UNIVERSITÀ
DEGLI STUDI
FIRENZE

Scuola di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Informatica

TITOLO IN ITALIANO

TITLE IN ENGLISH

TERROSI FRANCESCO

BONDAVALLI ANDREA
STRIGINI LORENZO

Anno Accademico 2018-2019

ABSTRACT

Abstract

CONTENTS

LIST OF TABLES

LIST OF FIGURES

1

INTRODUCTION

Nowadays, computer systems have a central role in our society. Originally intended to be computing tools for mathematicians, their evolution made them a central component of our society. Thanks to the progress in technology and in engineering techniques, these systems are now used for a multitude of purposes in very different domains: from the medical field, to the avionics, without forgetting about the relatively novel *"Internet of Things"*.

The new domains in which these systems are used, required a redefinition of the concept of *"computer systems"*. Many of these systems must respect hard deadlines to fullfill their tasks, or there may be serious consequences which could result even in fatalities.

Whenever the system must respect hard deadlines, we refer to them as *Critical Systems*. If a failure of such systems may be catastrophic, with serious damages to the environment, the infrastructures, or human beings, we talk about *Safety Critical Systems*.[24]

There are cases in which these systems perform in a physical environment. An example of such a system is an autonomous car: a system composed by a *computer system*, in charge of doing the computations needed to fulfill the assigned task, and a physical component, in charge of interacting with the environment by change both the states of it and of the system itself.

Because these systems are so pervasive and yet so dangerous, in case of a failure, to satisfy and guarantee their (usually) ultra-high *dependability* requirements is a key factor during the design and development phase.

The *dependability* of a System is a measure of how *"trustable"* a system is, i.e. its ability to provide a **correct service**.[23]

A *failure* is an event that causes a disruption of the provided service.

1.1 DEPENDABILITY AND SAFETY

The *dependability* of a critical system is defined as an ensemble of *qualitative* and *quantitative* measures. Some of the most important are listed here:

- *Availability*: a function that measures the alternation between correct and incorrect service

  $$A(t) = \begin{cases} 1 & if\ a\ correct\ service\ is\ provided\ at\ time\ t \\ 0 & otherwise \end{cases}$$

  $\mathbb{E}[A(t)]$, the expected value of the availibility, is the probability that the system is providing a correct service at time t

- *Reliability*: the ability of the system to provide a *continuous service*

  $R(t)$: probability of providing a correct service in the interval $[0, t)$

- *Safety*: the absence of catastrophic consequences if a failure occurrs.

- The *Safety* of a system is more often defined as the *Mean Time to the next Catastrophic Failure*, because the safety requirements require a quantitative measure (e.g. one failure every 10^n) years).

  $S(t)$: probability that **no failure** occurrs in the interval $[0, t)$

  $MTTF$: Mean time between the recovery from a failure and the next failure.

- *Maintenaibility*: the ability for easy maintenance and repair from a failure. This measure has an influence on the Availibility.

  $MTTR$: Mean Time to Repair the system from a failure

- *Coverage*: the measure of effectiveness of the system's fault-tolerance mechanisms, i.e. the ability of the system to prevent, avoid or correct failures

A *threat* is a menace to the system's dependability, that is an *"event"* that makes the system provide an incorrect service. Threats may be of different kinds and come from many sources, such as wrong specification or a wrong implementation of a specification, accidents... When considering the dependability of a system, we are interested in providing a *continuous correct service*. A *failure* is a transition from a state of *correct service* to a state of *incorrect service*. When designing a Critical System, the need to reduce the possible transitions (failures) from these two sets, guides the development process.

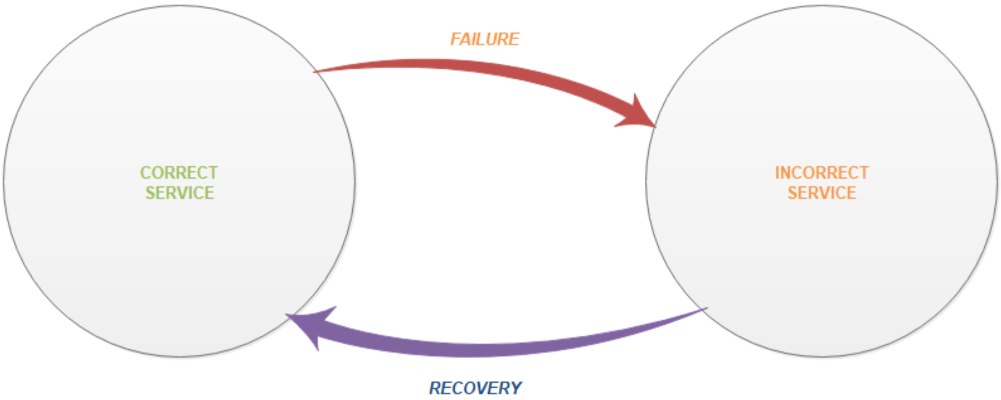

Figure 1

When it comes to *Safety-Critical Systems*, we must distinguish between *benign failures* and *catastrophic failures*. The first are failures that we can somehow accept: the system will not provid a correct service but it will still be in a safe state. The latter are the most dangerous kind of failures because an incorrect, unsafe service will potentially have catastrophic consequences such as damages to the environment, disruption of the system's infrastructure or even fatalities, in cases where these systems and humans work in close contact. As an example, think to an autonomous car. Imagine that the car is riding in "normal" conditions and suddenly an obstacle appers in front of it. A screeching halt and a consequent interruption of the ride is indeed a failed state, but it's considered as a *benign safe*, because no one got hurt. On the other hand, a situation in which the car beging to throttle towards the obstacle and eventually collides with it, is defined as an *unsafe failure* because people in the car could get seriously injuried (or killed).

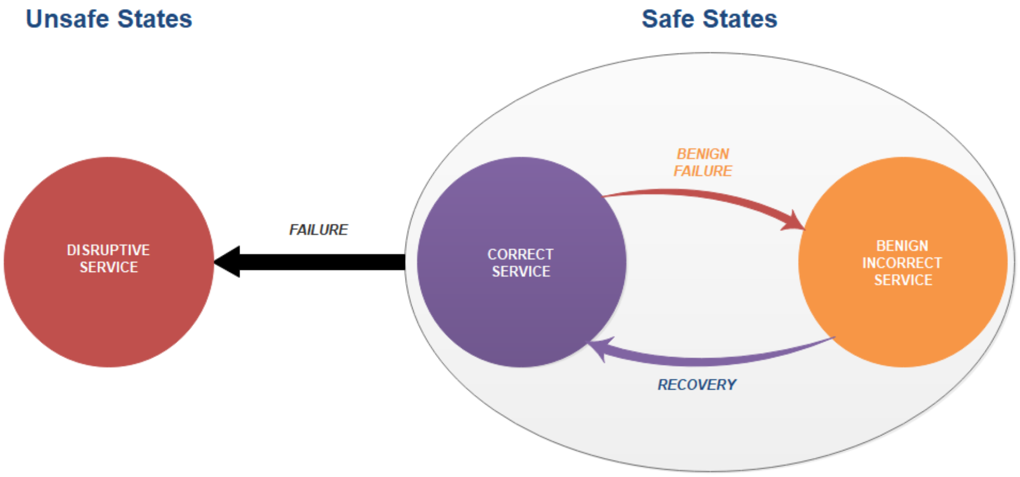

Figure 2

To study the dependability of Safety-Critical Systems, we want to find all the possible failure causes. To do so, the academics agreed on the *fault-error-failure chain* model, which is widely accepted both by academics and practitioners:

- *Fault*: the *adjudjed or hypothesized* cause of an error
- *Error*: that part of the system state that may cause a subsequent failure
- *Failure*: the situation in which the *error* reaches the service interface, altering the service of the whole system

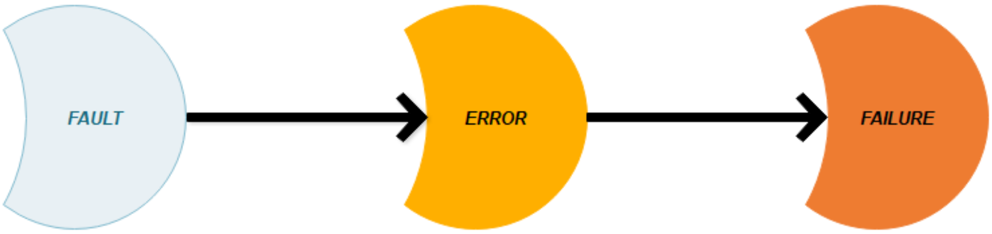

Figure 3

The dependability of a system comes from a set of four techniques that aims to prevent/mitigate the consequences of possible failures:

- *Fault Prevention*: means to prevent the occurence or introduction of failures
- *Fault Tolerance*: mechanisms to tolerate faults. In case of fault the system is still able to provide a correct service
- *Fault Removal*: reduction of the *number* or *severity* of faults in the system
- *Fault Forecasting*: use of statistical techniques to estimate the present number, the future incidence and the likely consequences of faults

The effectiveness of the measures adopted to achieve the dependability requirements of a system is measured through the *validation* process. The validation of the system requirements is a process that must be repeated in each phase of the system development, even at the very beginning of the design phase.

