on autonomous vehicles (Banerjee et al. (2018)), results indicate that the AVs are 15 to 4000 times worse than human drivers when it comes to accidents per cumulative mile driven. That study, however, targets level 3 AVs. As will be shown next section, level 4s are already being deployed commercially. And although I couldn't find any data on their accidents, safety remains a primary concern.

2.2 Autonomous vehicles nowadays

Although the large-scale deployment of fully autonomous vehicles won't happen anytime soon, there has been a significant increase of level 4 autonomous vehicles on the roads in the past months. A few private initiatives have arisen that feature fully driver-less vehicles. These cases provide a glimpse into the state of the art of AVs.

It should be stressed how important it is that those vehicles have gotten the green light to transit the same highways as other civilians. It is a testament to their safety and reliability and a major learning opportunity for future models.

Waymo launched last year a fleet of driverless taxis in Phoenix, Arizona (Lee (August, 2020)). The AVs operate only in some of the suburbs and there is still a human supervisor involved, but the supervisor operates remotely and only intercedes in cases where the vehicle is unsure of how to proceed. This presents an interesting possibility to the future of taxis, a business model in which one driver can oversee multiple vehicles, the trade-off being the cost of additional sensors to the standard cab and the study of the service area.

Similar to Waymo, Baidu also launched a driverless taxi initiative (Ramey (May, 2021)), but in Beijing. By calling a vehicle on their ApolloGo app, visitors and athletes will be riding AVs during the 2022 Beijing Winter Olympics.

Nuro, on the other hand, specializes in logistics and has started delivering pizzas using AVs (Phillips and Owens (May, 2021)). In a suburb in Houston, Domino's customers can have their online order be delivered via driver-less vehicle.

2.3 Autonomous vehicle simulators

Autonomous vehicle simulators are tools used to test AVs in a virtual environment. This is done by rendering a specific location with the utmost graphical fidelity possible, including the cars, the buildings, and the pedestrians. They are usually comprised of an engine and a client (or multiple clients), communicating through an API or a communications bridge. The engine renders the scene and the physics while delivering the data to the client. The client defines the simulation, controls the vehicle, and interprets the data delivered from the engine.

In the next subsections, some AVSs are analyzed as possible candidates for fault injection implementation. These were selected for being open-source, a mandatory quality for the feature's development. It should be noted that there are other simulators besides the ones discussed. NVIDIA DRIVE Sim, Cognata, Ansys Autonomous Vehicle Simulation, and rFpro are all commercial solutions to test AVs in a virtual environment. Some can even include human drivers in their tests. It is evident that there is a lot of interest in this topic, if not by the number of simulators being developed then by all the big industry names involved. CARLA is sponsored by Intel and both rFpro and NVIDIA Drive Sim are associated with more than a dozen major car companies.

Having established there is a huge interest in AVSs, an explanation on why they are so valuable is due. Simply put, it is much faster, safer, and cheaper to develop and validate using a simulator than it is to do it live. Take the task of chasing a high-speed vehicle as an example (Jahoda, Cech and Matas (2020)). The implementation can be tested virtually without harming any passengers or bystanders, and with no damages to any property nor the prototype. The system can also be validated in any place, weather, or pedestrian/traffic setup. With cloud computing and automated tests, many tests can occur at once without any human supervision.

2.3.1 CARLA

The first simulator to be discussed is CARLA (Dosovitskiy et al. (2017)), which stands for "Car Learning to Act". CARLA is an open-source AVS based on a server-client model.

The server was implemented on UE4. The Unreal Engine 4 is a gaming engine used in many current-gen video games. It is a good idea to use a gaming engine as a

simulator: there has been a lot of effort put in by the gaming industry to improve upon its engines.

The client communicates with the server either using either a Python-based API or a ROS-bridge. The client can control the simulation in a number of ways such as changing the map or the weather, spawning a vehicle or a pedestrian as well as reading sensor output and steering the vehicle.

Fault tolerance testing in CARLA is limited to the environment. In other words, it allows assessing whether the designed AV behaves correctly in a specific scenario (for instance, a pedestrian crossing the street on a rainy day) but it cant simulate any physical or software faults on the vehicle itself. To induce those faults, some tinkering is required.

One option would be to add a fault to the AV. However, this would mean changing the AV code directly to produce such faults, which is not very good for a number of reasons. First and foremost, the original AV wouldn't be tested, only a modified version. This could possibly undermine its validation. Also, having to remove the changes for deployment is another source of errors.

