Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Simulation-Based Robustness Testing of ADAS Systems

Scientific Students' Association Report

Author:

Attila Ficsor
Balázs Pintér

Advisor:

dr. András Vörös
dr. Oszkár Semeráth
Aren A. Babikian

2021

Contents

Kivonat

Napjainkban egyre gyakrabban használnak mesterséges intelligencián alapuló komponenseket a fejlett vezetőtámogató rendszerek (ADAS) fejlesztéséhez. Az autóktól kezdve a villamosokon át az önvezető ipari targoncagépekig számos alkalmazás létezik már, amelyekben autonóm komponenseket használnak, és még több van fejlesztés alatt. Ezeket a járműveket autonóm komponensek irányítják vagy segítik, amelyek az érzékelőik által gyűjtött adatok alapján hoznak döntéseket.

Ezek a mesterséges intelligenciát alkalmazó komponensek kritikusak, így kiemelten fontos a helyes viselkedésüket biztosítani, mivel hiba esetén jelentős anyagi károk keletkezhetnek, vagy akár emberi életek is veszélybe kerülhetnek. Még a legkorszerűbb tesztelési technikák sem képesek hatékonyan támogatni az automatizált tesztelést. Ezeknél a rendszereknél komoly kihívást jelent, hogy a paramétertér végtelen, valamint a fizikai szenzoradatok kezelése is nehéz. Továbbá a rendszer követelményei nem teljesen ismertek, mivel nem rendelkezünk a legjobb vezetési gyakorlatok teljes listájával.

Munkánk során egy olyan tesztelési módszer és eszközkészlet elkészítése és bemutatása a cél, amellyel hatékonyan lehet a járművek ADAS moduljait tesztelni. A komplex környezet miatt számos különböző forgatókönyvet kell lefednünk releváns mutációkkal, hogy teszteljük az MI-alapú komponensek robusztusságát.

Egy nagy teljesítményű gráfgeneráló algoritmus segítségével tesztforgatókönyvek absztrakt ekvivalenciaosztályait generáljuk. Ezek a tesztek meghatározzák az összes szereplő magas szintű viselkedését, beleértve a vizsgált járművet, valamint a környezet egyéb objektumait is, amelyek befolyásolják az ADAS-modulok döntéshozatalát. Először a fejlett absztrakciós technikák alkalmazásával szemantikailag különböző közlekedési szituációkat generálunk. Ezután minden egyes szituációhoz sokféle tesztesetet generálunk releváns, különböző, de szemantikailag megegyező mutációkkal, így tesztelhetjük az ADAS-modulok robusztusságát. A generált tesztekhez egy valószínűségi változókat és eloszlásokat kezelő forgalmi szituációkat leíró nyelvet (Scenic) használunk. A tesztesetek szimulációs környezetben (Unreal Engine-en futó CARLA) futtathatók. A szimulációs környezetben valós helyszínek használata is lehetséges: térképadatokból (OpenStreetMap, Google Maps) valósághű térképeket tudunk származtatni, így a teszteseteket valósághű környezetben lehet végrehajtani. A szimulátor emellett valósághű fizikai jármű- és gyalogosmodelleket és szenzoradatokat tartalmaz és ami a legfontosabb, a tesztesetek kiértékeléséhez szükséges, elvárt kimeneteket is rendelkezésre bocsájtja.

Dolgozatunkban egy olyan eszköztárat mutatunk be, amely lehetővé teszi az ADAS-modulok teszteseteinek automatikus generálását és végrehajtását. Az első lépésben végzett magasszintű modellgenerálás a tesztesetek szemantikai változatosságát biztosítja, míg a második lépésben bemutatott mutációk segítségével tesztelhetjük az ADAS-modulok robusztusságát is. Az absztrakt modellek generálásához egy nagy teljesítményű gráfgeneráló algoritmus továbbfejlesztett változatát használjuk. Egy valósághű szimulátor segítségével a tesztjeinket a valós környezetek szimulált változataiban is végre tudjuk hajtani.

Abstract

Nowadays components based on artificial intelligence are more and more commonly used for developing Advanced driver-assistance systems (ADAS). From cars to trams and forklifts, many applications already exist and use autonomous components, and even more are under development. These vehicles are controlled or assisted by such autonomous components, that make decisions using the data collected by their sensors.

The correct behavior of these components using artificial intelligence is critical from a safety point of view because, in case of an error, major property damage can occur, or human lives can be in danger. Therefore, the automotive industry specifies strict safety standards, which are challenging to satisfy. However, state-of-the-art testing techniques were unable to support automated testing. Complete verification of these components is not possible, because the feature space is infinite, even after abstraction methods, so we can't inquire physical sensor data. Furthermore, the requirements of the system are not completely known, since we don't have a complete list of driving best practices.