There are multiple validation techniques that may be specifically selected for each phase of the system development: [25]

- *Analytical/Numerical Modeling*
  - Techniques that models the system capabilities using numerical models that have a closed form solution, i.e. the changes in a system can be described as mathematical analytic function. Examples of such models are the *Combinatorial* and the *State-Based* model
- *Simulation*
  - Empirical estimation of the system dependability, deployed in a simulated environment. This method allows to inject faults, to check whether a specific fault-tolerance mechanism works or not
- *Measurement*
  - When a prototype of the system is available, its executions can be observed and the measures of interest computed

It is important to keep in mind that these techniques are non-exclusive and the validation process should be a combination of these techniques. As said before, the process of validation should be done during the *whole* lifetime of a system, starting from the design phase and continuing after the deployment. Some of the methods listed above are more suitable than others in certain phases:

- *Specification*: validation is done by the specification of the dependability requirements, that can be validated using Analytical/Numerical techniques, such as the *Combinatorial Models* to determine the failure conditions for the system's components. Failures are considered as independent
- *Design*: during the design phase, it's reasonable to model the state-space of the system using *State-Based Models*. Examples of such models are *Markov Chain*, *Petri Net* (and its evolutions)...
- *Implementation*: when the development is in an advance phase, it may be possible to build a prototype of the system that can be directly observed to measure the effectiveness of the fault-tolerance mechanisms on the system dependability
- *Operation*: once the system is deployed, it is possible to observe how it performs in a real environment

1.2 SYSTEM MONITORING

The observation of a system operating in its environment to collect measures and evidences about its properties, is a technique called *System Monitoring*. Nowadays it's considered a good way to measure a system's dependability and diverse approaches on how it has to be done were proposed in literature. In this work we stick with the methoda described in these works [26], [27], [28]. This technique aims at constantly monitoring the system running in its *final* environment, verifying that the *observed behaviour* and *performance* meet the defined requirements. The data collected in the monitoring activity *must* be validated:

- *Offline*: data are collected and stored somewhere, while the system is running, and are analyzed afterwards
- *Online (or at Runtime)*: data are analyzed *while* they are being collected

A good monitoring technique must consider all these aspects:

- *Identification* of the relevant events, measures and features of the system, that are necessary to assess the system dependability
- *Labeling* data in order to enrich the raw measurement with more informations

- *Transmission* of the data collected to the analysis node, in order to process them
- *Filtering* and *Classification* of data with respect to the measures of interest

In the definition of a monitoring activity, we call *Target System* the whole system as it is. If the monitoring activity is related to a specific hardware or software component of the system, we refer to it as *Target Component* or *Target Application*. Both practitioners and academics agree on the effectiveness of these two models, very different from each other:

- *Black-Box approach*
- *White-Box approach*

The choice on which approach to use depends on how much control it's possible to have on the target system, especially on its internal implementation. If e.g. the details implementation are unknown because the monitoring activity is performed on a third-party system, only a black box approach can be adopted: after the definition of a *workload* (i.e. an input or a series of inputs, representative of a possible execution), the workload is given to the system and its outputs are observed.

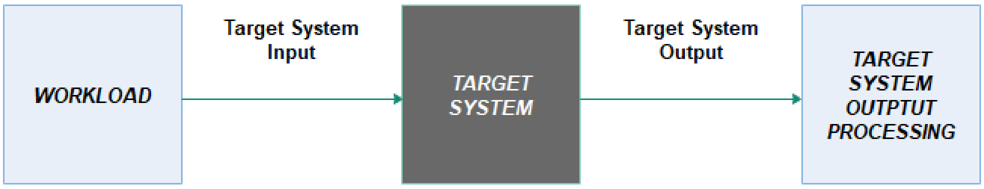

Figure 4

If the internal details of the Target System are known and freely accessible, it's possible to monitor the system "from the inside", attaching *probes* to the system in order to observe the intermediate outputs produced by the system *during* its execution. These probes, since they are attached directly to the system's internal components, can give much more informations than the one collectable just by observing the system's outputs. If this approach provides a lot of more informations on the system's behaviour on one hand, on the other hand it requires extra care on how the monitoring activity and system probing are done. In particular, it's mandatory to adhere these two basic rules:

- *Representativeness of Selections*: the probes must be able to observe an *adequate number of meaningful observations* to perform a good monitoring activity
- *No Intrusiveness*: the system's behaviour *must not* be affected by probing, otherwise the measures will be meaningless

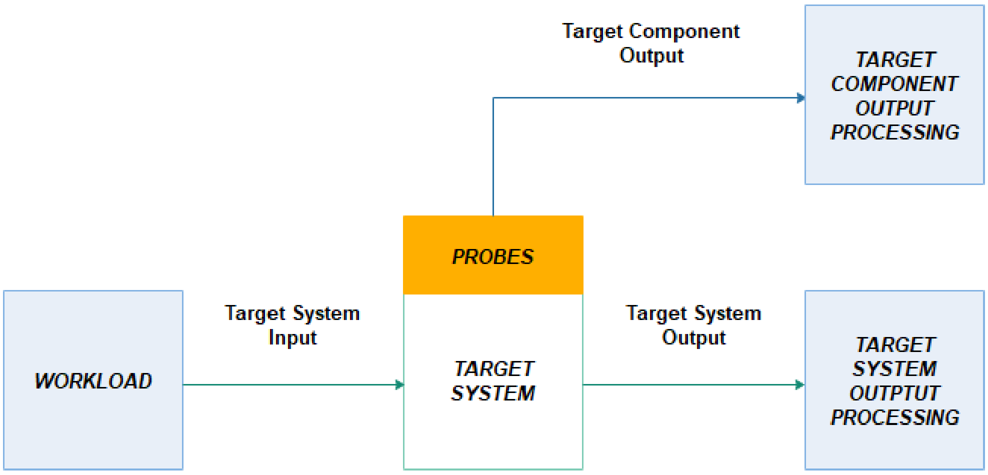

Figure 5

The design and development of the monitoring system requires additional reasoning, especially on how the probing is done. In particular, a monitoring activity can be:

- *Hardware Monitoring*
- *Software Monitoring*
- *Hybrid Monitoring*

A dedicated hardware circuit for monitoring is the best way to monitor a system because of the very low intrusiveness (outputs are read "on the fly" while being produced). However, systems are becoming more and more complex nowadays, making it very difficult, if not impossible, to install hardware probes. Software probes are very powerful instruments, because they have access to more informations than the ones available to hardware probes, because it's possible to know the *context* in which the specific output was produced. Software probing is done by adding instruction in the *Target Process*, specifically dedicated to data extraction and measurement, in the *Operative System* or by developing a new *probe process*. Hardware and software probes can be combined in a hybrid approach, with particular care on mitigating the cons of each technique.

1.3 MOTIVATION AND GOAL OF THIS WORK

In this work we are interested in monitoring a self-driving car composed by a Neural Network, in charge of driving the car, and a System Supervisor, a component that provides fault-tolerance mechanisms to avoid catastrophic failures.

Autonomous cars are one of the most recent and promising safety-critical systems in terms of the implications they may have on future system developments.

These kinds of system usually have ultra-high dependability requirements that are very hard to validate. Moreover, the nature of the software architecture, involving AIs and non-AI software interacting with each other.

The goal of this work is to develop an experimental method to assess the dependability of such complex systems, with particular care for the problem of *what* measures are relevant for these systems and *what* are the issues that have an impact on the analysis activity. An experimental activity was conducted in a realistic simulated environment to demonstrate the concepts proposed in this thesis.

We didn't have a trained Neural Network available, nor a System Supervisor. The network was trained from scratch and a simple System Supervisor was developed as part of this work.

It's important to notice that this work doesn't aim at providing an *exhaustive* treaty on the argument, but at begin a phase of exploration of these concepts and problems, that will require further validation and exploration in future works.

2

AUTOMOTIVE - STATE OF ART

Self driving cars are one of the hottest topics of the decade. Artificial Intelligences specifically trained to drive with machine learning techniques demonstrated that it's possible for a computer to drive cars. However, a failure in these kind of systems may have very serious consequences that could result in people being injuried, or killed. At the same time, it is a problem to certify the ultra-high dependability requirements of these systems. In this chapter, today's problems regarding the safety issues related to self driving cars are reviewed, for that it was decided to conduct this study.

2.1 AUTONOMOUS CARS AS CPS

In order for a car to be able to drive by itself, suitable hardware and software are required. This makes autonomous cars cyber physical systems, and the possible catastrophic consequences that a failure in/of these systems can cause, make them fall under the set of critical systems.

To sense and map the surrounding environment, the system collects data from multiple sensors. Some of the most important sensors and their purposes are listed here:

- GPS
  → High precision GPS sensors are used to estimate the exact position on the vehicle in the world
- Odometry & IMU sensors
  → These sensors are worth for detecting changes in the position of the car and of the objects in the environment over time
- Cameras

→ Cameras are literally the *eyes* of the system. Images captured are usually processed with image recognition software

- Lidars & Radars

→ Lidars can be seen as the evolution of conventional radars. Data combined from these sensors serve the purpose of mapping the environment and detect obstacles and objects around the car

Outputs from these sensors are combined and given to the car's control system. An abstraction of the software architecture is shown in this figure:

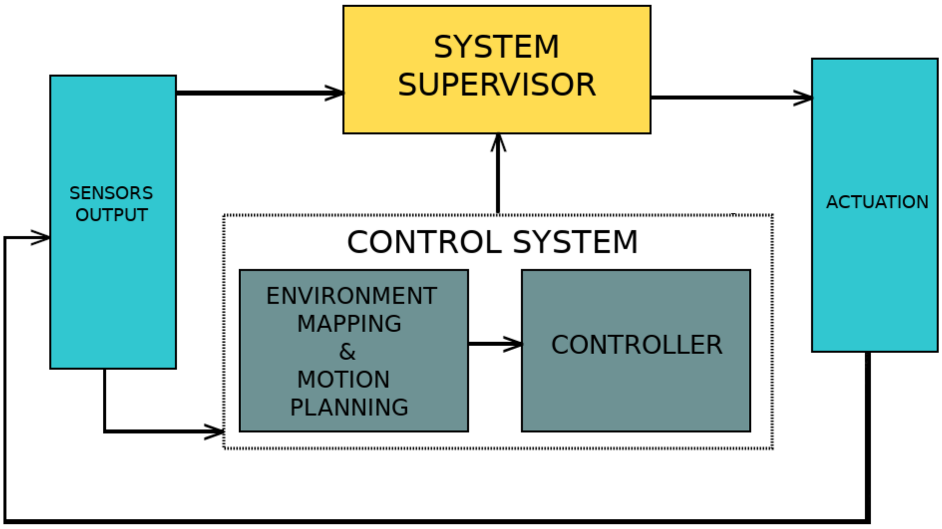

Figure 6: High-level abstraction of the system's software architecture

Data from sensors are inputs of the control system, here simplified as compose by two constituent systems: one in charge of collecting data directly from the sensors, process them in order to build an *occupancy grid*[1] to map the surrounding area and to create a physical model of the environment in order to follow the correct route to the destination without crashing. The Controller (usually composed of a Velocity Controller[2]

[1] A matrix mapping the environment, the cells $a_{i,j}$ are flagged with 0 if there's no object at coordinates i, j, 1 if occupied.