The better alternative would be to add faults in the simulator, so only the AVS would change. Furthermore, CARLA's server-client architecture enables having a separate client devoted to handling the fault injection on each test-case scenario.

2.3.2 LGSVL Simulator

Up next is the LGSVL Simulator (Rong et al. (2020)), an AVS developed by LG Electronics America R&D Center. Similar to CARLA, it renders the environment using a game engine (the Unity Engine) and it has a Python API to set up test case scenarios.

The LGSVL Simulator's architecture supports easier communication with different AV stacks. This simulator communicates with the AV only through a communication bridge, configured to the specific protocol used by the AV. Implementations are provided for CyberRT, ROS, and ROS2. This way, AV and AVS integration becomes trivial and the developer can focus on setting up test case scenarios. This simulator also comes with ready support for Baidu's Apollo and Autoware, two AV stacks.

Again, no fault injection is provided. Scenarios can be set to introduce jaywalkers and bad weather, as well as introduce specific traffic behavior, but there is no ready support for faulty sensors or physical issues. The LGSVL Simulator has no way of simulating such problems.

To induce faults on the LGSVL simulator's sensors and other physical components, modifications are required on the simulator and API. The first is needed to configure the fault induction strategy on each component; support should be added to enable the specific fault as well as when it will occur. The latter is mandatory to allow the user to configure the test scenario and enable the induction of said faults.

2.3.3 monoDrive

When trying to download monoDrive, the website presented problems during registration. Despite that setback, I did manage to clone the client repository on GitHub, but it was no good without the simulator. After I managed to get registered, to download the simulator I was given a choice between getting a 30-day free trial or requesting purchase information.

Upon further inspection, I realized that in every mention of monoDrive being open source only the client was mentioned as so. Since monoDrive as a platform is not free to the public, it will not be used here as a fault injection candidate. The rest of this subsection contains the information that was gathered on monoDrive.

The monoDrive platform consists of three products: the monoDrive Simulator, the monoDrive Scenario Editor, and the monoDrive Real to Virtual. The latter two are tools to design and customize scenarios, with the Real to Virtual tool allowing for use of real-world data to be incorporated in the scenario.

The simulator supports distributed batch testing. Communication with the simulator is made through a JSON messaging interface. Clients implementing this interface are provided in C++, LabVIEW, and Python.

2.3.4 AirSim

AirSim (Shah et al. (2017)) is an Unreal Engine plugin used to simulate autonomous vehicles. Unlike the previous simulators, AirSim simulates not only autonomous cars but also autonomous drones. It comes with flight controller support for hardware-in-the-loop and software-in-the-loop simulation.

Unfortunately, there seems to be no support for autonomous car controllers or stacks. However, all communications with the simulation are made through an API that

is available in a number of languages and it includes a standalone library that can be deployed to the actual AV. This ensures the software being tested can be shipped as-is.

AirSim also features a computer vision mode. It is a setting that removes the vehicle and the physics from the simulation leaving only the environment. The user can then use the simulator exclusively to gather footage from the scenario to be used by the machine learning algorithms.

There is no support for fault induction in AirSim. Much like CARLA, a set of calls will need to be added to the API to simulate faults in the vehicle. And although this work is focused on autonomous cars, some of those new sub-routines could be used to test malfunctions in drones as well.

2.3.5 Analysis

None of the simulators analyzed in the previous subsections had any built-in way of fault injection. Since it is necessary to test AVs performing in dangerous circumstances, modifications should be made to existing simulators to support that feature. To this end, this subsection will compare simulators and choose the best candidate for fault injection.

One important step in developing any changes to existing software is validation. More specifically, there has to be a way to assess the newly added functionality. The ideal way of doing so in an AVS is to have an autonomous vehicle compatible with the simulator to act as a baseline. Although it is possible to create one from scratch it would be a significant detour from the original objective stated in this paper. So when comparing the simulators, having ready or partial implementations of AVs is important.

Regarding existing work, the candidate simulators diverge greatly. The LGSVL simulator by itself isn't mentioned much in academia, but the driving stacks it supports have some research poured into them. Sadly, no open-source implementations were found for neither Apollo's nor Autoware's autonomous driving stacks. Work based on AirSim, although substantial, is focused on drone applications and the few car papers presented no public implementation.

CARLA, however, has many open-source implementations online. For example, a vision-based driving agent was developed (Chen et al. (2019)) capable of acing the CARLA benchmark and released all code to the public including different navmeshes for two of CARLA's maps. Being the only simulator with open-source implementations made CARLA the best candidate for developing fault injection support.