Our goal is to define a testing approach with a prototype toolchain that can effectively support the systematic testing of ADAS modules of vehicles. Due to the complex operating environment, we have to cover many different scenarios with relevant mutations to test the robustness of the AI-based components.

Using a high-performance graph generator algorithm, we generate abstract equivalence classes of test scenarios. These tests define the abstract behaviors of all the actors including the tested vehicle, as well as the other objects in the surrounding environment that may alter the decision-making of the ADAS modules. First, using advanced abstraction techniques we aim to generate a wide range of semantically different traffic scenarios. Then, for each abstract scenario, we generate a wide variety of test cases with relevant, diverse but semantically equivalent mutations to assess the robustness of ADAS modules. The generated tests are described in a probabilistic scenario description language (Scenic) and the tests can be executed in a simulation environment (CARLA running on Unreal Engine). Mapping real locations into the simulation environment is also possible: we can derive realistic maps from real map data (OpenStreetMap, Google Maps), so the test cases can be executed in a realistic environment. The simulator also provides correct physical vehicle and pedestrian models and realistic sensor data and most importantly ground truth.

Our solution provides a toolchain to automatically generate and execute new test cases for ADAS modules. The abstract model generation in the first step provides a diverse base of test cases, while the mutations introduced in the second step help us test the robustness of the ADAS modules. For generating the abstract models, we use an improved version of an already high-performance graph generator algorithm. Using a realistic simulator, we can execute our tests in simulated versions of real-life environments.

Chapter 1

Introduction

Advanced Driver-Assistance Systems (ADAS) and Autonomous Driving Systems (ADS) are nowadays one of the most common (safety) critical machine learning (ML)/deep learning (DL) based systems, therefore proving the correct behavior of such systems is important. These systems usually contain one or more machine learning or deep learning components, thus their formal verification is not possible.

To prove the safety of an ADAS or an ADS, it needs to be thoroughly tested. Testing the components is an important and challenging task (e.g. proving the correct behavior of a computer vision component), but only the system level testing can showcase the correct behavior of the whole system. One way to test the system is real-world testing, but as written in [18], it requires billions of driven kilometers with 100 test cars, just to prove the system's safety in a probabilistic way (compared to human drivers).

To make the testing process more scalable, the test cases must be executed in a simulation environment. There are many photo-realistic simulators with relatively good performance, but the real challenge is to create a set of diverse and meaningful scenarios. A test set, that covers the common scenarios and most of the possible corner cases as well. Manual scenario creation is possible, but it requires a lot of manpower. There are also many scenario (and maneuver) datasets available [10, 20], but most of these datasets contain concrete scenarios or trajectories, which means limited mutation/perturbation possibilities compared to functional or logical scenarios.

Using high-level (functional) scenarios is a good way to create concrete test cases. Functional scenarios can be derived to logical scenarios, even concrete scenario generation is possible. A functional scenario gives a group of semantically equivalent logical/concrete scenarios. Logical scenarios are general too (but more detailed than functional), there is a lot of freedom at generating concrete scenarios: we can generate many concrete scenarios with small mutations (environment, static or dynamic object behavior/property change, weather conditions, etc.), this is a good way to test the robustness of a system or find any adversarial scenario.

Generating and capturing scenarios are not only good for testing, but it is also a good way to create diverse, not biased training data. Simulators can provide many different sensor data (camera, lidar, etc), semantic segmentation view and the ego vehicle's actuator signals (execution trace). Such data is useful for training the ML/DL components of any ADAS/ADS.

Simulation environment can be synthetic, there are many realistic, but not existing cities in video games or simulators, however real world based maps can be helpful, when it comes

to testing the system in a concrete environment. Some firms automated the real world to simulation environment capturing process (e.g.: Waymo's self driving car builds up a 3D environment model while driving, and it's simulator can run scenarios in this virtual environment). Unfortunately there is no such open source project yet, but we can derive maps based on real world including 3D buildings with publicly available data and tools.

In our work we present a toolchain, which is capable of creating and executing test cases, targeting critical behavior of AI-based systems, in a real world based simulation environment. From these executions we export pictures and semantic segmentation images, which then we use as a test suite to measure the accuracy of an existing well-known image recognition AI model. Our approach relies on advanced simulation and graph generation technologies. In addition, our solution can help in the testing of existing AI application, or aid their training in dangerous situations.