[2] Controller in charge of adjusting the vehicle's speed

and a Steering Controller[3] uses these data to adjust the values on the actuator controlling the movements of the car: throttle, brake and steer. Due to the criticity of their task, it's mandatory to have a System Supervisor, a System in charge of detecting possible hardware failures or wrong outputs[4] from the Control System and, if needed, activate a corrective routine.

The System Supervisor is the main failure avoidance component of such systems. Of course there may be specific checks when data are processed, but the last decision is up to this system's monitor and the underestimation of its importance can lead to dramatic consequences, such as the 2018 accident in Arizona, where a woman was killed by a self-driving car.[3] Further inspections showed that the car's radar and lidar data detected the victim almost 6 seconds before the impact and it took 4 seconds circa to infer that there was an obstacle on the road and that an emergency brake was needed. However, this safety-checker was disabled during tests for "smoother rides", causing the accident.[5]

The extreme complexity of these systems raise concerns among the experts: the way te system's safety is studied and tested must be looked from a new point of view and to sensitise about safety culture.[6]

2.2 SAFETY AND AUTONOMOUS VEHICLES

According to a SAE International tentative to classify self-driving cars' autonomy, the level of automation can be divided in 6 tiers, ranging form 0 to 5. Level 0 means no autonomy: a human driver just drives the car, level 5 means that there's no need of human intervention at all and the car is not only capable of driving safely on the road, but it must be able to avoid catastrophic failures that may seriously harm (or kill) people. The more autonomous the car is, the higher the dependability requirements are for it to be put on public roads. It is well known that demonstrating a system's dependability is not an easy task for itself, it gets even harder with ultra-high dependability systems such as these are. In adition to the problem itself, demonstrating autonomous cars' dependability has two more problems to deal with: how to safely and effectively test the system and the need for neural networks to achieve the task.

[3] Controller that determines the steering angle

[4] With *"wrong"* is intended not only outputs out of the domain space but also outputs that would cause the system to fail (e.g. causing a crash)

Lots of studies demonstrated that it's unthinkable to just test cars on the roads. One of these, that we will refer to as the RAND Study, answers the question of how many mils of driving would it take to demonstrate autonomous vehicles' reliability using classical statistical inference, saying that if autonomous cars fatality rate was 20% lower than humans', it would take more than 500 years with *"a fleet of 100 autonomous vehicles being test-driven 24 hours a day, 365 days a year at an average speed of 25 miles per hour"*.[2]
The validation of ultra-high dependability requirements for safety-critical systems is a well known problem in safety literature and has not been introduced by the advent of autonomous cars. In fact, the RAND study is nothing but a specific case of the problem considered in a work published in 1993 by Littlewood & Strigini, in which the same concepts are discussed and generalized for every ultra-high dependability system.[11]
The main problem with the RAND study approach is that future failures frequency can not be predicted just using the observed one. Not just for the quantitative results of the impossibility of it, but also because of this approach can not work: an observed frequency failure of 0 would lead to optimistic (and possibly harmful) predictions. Luckily, this problem is surmountable, as shown in *this*[10] work by Zhao et al.

Validating the dependability requirements of an autonomous car seems a hard task already. Things are made even harder by the fact that these cars are driven by neural networks.
In these years there is a huge interest in the *machine learning* sector, and this has made that a lot of progress was done in the research. It's also thanks to these progresses that autonomous cars now seem like something we can achieve, since these AIs gave surprising results with their skills and big names such as *Uber* and *Tesla* are putting more and more efforts in AI research. This new wave of AI research is deeply changing the way we interact with computer systems, and surprising results were achieved with neural networks.
These tools have proven to outclass *"classic"* software solutions (intended as non-neural network) in a lot of tasks, ranging from *Object Detection*[12] to *Gaming*[13] problems, performing even better than humans.[14] The complexity of the environment in which the system performs and the need of quick decision-making procedures and fast responses to events that *cannot* be planned with *"classic"* software, make neural networks the perfect tool to achieve the task of a car being able to drive by itself, thanks to their ability to handle multiple situations that were not explic-

itly written in the software. However, this raises serious issues about the safety of the whole system for many reasons.

If neural networks gave promising results on one hand, and they seem the only way to achieve goals such as autonomous cars, on the other hand it has been shown many times how weird a network's prediction can get when *minimally* perturbating the inputs[16] and how high the confidence interval can be.[15] The lack of official regulations and certifications of these kinds of software, as well as the need to truly understand neural networks, is raising concerns on how dependable these systems can be and consciousness is now growing on the topic, asking for more regulations on companies developing advanced AIs.[4]

2.3 CONTROLLER - CHECKER PROBLEM

The interaction between the Control System and the System Supervisor is at the core of the car's movements. The Control System, or *Primary Component*, is the software performing the main computations of the system, required to drive the car. In a context like this, it's mandatory to have fault-tolerance mechanisms such as the System Supervisor, to avoid catastrophic failures. This kind of architecture is a must for these systems, due to the extremely high dependability requirements they have, in order to try to cover all the possible failures that may happen. The state space of such systems may be sketched as in figure 7.

We consider *safe states* all the states in which the Control System produces an output that would not result in a crash.

Imagine that an autonomous car is riding when suddenly an obstacle appears. If the Primary correctly detects the obstacles it should apply a safety-measure to **avoid** a transition in an *alert state* (e.g. a safety-brake). If the controller doesn't see *or* detects the obstacle but keeps throttling, there is a transition from a *safe-state* to an *alert-state*, in which the failure-prevention components turn in. The System Supervisor's duty is now to launch a corrective-routine that will put the system in a fail-safe state (e.g. by applying the safety-brake not done by the controller and turning off the engine). An error of the Supervisor will inevitably cause the system to fail, as a result of the failure of both components, leading in a failure state (the crash happened). If we model the system's failures in this way, the level of safety of the system can be represented as the union of the failure area covered by the Controller and the one covered by the Supervisor

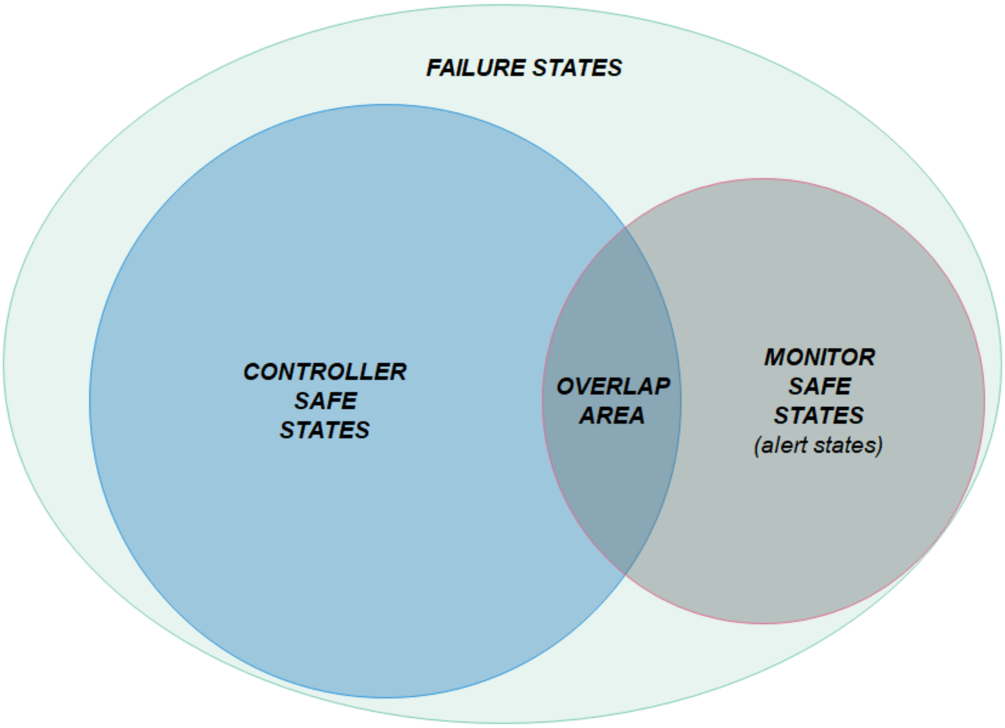

Figure 7: Sketch of the system's safe states

This problem is nothing but a generalization (related to safety) of the asymmetric fault-tolerant architecture for computer systems: the idea of having a *Primary Component* that does the main computations, and a *Primary Checker* in charge of detecting (and correcting) error of the Primary.
The problem of assessing the dependability of these simpler (but still complex) systems is a well known topic in literature and was explored in different studies. In a relatively recent work published by *Popov* and *Strigini* in 2010, it is shown that the probability of a system failing on a specific input (or set of inputs), strictly depends on both the coverage of the Primary *and* the Primary Checker, as shown in the figure above.[1]

In the context of self-driving cars we want the area covered by the primary to be the largest possible. This is done by intensive training of the neural networks that will control the car. As long as the network is trained *"properly"*, the control system should be able to handle most of the dangerous situations that may happen. At one point, it is possible that the areas covered by the Controller and the Supervisor will eventually overlap, reducing the overall contribution to the system's safety given by the latter.