2.4 Fault injection

When testing, it is often desirable to evaluate how the system operates under adverse circumstances. Taking the AV as an example, developers will wish to test how it behaves when facing a jaywalker, a rogue vehicle, or even intense weather. They will also want to test its behavior when the system is not running at its best (e.g., unresponsive brakes, faulty steering, glitching sensors). In that context, fault injection serves as the introduction of those conditions to the AV.

To be able to inject faults on an AV can be very valuable. Since at the highest level of automation there will be no user supervision, the AV's will have to be able to deal with any setback entirely on their own. By injecting faults in an AVs system, the developers can assess how well their fault tolerance (AVIZIENIS et al., 2004) mechanisms keep people safe. Fault injection's primary use is, therefore, to help to validate fault tolerance.

Fault injection is usually done through two approaches. The physical approach involves tampering with the system by subjecting it to radiation, introducing changes to the magnetic field around it, or even producing a short-circuit in the system's circuits. It should be noted that currently there is no parallel to this level of tampering in AVSs since usually the mechanical and electrical parts of the vehicle are not directly simulated.

The software approach of fault injection, however, is easily translatable to the simulators. It can be done either at runtime or compile time and consists of tampering with the code or the underlying computer system.

3 DESIGN AND IMPLEMENTATION

Before I start to discuss matters of implementation, I would like to go over what exactly I expected to accomplish.

As was discussed in chapter 2, there is no fault injection in CARLA and it is an important feature to develop fault-tolerant systems and strategies. Starting with the sensors, if we could tamper with a single byte of data from the sensors it would go a long way in assessing the robustness of a given AV prototype. This was my primary goal. Having ensured that, there were some other points I deemed desirable but not mandatory.

One of which was versatility. It is impossible to know what kind of faults would be useful for the user beforehand, so the end solution should be able to introduce to the AV any faults imaginable (preferably in any aspect of the car). I also hoped to design it in a way where fault injection could be set up separately from the AV to avoid tampering with the original code that was to be evaluated in the simulator.

When studying existing work, I noticed how often the solutions tested on CARLA were hardbound to a specific version of the simulator. It made me realize that although going deeper into the code could improve performance by reprocessing sensor data closer to where it was generated, it would also have made it much harder to maintain as CARLA develops. It would further bind my prototype to a specific version, limiting access to only those solutions that were compatible with it. Even worse, it could introduce errors to the simulator itself and invalidate the AVS. I decided then to keep my implementation as simple as possible to ease extendibility and accessibility.

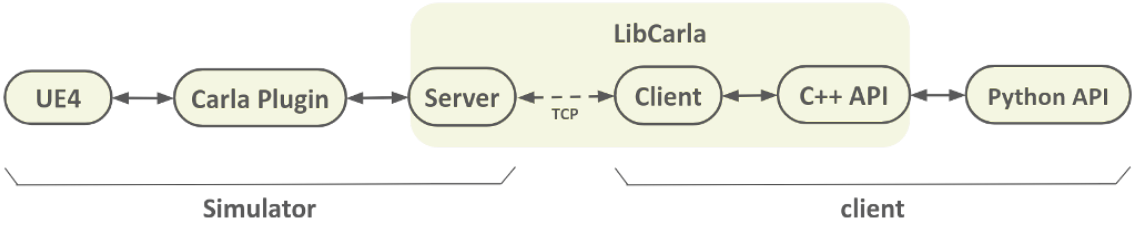
With the above-mentioned goals to orient me, I began designing the prototype.

3.1 First plans

At first, I analyzed the architecture of CARLA, in particular how the user communicates with the simulator (figure 3.1). Since setting up fault injection will be done by the user and the only mean of communication with the simulator is through the Python API, I began observing a typical sensor use case.

The user starts by making a few calls to the API to get a sensor's blueprint and instantiate it. This is how the user chooses which type of sensor he is using and where the sensor will be located relative to the AV. Next, the user passes a function to be run whenever the sensor delivers data (through the "listen" method).

Figure 3.1: CARLA's sensor communication pipeline



Source: CARLA Documentation

A fault could be injected here in two ways: by moving the sensor or by altering the data it delivers. I chose to start with altering the sensor's data because it can be achieved by simply passing another function to the listen method (one that injects faults on the data before the original function can read it). Translation of the sensor will be left for future work.