This report has the following structure:

- First, in Chapter 2 we will discuss the most import technological and theoretical background knowledge used in our work. This includes an overview of advanced driver assistance systems and modeling and model generation technologies.
- Next, we will provide a detailed overview of our proposed approach in Chapter 3.
- Then in Chapter 4 we further detail our implementation, and we provide a technological overview of the tools we used during our work.
- Next, we will evaluate our proposed test generation tool and the quality of the generated test cases in Chapter 5.
- In Chapter 6 we will summarize the current state of machine learning related testing approaches,
- and finally we draw a conclusion on our work and describe our plans for the future in Chapter 7.

Chapter 2

Preliminaries

2.1 Advanced driver assistance systems

Advanced driver assistance systems are popular, nowadays not only high-end cars have assistance features: almost every new car includes an anti-lock braking system (ABS), traction control, and many have adaptive cruise control (ACC), forward collision warning or others. Lately newer features are also popular: lane keeping or parking assistant, adaptive variable suspension and more.

Various driver assistance systems exist, with different features and usage. Some of them make driving comfortable (e.g. adaptive cruise control), some of them protect the user and other traffic participants (e.g. precrash systems). There are ADAS with or without actuators: emergency brakes can physically intervene when danger is detected, but alert systems only help at the driver's decision making.

Autonomous driver assistance systems are up to 2 on the driving automation level[1], meaning that it automates the driving only partially. The level 1 assistance systems can control the vehicle either laterally or longitudinally, level 2 assistance systems can do both: accelerate, brake and steer at certain scenarios, but the driver must supervise the system at all times.

Many vehicles include an automatic emergency braking (AEB) system, it can recognize dangerous scenarios and brakes (forward collision warning, but it can intervene). Some vehicles use radar, lidar, camera or a combination of these sensors, to sense the environment. Based on the perception it can calculate the crash potential, and intervene if needed. Tesla Autopilot is an ADAS too, it can keep lane and do basic maneuvers like turning, overtaking, but does not meet the Level 3 requirements, a human driver must always supervise the system.

For obvious reasons, driver assistance systems must be extremely safe and reliable, from software and hardware aspects as well. Testing systems without ML/DL components is challenging, but in our work, we focus on systems with ML/DL components.

[1]SAE J3016: Levels of driving automation `https://www.sae.org/standards/content/j3016_202104/`

2.2 Toward the testing of AI components

2.2.1 Functional overview of autonomous components

Let us review the functional overview of the components (illustrated in Figure 2.1) as based on state-of-the art publications [11] and autonomous driving standards like [4, 5].

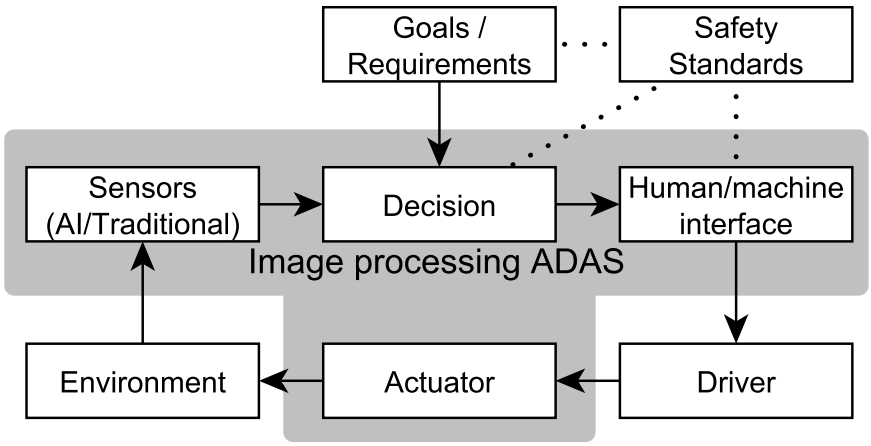


Figure 2.1: Functional overview of a generic ADAS component.

- An *Advanced driver-assistance systems (ADAS)* are operating in a complex *environment (context)*.
- The environment of the system is observed by *sensors* (e.g. camera, LIDAR and radar subsystems to measure distance).
- Then the ADAS component makes *decisions* based on the observed situation.
- The decision made by the ADAS component is translated to changes in the *Human/Machine Interface* to inform the driver.
- The *driver* in turn controls the vehicle through *actuators*, which has impact on the environment.
- *Safety standards* and best practices impose strict *requirements* and design goals on the implementation of the ADAS component.

2.2.2 Fault hypotheses

With this functional architecture, we identified the following main fault hypotheses:

- **Accuracy of Object Detection**: Despite the wide range of vision-based approaches using AI components, the accuracy of such solutions is still far from the reliability metrics expected in safety-critical systems. For example, in a well-referenced vision based benchmark [17] with clearly visible geometric objects the state-of-the-art accuracy is still only 99.8% [29][2]. Therefore, a primary goal is the systematic evaluation of object detection components in order to compare the accuracy with an expected reliability.