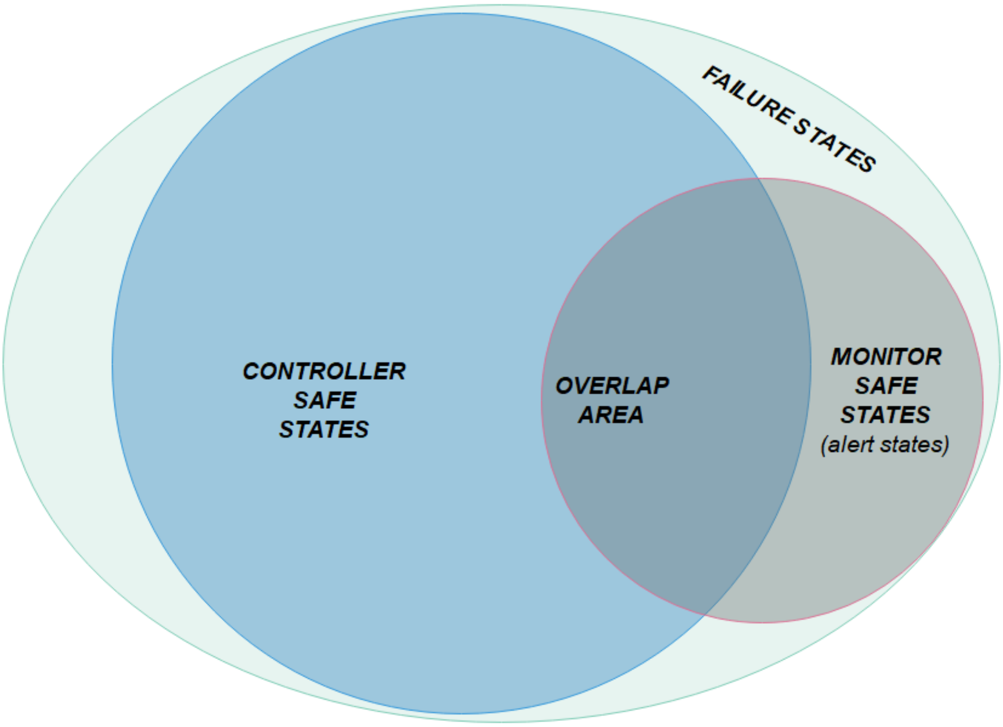

Figure 8: The states covered by the Controller are now the ones previously covered, plus some states previously covered just by the Monitor

Another possibility is that during the training, a portion of the failure area covered by the controlller becomes uncovered. This could result in a situation in which some of the previously safe states are now alarm state, representing a serious harm to the system's dependability. Since the coverage area provided by the System Supervisor can not change without changing its implementation (it doesn't "learn" automatically), a transition to one of these states would now inevitably result in a failure.

All these considerations and the lack of literature on the topic for these new systems such as autonomous cars, lead us to begin a study on the emergent behaviour resulting from the interaction of a neural network control system and a *"classic"* error checker, what happens when the network is *taught* by a supervisor during the training and how performance metrics of these 2 components can be computed.

In the next sections we present and discuss the development and implementation of an experimental methodology to study these aspects.

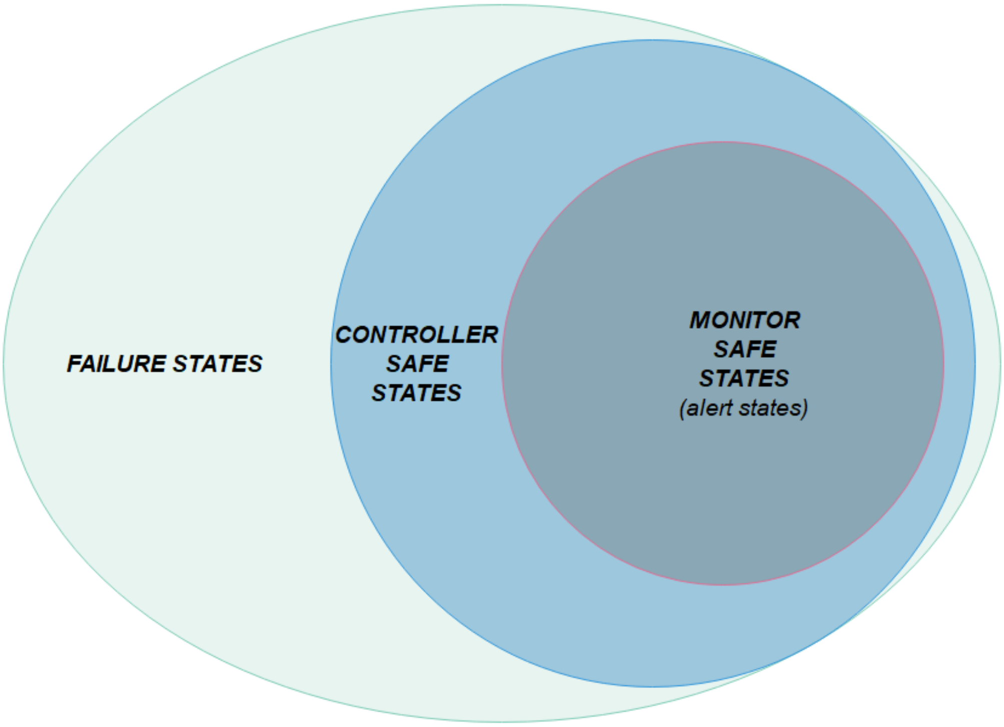

Figure 9: The Controller now covers all the states previously covered by the Monitor, but doesn't cover anymore some of the states he covered before the training

3

SYSTEM ANALYSIS METHOD

3.1 INTRODUCTION AND PRELIMINARIES

The goal of this study is to develop a first experimental methodology intended to observe emergence-related aspects coming from the interaction of a Control System and a System Supervisor, analyzing the system in a *simulated* environment.
The software architecture of an AV was simplified in two main components:

- A *Controller*: a Neural Network trained with *reinforcement learning* algorithms to drive the car
- A *Safety Monitor*, a submodule of the System Supervisor, that checks whether the car is going too fast towards an object, processing data received from a LiDAR sensor, and if that's so, apply an emergency-brake

This work focuses on exploring the topic from a different point of view and to assess its feasibility in an experimental, simulated environment. Due to the system being composed by two constituent systems: the Controller and the Monitor, we think that a point of view based on the emergent behaviour resulting from the interaction of these systems can improve the quality of the assessment.

In this chapter is presented and discussed a method to study the safety level of an autonomous car over time, observing the emergence resulting from the interactions of a neural network controller and a safety monitor in a simulated environment.

The proposed framework is designed with particular attention on studying the emergence resulting from the interaction of these two consitutent systems.

The main aspects we are interested in this first stage of exploration are:

- How the effectiveness of a monitor evolves when the neural network is learning
- Effects of training strategies on the effectiveness of a safety monitor

One of the most appealing features of neural networks is that they can be *trained* on data sets to improve their performance. One stage of training is done by collecting data over *n* steps and updating the weights of the prediction function. The weights of the function after i training stages represents the *state* of the network at **epoch i**.
A neural network will likely give good results after "enough" epochs. The harder the task, the more epochs are needed. Driving a car is a quite hard task and it's inimmaginable to save the weights of each epoch. Therefore, given a neural network N, we define a **checkpoint** as a generic epoch of N. Say we trained N for 1000 epochs. If we save the weights of the prediction function every 100 epochs, we will end up with 10 checkpoints:

$$checkpoint_1 < checkpoint_2 < \ldots < checkpoint_{10}$$

where $checkpoint_1$ contains the network's weights at epoch 100, $checkpoint_2$ at epoch 200 and so on.

Let's now consider a self-driving car that is being tested on the road (either real, or simulated). Its task is to ride the car the longest possible, without crashing. During the ride, the environment surrounding the car will change as it procedes in its run. It may happen that in some of the system's state, the probability of a subsequent crash becomes very high, like a pedestrian suddenly crossing the road, if and only if the action taken by the Controller would result in the pedestrian getting hit we will address it as a failure (of the controller). The same reasoning applies if the pedestrian is actually detected, and the car hits something else while trying to prevent it:

- If the Controller takes *any* action that would result in a crash, it's considered failed
- If a hazardous event happens, i.e. a situation in which the probability of observing a crash is higher than usual, the Controller is considered failed if and only if its actions will not avoid the imminent failure

In this sense, at this stage of the work, we don't distinguish between changes in the environment that raises the probability of a crash (e.g. a pedestrian crossing the street) and hazardous actions take by the Controller (e.g. a sudden steer towards a wall).

If the controller fails in the way just described, it's the Monitor's duty to run a safety-routine in order to prevent the imminent failure.