The method of simply passing another function to the listen method complies with most of the goals at the beginning of this chapter. Any bytes of data could be changed, and it would only need changes close to the start of the pipeline. However, this solution would require the user to explicitly tell how to tamper with the data every time the listen method is called during setup. Ideally setting up the AV and setting up the fault injection should be done separately, but I couldn't find a way to successfully do that.

My first idea was to iterate over the sensors and rerun the listen method with the additional function. To implement that I would need a list of sensors to be tampered with and a way to rerun the listen method with the new fault injection function. The first requirement could be circumvented by having the user pass me the list of sensors to be tampered with (it still would not constitute a completely separate fault injection setup, but it would be a step in the right direction). However, retracing the listen method proved to be impossible.

Once the callback was registered on the listen method, the original function passed as an argument by the user was lost completely. I tried saving that function inside the sensor object (so that I could use it when rerunning the listen method). But I couldn't get working both the added parameter in Boost[reference] (the C++ library the CARLA developers used to program the Python API) and the saved function pointer on the object. As much as I had studied the integration between Python and C++ in the project, it wasn't enough for this task and if I kept insisting on this approach I feared I wouldn't get anything done.

And so I decided to go for the original idea and pass the fault injection function

with the original function on the listen method.

3.2 Working prototype

Having decided on the course of action, I moved on to implementation.

Since I couldn't manage to get the changes into the C++ code, I had to add them as Python libraries. To add this feature to sensors, I created a FaultySensor class in Python (see listing 3.1), one that inherits CARLA's sensor. The only change on this child class from the parent is that the listen method has been overridden to pass a different function (masterCallback), one that first calls upon the fault injection function before calling the original function.

Lastly, two details had to be addressed. Python does not support downcasting, so to turn a CARLA Sensor into a FaultySensor I would have to implement it myself. Type conversion in Python usually involves transferring data from an object before throwing it away. But I feared doing so here could break the simulation by registering the same sensor twice, so I opted for a simpler alternative and wrote a method (ToFaultySensor) that changed the class reference of the sensor into the FaultySensor.

Listing 3.1 – Early prototype

```
import carla

def masterCallback(image, callback, faultCallback):
    faultCallback(image)
    callback(image)

class FaultySensor(carla.Sensor):
    def listen(self, callback, faultCallback):
        super().listen(lambda image: masterCallback(image,
            callback, faultCallback))

def ToFaultySensor(sensor):
    sensor.__class__ = FaultySensor
    return sensor
```

Changing the __class __ variable isn't ideal, and it is frowned upon by the Python

community. But considering how little is added by the new class, and the fact that the sensor won't be instantiated again, this approach carries no risk to the user.

The other detail left to address is how CARLA configured the memory area in which the data delivered by the sensor is stored. The data memory was configured by the CARLA developers as read-only (using the value PyBUF_READ for Python 3 and the method PyBuffer_FromMemory for Python 2), but I needed to be able to write on that memory area to tamper with the data. To remedy this, the value and the method were altered to PyBUF_WRITE for Python 3 and PyBuffer_FromReadWriteMemory for Python 2 (shown in listing 3.2), respectively. This was the only change to library code, and it introduced a side-effect: whereas before if the user tried to change any of the bytes in the buffer an exception would be raised, now any byte could be changed by the user.

Listing 3.2 – Fragment from SensorData.cpp after the read-write changes

```
template <typename T>
static auto GetRawDataAsBuffer(T &self) {
  auto *data = reinterpret_cast<unsigned char *>(self.data
      ());
  auto size = static_cast<Py_ssize_t>(sizeof(typename T::
      value_type) * self.size());
#if PY_MAJOR_VERSION >= 3
  auto *ptr = PyMemoryView_FromMemory(reinterpret_cast<char
      *>(data), size, PyBUF_WRITE);
#else
  auto *ptr = PyBuffer_FromReadWriteMemory(data, size);
#endif
  return boost::python::object(boost::python::handle<>(ptr)
      );
}
```

4 STRATEGIES AND CONTRIBUTIONS

In the previous chapter, I exposed how the basic functionality was implemented, as well as all the compromises made in developing fault injection support in CARLA. This chapter will describe a few additional details that were included in the end prototype (exhibited in the Anex at the end) and some examples of their usage.

4.1 Fault injection strategies

The main features added on top of the basic fault injection mechanism were fault injection strategies.