In the case of a failure of the Controller, the Monitor not only needs to detect whether it failed or not, but it also must run a safety-routine to prevent a failure of the whole system. In this first phase of analysis, we consider the action taken by the Monitor to be always safe. This means that:

- If the Monitor executes **all** the steps in the safety-routine, the system will be in a safe-state.
- A failure of the Safety Monitor may be one of the following:
    1) The obstacle is not detected
    2) The obstacle is detected but the routine fails to terminate its execution (i.e. the detection was too late)

We consider the system failed if and only if both the Controller and the Safety-Monitor failed, as described above, leading to a crash. With this scheme in mind, the states space for this system can be modeled in this way:

As it's shown in this figure, we can classify the states of this system in 3 sets:

- Safe States: states in which the Controller doesn' need an intervention of the Monitor
- Alert States: states in which the Monitor is required to intervene. A correct detection (and prevention) of the potential accident would result in a transition to the safe states space again
- Failure States: states in which an accident happened. It is important to notice that **not** all the situations can be detected by the Monitor. There are accidents that can not be prevented at all, in which no Monitor could save the system. These states are represented by the red area on the right

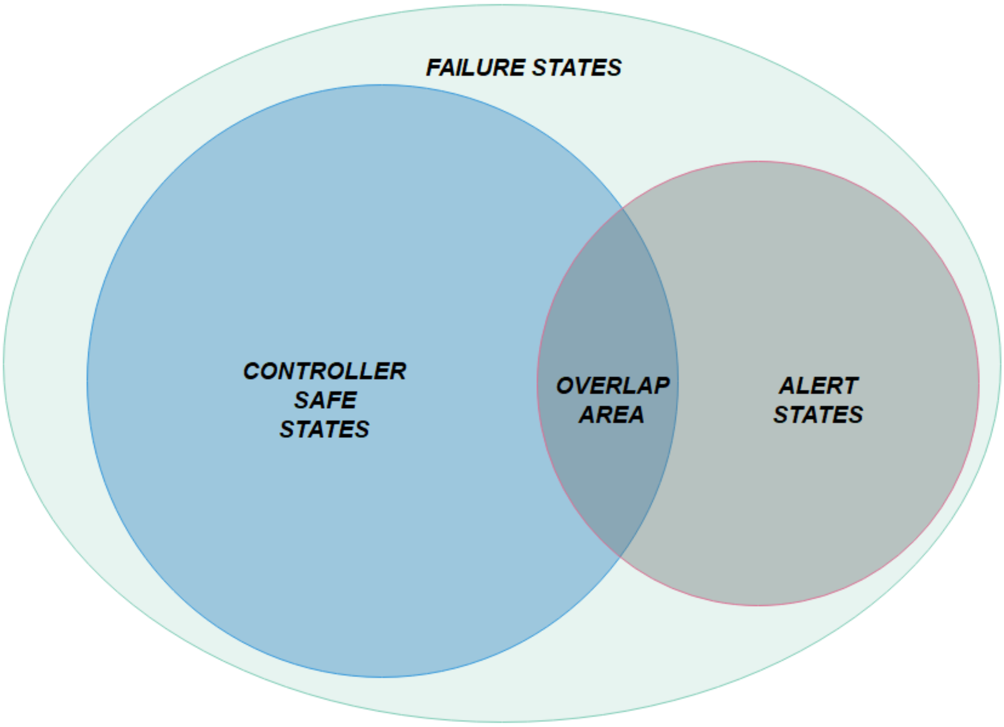

Figure 10: Representation of the system's states space

At the system level, we are interested in observing the probability of having a failure (crash) and how to minimize it. At the same time we want to observe how the effectiveness of the Safety Monitor changes when the Controller is trained over time.

In order to achieve this, c checkpoints are saved for later testing and comparison. This is useful not only for checking that the network is improving during the training, but also to have a better understanding on how useful is the Monitor when the Controller becomes more and more expert. This is mandatory for the experimental activity as different checkpoints of the same network are tested under the same condition, to observe how the behaviour of the Controller changes in the first stages of the training.

The need for multiple checkpoints is mandatory not only to test that the network is improving, but also to observe how the monitor's effectiveness change over time. Moreover, if many checkpoints of the same network are tested under the same scenarios, it is possible to extract interesting measures, as we'll see in the next section.

Before the analysis can start, n_h scenarios must be defined. A *Scenario* is a set of initial conditions (e.g. the spawn point of the car, seeds used in random number generators...) in which the car is intended to be tested. The pedix h represents the difficulty level for the specific case. The purpose of this is that we are interested in testing the car under the same initial condition but for one factor, in order to have a better understanding on what makes the system fail more often. The h variations should be developed with growing difficulty and at the same time they must be realistic. Given a Scenario S, examples of variations may be: to increase the number of cars in the scenario, or to simulate adversal weather conditions. A combination of these 2 variations should result in a *harder* variation of the scenario[1]. It's important to keep in mind that these scenarios, once defined, should not be changed as they will be used for testing all the checkpoints. A change in the settings of a scenario (except for variations) should be considered as a new one.

We hope that the results collected here will help in the process of understanding what makes a situation *"harder"* than others for such systems.

3.2 EXPERIMENTAL METHOD

The approach used for this exploratory work is divided in 3 phases:

- Phase 1: Given c checkpoints of a neural network and n_h scenarios, the Controller is tested in all the scenarios and its runs are recorded
- Phase 2: The Monitor is tested by *repeating* the runs recorded in phase 1, attaching the Safety Monitor to the system
- Phase 3: The network is retrained from the last checkpoint recorded, using different strategies to improve its performances. The new networks obtained and the (same) Safety Monitor are then retested in all the n_h scenarios

In the first phase we are interested in assessing the goodness of the neural network and of the Monitor. This is done in 2 different steps.

In the first step, the c checkpoints of the Controller are tested in all the scenarios. In this step we are mostly interested in observing how the *Reliability* of the Controller changes with respect to these checkpoints.

One of the main problems when testing a neural network is that of *repeatability*. It is very unlikely that the **same** neural network, under the

[1] It is important to point that what we think to be "harder", may in reality be easier to handle for the car.

same initial conditions, will behave in the same manner in multiple runs. Due to this property of networks, it may happen that the failure mode observed in one of the runs of the scenario s_i will never happen again, or the time needed to make it happen again may be very long. How the scenarios and their variations are created is fundamental to observe specific failures, however it's impossible to think to *all* the possible failure situations that may happen, for this reason we think that the scenario-variation approach may help in solving this issue, creating harder operational situations in which the factors that lead to a crash may be studied.

The repeatibility issue was solved by creating a *black box* for each run of the scenarios developed for testing. This approach is used to keep a trace of the Controller's actions, so that the specific run may be studied more fully to understand what made tha Controller fail. These data can be then used to better study what hazardous situations are covered by the Controller at a specific checkpoint j, and if these situations are still covered when the network is tested at the checkpoint $j + x$.

3.2.1 *Controller Testing*

The Controller is tested in isolation in each scenario, for each difficulty level, until a crash occurs. The situation in which no failure is recorded is still a possibility (even if the difficulty level somehow mitigates this issue). A reasonable alting criteria is out of the scope of this work and is still a problem in the academic community, however, as noted in the previous section, this problem may be solved.[10]

The reasoning behind the choice of isolating the in order to test it Controller, comes from some of the problems issued during the method development phase. The main problems are the *repeatibility* and *non-intrusiveness*. As pointed before, the *repeatibility* issue for the neural network is solved by creating a *black box* containing informations about the car's state in each frame. At the same time we can not think about testing the whole system at once (Controller *and* Monitor) because a safety-brake executed by the monitor will most likely change the environmental conditions for the rest of the simulation and we would not be able to compute measures about the goodness of the Controller itself. Testing the Controller in isolation helps to solve these issues and it's preparatory to the second phase.

Recording the actions taken by the Controller for *each* frame (as well as other measures such as the vehicle's speed in that frame) make it possible to have a great control over the data, as a single run may be repeated many times to gather additional data if needed.

When the controller has been tested for each difficulty, in all the scenarios at least the followings must be computed:

- $\mathsf{MDBF}_{i,j} = \frac{\#\,\mathrm{of\ faults}}{\mathrm{meters\ travelled}}$
  - Mean Distance Between Failure for the i^{th} checkpoint, at the j^{th} level of difficulty
- $\mathsf{MTBF}_{i,j} = \frac{\#\,\mathrm{of\ faults}}{\mathrm{operational\ time}}$
  - Mean Time Between Failure for the i^{th} checkpoint, at the j^{th} level of difficulty
- $\mathsf{FR}_{i,j} = \frac{1}{\mathsf{MTBF}_{i,j}}$
  - Failure Rate of the i^{th} checkpoint at the j^{th} level of difficulty
- $R_{i,j}(t) = e^{-FR_{i,j} \cdot t}$
  - Reliability Function of the i^{th} checkpoint at the j^{th} level of difficulty, i.e. the probability that the Controller C_i is not failed at time t when operating at difficult j

When measuring the reliability function $R(t)$ of a system, one of the main measure of interest is its Mean Time To Failure, because it is easy to compute in simulated environments, and it's used to compute the rate λ of the exponential function. In the Automotive Sector however, data are usually computed with respect to the travelled distance, i.e. Mean Distance to Failure, rate of crashes per kilometers, and so on. With our approach, if the simulated environment and the hardware running the simulations are powerful enough to run the simulations at a fixed time-step, it's very easy to switch the point of view on the data.

As long as the neural network is trained properly, we expect the following disequation to hold: $R_i(t) \leqslant R_j(t)$, where i and j are 2 checkpoints, with $i < j$. This can be easily verified using the approach defined above to collect the data.

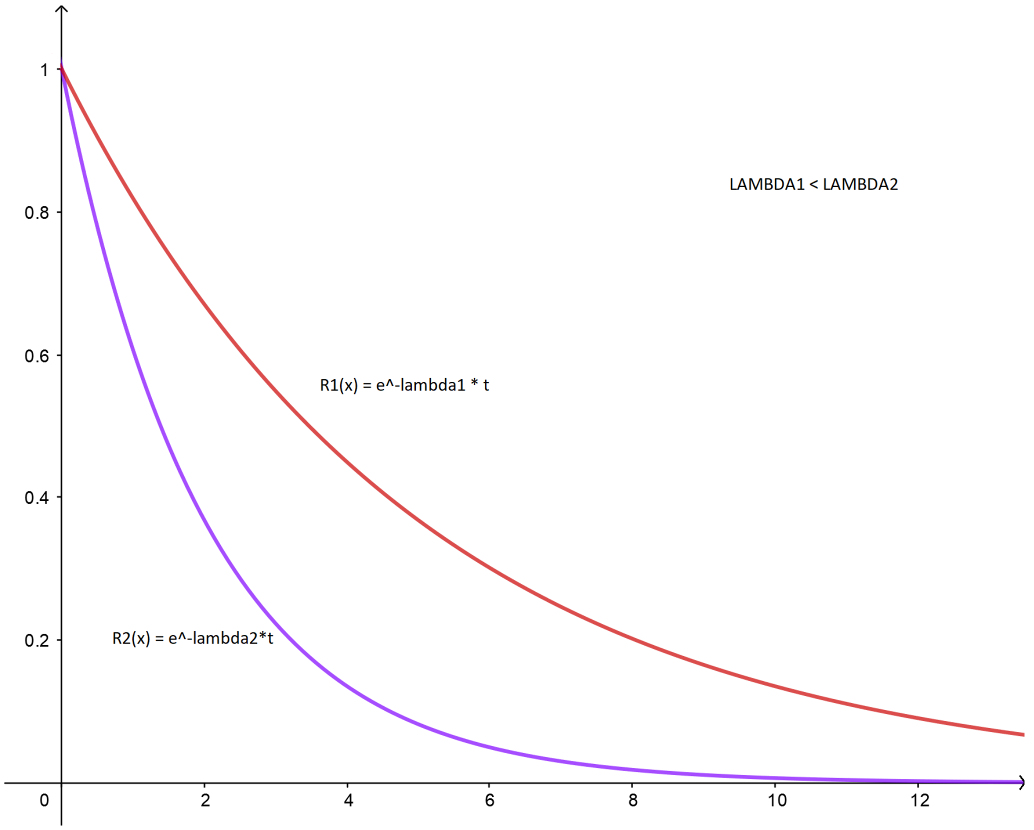

Figure 11: The Reliability function measures the probability that at time t the system is still operating. We expect that more expert drivers (i.e. more trained networks) are capable of longer runs than little trained networks

If the Simulator used for testing permits it, other data should be recorded too, in order to enhance the comprehension of the Controller's behaviour, such as:

- The car's instantaneous speed and acceleration vector at each frame
- With *what* the car crashed (e.g. a vehicle, a pedestrian, a generic obstacle. . .)
- Environmental conditions at time $t - x$, if a crash occurred at time t

These data are fundamental to distinguish between *"safe"* and *"catastrophic"* failures. For example, if a fence is hit at a speed, let's say, less than 10km/h, it may be flagged as a less serious crash than hitting a pedestrian at the same speed.

As the Controller learns, we expect the *Reliability Function* and the *MTBF/MDBF* to increase, since a trained driver should in principle cause less crashes than a non-expert one, therefore increasing the value of the MTBF and possibly the y values of $R(t)$.

If the *black box* makes use of enhanced data (such as the ones listed above), it is also possible to observe changes in the car's behaviour with respect to the scenario and the difficulty level. For example: if we define a difficulty k by doubling the number of cars in the scenarios, it may happen that it is observed an increasement on the amount of collisions against other obstacles. If that's so, the simulations should be studied more deeply to enforce the training strategy in a certain direction, as this may mean that the Controller is going to crash on walls while trying to avoid other vehicles.

3.2.2 *Monitor Testing*

Once the Controller's runs are recorded as described previously, the Monitor testing can start.

The goal of this phase is not only to see how good the Safety Monitor is in preventing crashes, we are also interested in observing evidences about which situations are *"hard"* for the Monitor, and which for the Controller.

As the network becomes more expert, it may happen that its behaviour evolves in a way such that the Monitor is no more able to detect imminent failures, because the Controller is now good enough to cover all the failures previously covered by the Monitor, and the hazards caused by its novel behaviour are such that the Monitor is not able to detect them.

However, the opposite is also a concrete possibility. For example, in the first epochs the Controller may drive in a "crazy" manner, e.g. with a lot of sudden, high-angle steerings and riding at high speed. The Monitor will obviously have more problems in predicting what the next state will be due to the unpredictability of the Controller's behaviour. The more the network learns, the more it will (hopefully) ride smoothly, making it easier for the Monitor to detect possible, imminent crashes.

The main problem that should be addressed here is that the Monitor effectiveness may decrease, while the network is learning, resulting in a useless component that may even be detrimental to the system's *performability*: the Controller may become good enough to be able to cover all the hazardous events covered by the Monitor at an earlier checkpoint. This may not change the whole safety of the system, if we consider the actions taken by the Monitor as inconditionally safe, but it would result in lesser smooth rides, due to the safety-brakes applied by the Safety Monitor.

The Monitor is tested as follows: the black boxes create during the Controller testing phase are used to *repeat* the runs. The initial conditions must be *identical* as well and should be saved as part of the black box in the previous stage. The runs of the Controller previously recorded are now repeated, attaching the Monitor to the System and recording the alarm raised during the run and if it was able to prevent the crash that originally occurred.

In this stage of the analysis, the major concern was that of *non-intrusiveness*. For the reasons pointed above, we can not think about rerunning the simulations just by attaching the Safety Monitor and watch how it goes. The Safety Monitor, by definition, *overwrites* the Controller's action if an alert is raised, changing the next part of the run. This is theoretically not a problem, if one can distinguish between false and true positives. However, we can not know in advance what a false positive will be without creating a software capable of sensing an mapping the environment. But this is a Safety Monitor itself and, as every tool of this kind, it will have false positives even if extremely low, therefore this approach can not solve the problem.

The idea is to record the alerts generated during the run, without enabling the safety-procedure (e.g. the brake). Since we are going to repeat exactly the runs recorded in phase 1, it's possible to know the instant of time t in which the transition to the *alert state* before the crash occurred. With this information, all the alerts raised before t are indeed false positives, or false alarms, since we know t. Enabling the Monitor to activate the safety-procedure after this instant of time, makes possible to approximate its coverage.

Recall that what we call a *"failure"* of the Controller is a transition from a *safe state*: a state in which the Controller can handle what's happening in the environment without crashing, to an *alert state*: a condition in which without a Safety Monitor, the System would inevitably go to a failure state. For this component, we want to measure its ability to distinguish between safe states and alert states. To do so, we defined positives and negatives predictions in this way:

TP: True Positive

- The system went in an alert state due to a Controller's failure, correctly detected and prevented by the monitor

TN: True Negative

- The system is in a safe state and the Monitor doesn't raise any alarm

FP: False Positive

- The system is in a safe state, but the Monitor raises an alarm

FN: False Negative

- The system is in an alert state and the Monitor doesn't detect the hazard

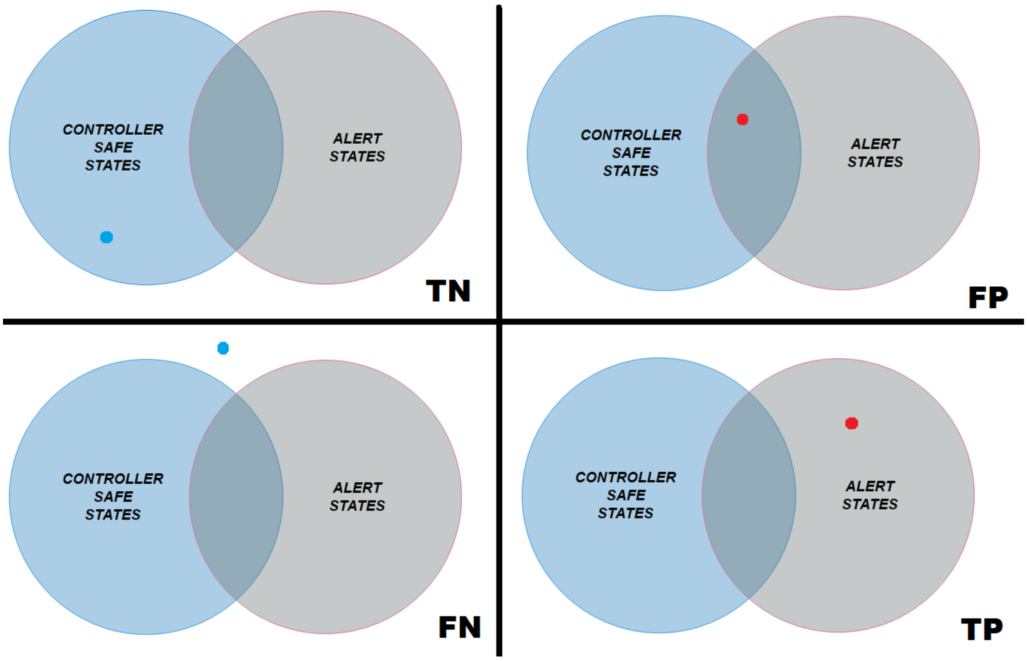

Figure 12: Graphical Representation of what true and false predictions means in the system's state space. Dots represent the current state of the System. A blue dot means that no alarm is raised, a red dot means that the Monitor raises an alarm

These measures are commonly used to see how good a model is in classification problems. However, for real time critical system it's not easy, if not impossible, to measure all of them. In particular, it's not always possible to understand when the Controller avoided a crash and the monitor *did not* raise an alarm. This issue makes very hard to measure the amount of true negatives because in most of the cases, this kind of situations may be seen only when a human operator analyse the specific run.

In particular, an input for the Safety Monitor is composed by multiple, timed inputs, that can be seen as the evolution of the environment while the system is performing its operations. For this reason, it is hard to understand whether the next state will be an alert or a safe state because we don't know *when* the series of inputs that will eventually result in a crash will begin.

This puts some limits on the metrics and the ratios that can be computed for the Monitor as they require the amount of true negatives. However, fortunately, a true negative does not change the behaviour of the system because there is no safety-brake, therefore after some analysis it was decided to ignore the true negative rate. At the same time, we want to observe at least the amount of false positives and convert it to a rate, for the purpose of comparison.

Since we do not know the *real* amount of *True Negatives*, for the reason just described, some of the measures usually considered when assessing the prediction performances of a model could not be approximated. To give a rate measure for false positives, it was decided to compute the amount of false positives over the distance travelled in a run.