After converting to a FaultySensor, the user can set one of the new attributes of the class ("FaultySensor.strategy") to one of four values based on the new Strategy enum to configure when are the faults going to be injected. This value is set to CONSTANT by default, meaning the fault injection function will always be called to tamper with the data, which corresponds to the original behavior in listing 3.1.

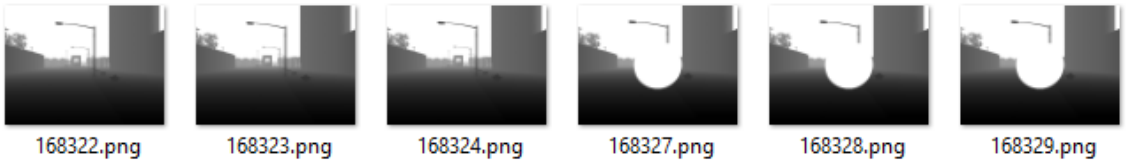
The INTERMITTENT strategy value can be used to introduce faults periodically. To do this, the user must write the "FaultySensor.target" attribute with an integer. Whenever the sensor delivers data, a counter is incremented before running the default function. Whenever the counter reaches the target, the target is reset and the fault injection function is run before the default function. To exemplify this, I ran a depth camera sensor on Depth mode and configured the FaultySensor to draw a white circle every two frames (see image 4.2).

Figure 4.1: Example of the intermittent strategy with a target of 2

Source: taken from a folder where I stored the data captured from a sensor

The TRANSIENT strategy value is used to introduce a fault after a specific amount of times the sensor delivered data, after which the fault will always be injected. To determine the point at which this switch in behavior happens, the attribute "FaultySensor.target" is used. As an example, I ran a FaultySensor on a depth camera sensor LogarithmicDepth mode and told it to draw a white circle after three data deliveries.

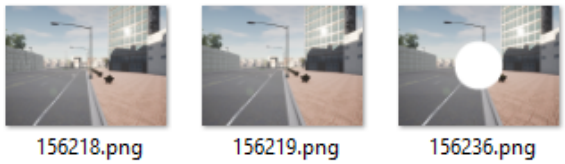
Figure 4.2: Example of the transient strategy with a delay of 3



Source: taken from a folder where I stored the data captured from a sensor

Lastly, the CRASH strategy value is used to silently stop the sensor after a specific amount of data deliveries. Unlike the other strategies, the fault injection function is ran only once on the last data delivery (as a way of corrupting the image before the crash). The "FaultySensor.target" attribute is used to specify after how many data deliveries the crash will occur. Figure 4.3 shows the output of a crash strategy example I set up to crash after three deliveries, this time with an RGB camera sensor.

Figure 4.3: Example of the crash strategy with a delay of 2



Source: taken from a folder where I stored the data captured from a sensor

4.2 Sum of contributions

Having explained exactly what my prototype can do (and how it came to be), I would like to use this space to register all of the contributions included in this work.

With this prototype, a CARLA simulator user can introduce any kind of fault to a sensor's data. It doesn't matter what kind of sensor it is (RGB camera, LIDAR, collision detector, etc.), all are covered by my prototype.

A few injection strategies have been included (constant, intermittent, transient, and crash faults), but it is fairly simple for the user to implement his/her own strategy. And if the user thinks the triggers for these strategies are not optimal, the "Image.frame" and "Image.timestamp" attributes provided by CARLA can also be used to configure fault injection.

It is also moderately easy to incorporate this prototype into an existing AV. It takes two changes on the CARLA source code and the importing of the Python library, and then

the users are free to set up fault injection on their projects with only a couple of lines of Python code.

5 CONCLUSION

Autonomous vehicles are slowly but surely becoming a reality, and simulators can help speed the process tremendously. But to truly validate the AVs, faults must be simulated as well.

In this work, I designed a fault injection proof-of-concept for the open-source autonomous vehicle simulator CARLA, based entirely on its Python API. It is capable of injecting any fault to any sensor, and I added built-in support for a few fault injection strategies (constant faults, transient faults, intermittent faults, and crashes). This prototype can assist greatly with the development and evaluation of autonomous vehicles, and can also provide researchers with a tool to further enhance study in this area.

In the future, I hope to extend this feature to other aspects of the simulation, such as steering and braking. Faults in those systems can be catastrophic, so it is important to simulate them to develop and validate viable fault tolerance tactics.

I would also like to better validate the performance of the fault injection. Depending on the fault that is being injected, performance might not be optimal, in which case alternatives to the implementation would have to be investigated.