- True Positive Rate:

$$\mathsf{TPR} = \frac{\mathsf{TP}}{\mathsf{TP} + \mathsf{FN}} \tag{3.1}$$

- False Negative Rate:

$$\mathsf{FNR} = \frac{\mathsf{FN}}{\mathsf{FN} + \mathsf{TP}} = 1 - \mathsf{TPR} \tag{3.2}$$

- False Positives per Meter:

$$\mathsf{FPM} = \frac{\mathsf{FP}}{meters\ travelled} \tag{3.3}$$

These rates gives a good approximation of the Monitor's behaviour and these measures can be used for the purpose of comparison.

When testing the Monitor, it is advisable to *"stress"* it under detrimental conditions, in order to collect more possible evidences and linkage among the data. Note that this aspect is independent from the *difficulty level* defined above, as it is something related to the environment itself that may give evidences about the *situations* in which the Monitor performs well, and those in which it doesn't (e.g. think about a LiDAR sensor in a very rainy environment: the observations will likely to be fuzzy/noisy) and it is more related to *fault injection* at a higher level.

The ways in which the parameters of a Safety Monitor can be tuned really depend on its implementation, and it's up to the developing team the decision on what and how many faults inject in the software. A very simple example can be that of reducing the amount of data read by the sensors (which will result in low-quality data and a poor mapping of the environment), or introduce some noise in the observations.

Doing this step when testing the Monitor allows also to understand how to tune the internal parameters of the software in order to have its *"best"* version, that will be used in the Controller retraining phase.

3.2.3 *Controller Retraining*

At this point we have collected data about the behaviours of the Monitor and the Controller. In this phase we are going to study how much this values vary with respect to the learning strategy adopted, when the neural network is retrained from the last checkpoint.

As said at the beginning of this chapter, we are considering a Controller composed by a neural network, trained with reinforcement learning techniques. These kind of algorithms make use of a *reward function* to tell to the network if it's behaving *"good"* (positive value) or *"bad"* (negative value) and it is calculated at each prediction step (i.e. at each action taken by the Controller).

The training function needs the following parameters, that are recorded at each algorithm iteration:

a) Action taken by the Network

b) State of the network in which the action was taken

c) Reward given for the action taken in a certain state

In this phase we want to discover how training strategies for the neural network may affect the overall behaviour of the system, with particular attention on the effectiveness of the Safety Monitor.

To start an exploration in this topic, we defined 4 strategies and the outcomes we expect after the training is completed. These outcomes are *"forecasted"* in the sense that we don't know how the network will react to a change in the training strategy, nor if the observe behaviour will be the supposed one.

The strategies developed in this method are mostly based on the reward function and on the actions taken by the network:

S1) The reward function is now more punitive when the car hits something, while braking is slightly more rewarded, provided that the Car won't stop moving

S2) The Safety Monitor is attached to the system and, if an alert is raised, the action of the network is replaced by the Monitor's action (the safety-brake)

S3) The network's action is replaced by the monitor's if it raises an alert. The network is given a positive reward for behaving like the Monitor

S4) The training step is stopped and the network is given a *negative* reward if an alert is raised

Using S1 we expect at least the Controller's mean time/distance between failures to be lower. Giving a more negative reward for collisions and a little positive reward for braking (if the car doesn't stop moving) should force the car to prefer a brake instead of a swerve (that could result in a *new* hazardous situation), therefore improving its time between failures and its travelled distance.

The purpose of S2 is to *"teach"* to the network to brake whenever an alarm is raised by the Monitor. A possible outcome is that the alert states previously covered by the Monitor become now safe states. However the Monitor's false positives will inevitably have an impact on the system's availability, caused by erroneous safety-brakes.

S3 is pretty much similar to S2. We expect that giving a positive reward when behaving like the Monitor may increase the learning speed.

S4 is the most promising strategy and it may give the most interesting results. Giving a negative reward and alting the training step when the Safety Monitor raises an alarm, regardless if it's a true or false positive, should force the network to completely *avoid* the situations in which the Monitor intervenes. Essentially we expect that the effectiveness of the safety component will be drastically reduced.

After the four Controllers are trained enough, the new systems are tested as in phase one. The same measures listed before are approximated for the Controllers and the Monitor and the results are compared with the original checkpoints.

===

FORSE

===

As an additional step, we are going to treat the new Controllers as a Monitor for the runs recorded with the last checkpoint, to try to observe how the states space has changed.

4

METHOD IMPLEMENTATION AND RESULTS

In this chapter the tools used, the software infrastructure and method implementation and the results collected during the analysis are reviewed.

A DDPG Agent[1] was trained to drive in an urban environment. Checkpoints of the network's state during the training were recorded for the purpose of comparison. These checkpoints of the network were then tested with and without a simple safety-monitor in order to provide a new point of view to study AV's behaviours.

4.1 TOOLS AND SOFTWARE

4.1.1 *Carla Simulator*

In order to have a realistic environment, with accurate physics simulation and data sensors, the open-source simulator CARLA[7], developed by researchers at the University of Barcelona, was used. This simulator was developed with the purpose of offering an environment where AI agents can be trained to drive, with high control of the simulation parameters and the simulation of realistic sensor, which can be tuned to increase or decrease data quality, or to inject faults.
CARLA is developed with a client-server architecture in mind. The *server* is basically a game, developed with *Unreal Engine 4* in C++. C++ performances are with no doubt essential to the functionality of the server: not only the environment must be simulated (including movements of pedestrians/vehicles, weather simulation...), but also all the data needed from the sensors attached to the system.

===

[1] An agent whose actions are taken by a neural network trained with the reinforcement learning algorithm DDPG

IMMAGINE CARLA

==

CARLA is currently at version 0.9.7 and huge improvements are done at every release, gaining more attention from the experts for its realism. Unfortunately, when this study started, CARLA 0.9 was recently released and the tools needed for our work couldn't be found online. Thanks to the quantity of work done for the last *stable* version of CARLA, 0.8.4 was used at first.
Versions prior to 0.9 have some limitations on the control one has of the simulations parameters and on the data collectable from it. This doesn't impede our study, but of course limited in some way the informations on the environment and system. Some of these problems are still present in later versions of the simulator, but most of them were solved in the transition from 0.8 to 0.9.

One of the main problems found was with the coordinate systems. Before version 0.9, developers were using UE4's default coordinates system which is left-handed, while the standard is considered to be right-handed. This looks like not a big deal since things could be easily solved by applying a transformation matrix. However, due to performance issues (a Python client should do the real-time processing of *loads* of data at each timestep, resulting in considerable slowdowns as a result of all the processes running at the same time), it was decided to stick with the developers' decision and convert the data during analysis phase.

Unfortunately, this version of CARLA has only 4 sensors available, which were all used during the experiments. They can be easily accessed via the Python APIs provided:

- Cameras
  - The *scene final* camera provides a view of the scene (just like a regular camera)
  - The *depth map* camera assigns RGB values to objects to perceive *depth* of the environment
  - A *semantic segmentation* is used to classify different objects in the view by displaying them in different colors, according to the object's class
- Ray-cast based Lidar
  - Light Detection and Ranging is use to sense the environment and measures distance from objects by illunating the target

with laser beams and measuring the time reflected light needs to "go back" to the sensor

The three cameras were used during the training phase of the network. Three *scene final* cameras are attached to the car to actually *see* the environment (one on the front and one per side). The *depth map* camera allows the car to get a colormap of the distances from objects in the scenario. The *semantic segmentation* provides image classification features by querying the server for ground-truth values. This is with no doubt a semplification of a real system, where the most powerful image-classification softwares are essentially other neural networks trained separately. At the same time a misclassification can be considered as an error of the control system: if the safety monitor detects the possible hazard it will not "correct" the misclassification but it must react fast and safely to avoid the possible consequences of it, therefore this simplification won't have an impact on the overall method.

A ray-cast based Lidar is the only other sensor available for this version of CARLA. Parameters of this sensor can be easily tuned to simulate real lidars such as the *Velodyne LiDAR* or to simulate faults such as low data quality, noisy data or data loss. . . This data are generated using the *Point Cloud*[19] format

In the simulations, due to the high hardware resources requirements to simulate a real LiDAR, a slightly modified version of the *Velodyne64 LiDAR* is implemented with the following parameters:

- Channels = 64
  - The number of laser beams used by the system. These lasers are distributed over the vertical axis. The more the lasers are, the more accurate will be the scannings
- Range = 75m
  - Lasers' range in meters
- Rotation Frequency = 15 Hz
  - This parameters define the rotation frequency (in Hz) of the scanning beams.
- Points Per Second = 1.000.000
  - The actual number of points generated each frame by the sensor

- Vertical FOV bounds (height = 24m, low = -2m. Distances are relative to the position of the sensor)
  - Maximum and minimum height of the scannings

The simulator provides Python APIs not only to modify sensors, but also to have a great control on what is being simulated, such as seeds definition for the spawning points and the behaviours of pedestrians and vehicles, and on the state of *"actors"* in the scene such as their position, their speed.... All these data are directly provided by the simulator with ground-truth values. These kind of measurements can be simulation-related, such as the simulation time-step, or the FPS. Actors-related measurements include for example vehicles' speed, intensity of collisions (if any) and the 3D acceleration vector.

While this tool was being tested a problem was found with the LiDAR sensor data that is still not resolved.[**?**] The bug resulted in the vehicles' bounding boxes to be distorted when moving, providing very poor data. After some research an improved version of the simulator was found, where the bounding boxes for each vehicle were redefined to provide accurate data.[8] As of today, developers are working on this issue but it is still unresolved without manual intervention on the source code.

4.1.2 *Controller Implementation*

The main component needed for the Controller implementation is the neural network that will be in charge of deciding the action to perform to drive the car. We looked for a framework with the following characteristics

1 Training code must be available

2 No critical issues in the codebase

3 Provide an environment to interface the network with CARLA

4 Provide a default training strategy

After analyzing all the machine-learning related projects for CARLA, we chose the reinforcement-learning framework *Coach*, developed by *Intel AI Lab*.[9]

This framework satisfies all the requirements listed above: it is distributed as an *editable Python package*. The development team behind this project assure us the quality of the product. Moreover, several presets are available as a starting point. Two of these presets offer an interface to the CARLA simulator, as well as a default training strategy, which is exactly what we required.

These presets implement the *Deep Deterministic Policy Gradient* algorithm, proposed in 2015.[18] This algorithm proved to perform well in tasks such as car driving, as shown in the original paper, and it's specifically adapted to perform in environments with continuos action space, such as the one we are considering.

P1) The first preset uses a single, front camera to perceive the environment, and the other data augmentation cameras provided by CARLA. The agent resulting from using this preset will likely not have any sense of depth created by the regular camera and will rely solely on the depth camera

P2) This preset uses all the cameras available in CARLA and have a much more interesting architecture since the car is equipped with three regular cameras (Front, Left, Right) and all the data augmentation cameras available in CARLA, listed in the previous section

The usage of the data augmentation cameras (depth and semantic segmentation) allows us to ignore objects misclassifications (e.g. labeling a pedestrian as a light pole) as well as some other easinesses such as computing distances from objects. As pointed in the previous section, this is indeed a simplification of a real architecture, where object detection modules are neural networks themselves, that must be trained in advance and tested separately. However, again, a misclassification will probably cause the Controller to behave in unexpected manners that the Safety Monitor must be able to detect and possibly correct.

Unfortunately, the first preset suffers from an issue that causes the car to stop throttling, while being positively rewarded[17]. It would have been interesting to test this preset as well, to understand what's the impact on the effectiveness of the Safety Monitor with respect to the quantity of data available to the network.

It's important to notice that our goal *is not* to build the *"perfect"* agent, nor the perfect autonomous cars. The codebase was studied but, due to reasons of time, we could not inspect all the details of the provided implementation. This framework was used as an example to demonstrate the concepts described in this work.

4.1.3 *Safety-Monitor Implementation*

In order to start the experimental activity, we needed a software capable of processing LiDAR data to map the environment and sense the obstacle. Moreover, this software needs to be *real-time* and capable of interacting with CARLA. The latter is obvious: whenever an alert is raised, the *"brake"* command must be sent to the CARLA server. The first required some reasoning: one could think to record the LiDAR data in advance and then run the simulations feeding the Monitor with these data. Unfortunately this approach can not work: this comes from the fact that we are not interested in having a 100% prediction accuracy, but also in the *goodness* of the safety-measure adopted by the Monitor. The same measurement in two different simulations could give *slightly* different results, that could cause the Monitor to behave in a completely different way than the way it could have behaved using the *real time* data produced during the simulation. If the accuracy of measurements is strictly dependent on the simulator used for the experiments (CARLA is improving at every release on this side), we could not ignore the effects that a brake can have on a single run.

In order to develop a decent Safety-Monitor, a research on the best, *"not-neural-network"* techniques was conducted, as well as a research on the open-source instruments available. The choice fell on the *Point Cloud Library*[21], an open-source library specifically dedicated to the processing of Point Cloud data, developed using C++ to achieve great performances when processing huge amount of data. This library was first released in 2011[20] and it's been improved at every release, also thanks to the big community testing and debugging the new features.

This library offers a multitude of functions that implements the most known techniques for Point Cloud processing. The steps to follow in order to develop an object-detection module are the following:

1) Downsampling

   → The data generated by a single scan can contain more hundreds of thousands records, with possibly a lot of redundancy and noise. A downsample is usually necessary as a preliminary step to discard all the *"useless"* data. After some reasoning, the Voxel Grid Filter was chosen for this step

2) Ground Segmentation

   → After the downsampling (if needed), the first mandatory step is to filter the data that are useless for the purpose of object detection, such as points relative to the ground. These point must be filtered in order to separate the ground from the objects we want to detect. In our implementation this is done using the RANSAC[2] algorithm[**?**], a technique to separate *"inliers"* (i.e. data whose distribution is characterized by the model's parameters) from *"outliers"* (i.e. data that don't have a representation in the chosen model)

==

IMMAGINE PRE-POST SEGMENTATION

==

3) Clustering

→ This final step is required to effectively identify what is an object in the scene, and what data points are relative to the same object. This can be done using clustering algorithms based on the distance among points. We chose the Euclidean Clustering Algorithm, a range search algorithm based on the euclidean distance between points, with the assumption that *very dense* points represent the same object

==

IMMAGINE CLUSTERING

==

4) Object Tracking and Avoidance

→ Objects detected in the first three steps must be tracked over time in order to recognize if two objects observed in two consecutive steps, are actually the *same* object, usually using physical models to predict their behaviour over time, such as the Kalman Filter, and combine this model with a failure-prevention routine

Unfortunately, implementing a Kalman Filter for such a complex model would have been too much time-consuming, alting our study. Therefore we simplified the object-detection and the failure-prevention routine in this way:

[2] Random Sample Consensus

- Only data *in front of* the car are recorded. This is a huge semplification of the model, as the Monitor now can only detect obstacles ahead of the car. However, this will for sure have an impact on the effectiveness of the Safety Monitor, but the concepts explained in the previous section still holds
- The failure-prevention routine implemented takes inspiration from the Responisbility-Sensitive Safety model proposed by Mobileye, an Intel company.[22] If an object is detected, its speed relative to the system is computed. If the distance travelled by the system in 1 second plus the braking distance is greater than the distance travelled by the object plus the distance between the system and the object, a safety brake is applied.

Since the Safety-Monitor is developed in C++, both for taking advantage of the Point Cloud Library and due to performance reasons, an infrastructure to exchange data with CARLA was developed.

The software is composed of a C++ server that receives LiDAR Point Clouds from the CARLA client. This data are processed as in the steps above (with the semplification described) and, if needed, an alert is sent back to the client. If the message received by the Monitor is an alert, the actions of the Controller are ignored and a brake is performed.

4.2 EXPERIMENTAL ACTIVITY

LEGGERE GROUND-TRUTH PERMETTE DI IGNORARE ERRORI DI MISCLASSIFICATION DEGLI OGGETTI

DISCUTERE LE STRATEGIE DI TRAINING

4.3 RESULTS

METTI QUI I NUMERINI CHE TI SONO VENUTI

5

CONCLUSIONS

BIBLIOGRAPHY

[1] Peter Popov, Lorenzo Strigini - *Assessing Asymmetric fault-tolerant Software* (Cited on page 24.)

[2] Nidhi Kalra, Susan M. Paddock - *Driving to Safety: How Many Miles of Driving Would It Take to Demonstrate Autonomous Vehicle Reliability?* (Cited on page 22.)

[3] Arizona 2018 Uber Incident - *https://en.wikipedia.org/wiki/Death_of_Elaine_Herzberg* (Cited on page 21.)

[4] Elon Musk declarations - *https://techcrunch.com/2020/02/18/elon-musk-says-all-advanced-ai-development-should-be-regulated-including-at-tesla* (Cited on page 23.)

[5] Uber incident Preliminary Report - *https://www.ntsb.gov/investigations/AccidentReports/Reports/HWY18MH010-prelim.pdf* (Cited on page 21.)

[6] Philip Koopman - *http://safeautonomy.blogspot.com/* (Cited on page 21.)

[7] CARLA - *http://carla.org/* (Cited on page 43.)

[8] Improved LiDAR CARLA - *https://github.com/ZhuangYanDLUT/carla*

[9] Nervana Systems - Coach - *https://github.com/NervanaSystems/coach* (Cited on page 46.)

[10] Xingyu Zhao, Valentin Robu, David Flynn, Kizito Salako, Lorenzo Strigini - *Assessing the Safety and Reliability of Autonomous Vehicles from Road Testing* (Cited on pages 22 and 32.)

[11] Bev Littlewood, Lorenzo Strigini - *Validation of Ultra-High Dependability for Software-based Systems* (Cited on page 22.)

[12] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, Piotr DollÃ¡r - *Focal Loss for Dense Object Detection* (Cited on page 22.)

[13] DeepMind - https://deepmind.com/research/case-studies/alphago-the-story-so-fa (Cited on page 22.)

[14] Google AI defeats human Go champion - *https://www.bbc.co.uk/news/technology-40042581* (Cited on page 22.)

[15] Anh Nguyen, Jason Yosinski, Jeff Clune - *Deep Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images* (Cited on page 23.)

[16] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, Rob Fergus - *Intriguing properties of neural networks* (Cited on page 23.)

[17] Coach Issue - *https://github.com/NervanaSystems/coach/issues/428* (Cited on page 47.)

[18] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, Daan Wierstra - *Continuous control with deep reinforcement learning* (Cited on page 47.)

[19] Point Cloud Format - *https://en.wikipedia.org/wiki/Point_cloud* (Cited on page 45.)

[20] Point Cloud Library Info - *https://en.wikipedia.org/wiki/Point_Cloud_Library* (Cited on page 48.)

[21] Point Cloud Library - *http://pointclouds.org/* (Cited on page 48.)

[22] Responsibility-Sensitive Safety - *https://www.mobileye.com/responsibility-sensitive-safety/* (Cited on page 50.)

[23] J.C. Laprie - *Dependability - its attributes impairments and means, Predictability Dependable Computing Systems* (Cited on page 9.)

[24] J.C. Knight - *Safety Critical Systems: Challenges and Directions, Proceedings of the 24th International Conference on Software Engineering* (Cited on page 9.)

[25] Andrea Bondavalli - *L'analisi quantitativa dei Sistemi Critici, Fondamenti e Tecniche per la Valutazione - Analitica e Sperimentale - di Infrastrutture Critiche e Sistemi Affidabili* (Cited on page 13.)

[26] G.J. Nutt - *Tutorial: Computer system monitors* (Cited on page 14.)

[27] B. Plattner - *Real-time execution monitoring* (Cited on page 14.)

[28] B.Plattner, J. Nievergelt - *Special feature: Monitoring program execution: A survey* (Cited on page 14.)