REFERENCES

AVIZIENIS, A. et al. Basic concepts and taxonomy of dependable and secure computing. **IEEE Transactions on Dependable and Secure Computing**, v. 1, n. 1, p. 11–33, 2004.

BANERJEE, S. et al. Hands off the wheel in autonomous vehicles?: A systems perspective on over a million miles of field data. In: . [S.l.: s.n.], 2018.

CHEN, D. et al. Learning by cheating. In: **Conference on Robot Learning (CoRL)**. [S.l.: s.n.], 2019.

DOSOVITSKIY, A. et al. CARLA: An open urban driving simulator. In: **Proceedings of the 1st Annual Conference on Robot Learning**. [S.l.: s.n.], 2017. p. 1–16.

JAHODA, P.; CECH, J.; MATAS, J. Autonomous car chasing. In: **Proceedings of the European Conference on Computer Vision (ECCV) Workshops**. [S.l.: s.n.], 2020.

Koopman, P.; Wagner, M. Autonomous vehicle safety: An interdisciplinary challenge. **IEEE Intelligent Transportation Systems Magazine**, v. 9, n. 1, p. 90–96, 2017.

LEE, T. B. Waymo finally launches an actual public, driverless taxi service. **Ars Technica**, August, 2020. Available from Internet: <https://arstechnica.com/cars/2020/10/waymo-finally-launches-an-actual-public-driverless-taxi-service/>.

PHILLIPS, F.; OWENS, B. Domino's pizza tests delivery by autonomous vehicle. **JDSupra**, May, 2021. Available from Internet: <https://www.jdsupra.com/legalnews/domino-s-pizza-tests-delivery-by-4912435/>.

RAMEY, J. China's first robotaxi service launches in beijing ahead of the olympics. **Autoweek**, May, 2021. Available from Internet: <https://www.autoweek.com/news/technology/a36354137/beijing-olympics-robotaxi-china/>.

RONG, G. et al. Lgsvl simulator: A high fidelity simulator for autonomous driving. **arXiv preprint arXiv:2005.03778**, 2020.

SHAH, S. et al. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In: **Field and Service Robotics**. [s.n.], 2017. Available from Internet: <https://arxiv.org/abs/1705.05065>.

ANEX

Included here is a transcript of the prototype. For instructions on setting up the slightly altered simulator as well as an example illustrating how simple it is to inject faults using this proof-of-concept, please refer to <https://github.com/cdflion/carla_fault_injection>.

Listing 5.1 – End prototype with additional features

```
import carla
from enum import Enum

class Strategy(Enum):
    CONSTANT = 0
    INTERMITTENT = 1
    TRANSIENT = 2
    CRASH = 3

class FaultySensor(carla.Sensor):
    strategy = Strategy.CONSTANT
    target = 0
    counter = 0

    def __masterCallback(self, image, callback,
        faultCallback):
        faultyImage = faultCallback(image)
        callback(faultyImage)

    def __intermittentCallback(self, image, callback,
        faultCallback):
        self.counter = self.counter + 1
        if(self.counter >= self.target):
            self.counter = 0
            self.__masterCallback(image, callback,
                faultCallback) #called after interval
        else:
```

```
            callback(image)

    def __transientCallback(self, image, callback,
        faultCallback):
        if(self.counter >= self.target):
            self.__masterCallback(image, callback,
                faultCallback) #called constantly after time
        else:
            self.counter = self.counter + 1
            callback(image)

    def __crashCallback(self, image, callback,
        faultCallback):
        if(self.counter == self.target):
            self.counter = self.counter + 1
            self.__masterCallback(image, callback,
                faultCallback) #called once
        elif(self.counter < self.target):
            self.counter = self.counter + 1
            callback(image)

    def listen(self, callback, faultCallback):
        if(self.strategy == Strategy.CONSTANT):
            super().listen(lambda image: FaultySensor.
                __masterCallback(self, image, callback,
                faultCallback))
        elif(self.strategy == Strategy.INTERMITTENT):
            super().listen(lambda image: FaultySensor.
                __intermittentCallback(self, image, callback
                , faultCallback))
        elif(self.strategy == Strategy.TRANSIENT):
            super().listen(lambda image: FaultySensor.
                __transientCallback(self, image, callback,
                faultCallback))
```

```
        elif(self.strategy == Strategy.CRASH):
            super().listen(lambda image: FaultySensor.
                __crashCallback(self, image, callback,
                faultCallback))

def ToFaultySensor(sensor):
    sensor.__class__ = FaultySensor
    return sensor
```