[2]State-of-the-art accuracy for benchmark [17]: `https://paperswithcode.com/sota/visual-question-answering-on-clevr`

- **Robustness:** In vision-based AI applications, robustness is a known issue; even in simple, well-studied domains (e.g. as number detection [19]) existing approaches consistently fail with the modification of a few (1-2) pixels. Therefore, several approaches train or validate such AI components with data sets enriched with modified (or *mutated*) images.
- **Coverage:** Decision making in visual input is a challenging data-oriented problem, where traditional testing techniques, equivalence partitioning approaches, and coverage metrics are insufficient. As a consequence, the training and validation of AI components is carried out using data with realistic distribution. Since dangerous situations are rare, a huge amount of test data is required to reach statistically significant confidence. (As a comparison, this requires almost 8 billion kilometers for a fleet of 100 self-driving cars to show the failure rate of an autonomous vehicle is lower than the human driver failure rate [26].)
- **Missing Requirements or Training Data:** Finally, the list of requirements and driving best practices is incomplete. Therefore, it is necessary to provide tools to systematically synthesize untested traffic situations to discover missing requirements. However, synthesizing such scenarios is a computationally challenging task [23], which requires specialized logic reasoners [7].

2.3 Modeling technologies

The purpose of domain-specific languages is to provide an efficient modeling language for specific domains. Such domain-specific languages and modeling languages can be created with Eclipse Modeling Framework (EMF). The model describing the language is called a metamodel, and it summarizes the main concepts and relations of the language. Using this metamodel we are able to generate Java classes, or we can use it directly in other applications, for example as part of a graph generator or graph search algorithm.

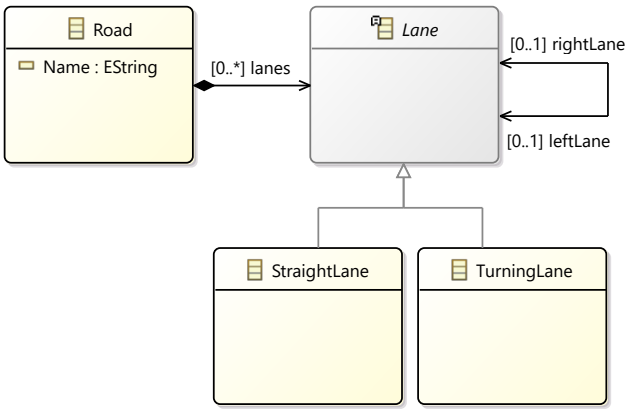


Figure 2.2: Metamodel of a simple road network

We will explain the concept and creation of a metamodel, and the relevant parts of EMF through a case study describing the simplified metamodel of a road network as seen in Figure 2.2.

The datastructures are built from the following base elements and their specializations:

- EClass: a class with zero or more attributes and zero or more references.

- EAttribute: an attribute with a name and a type.
- EReference: an association between two classes.
- EDataType: the type of an attribute, e.g. `int`, `float` or `java.util.Date`

Classes may be set as abstract, such as the `Lane` class in our example. This means they cannot be instantiated, but their non-abstract specializations can be. In the example, `Lane` has two specializations, `StraightLane` and `TurningLane`. This relation is represented in Figure 2.2 with an arrow pointing toward `Lane`.

An EReference can denote several types of relations, but all of them must have a name and a multiplicity, which may be a specific numeric value, or an interlval. For the latter, the lower bound is always a number, while the upper bound can be a number or infinity denoted by either -1 or the * character.

An EReference usually means a simple reference. A special case of this is the bi-directional or inverse reference, which creates two EReferences for a given relation. These are the inverse of each other, they represent the same relation from two opposing directions. On the diagram this is represented by a line with an arrow at both ends. An example for this can be seen in Figure 2.2, as the `rightLane` and `leftLane` references of the `Lane` class.

EReference may also represent a composition, which appears between classes `Road` and `Lane`. In the models usually, there is a single class that contains all the other classes.

Chapter 3

Generating test cases for simulators

Currently, every AI developer of advanced driver assistance systems or self driving vehicles uses their own internal methods for testing these autonomous components. There are a few publicly available sets of test cases [10, 20], but these only list a limited number of vaguely defined requirements that the vehicles must meet (for example, Voyage.auto provides an informal description of the test situation, which cannot be automatically executed in a simulator). Satisfying these requirements is necessary, but not sufficient to guarantee the safe operation of the vehicles. Our testing framework on the other aims to provide a way to test the AI components in diverse scenarios, while also ensuring the robustness of the system.

3.1 Scenes and Situations

During our test generation, at the scenarios we followed the three abstraction levels described in [22]: functional, logical and concrete. Concrete scenario is a sequence of scenes, logical scenario is a sequence of situations, a functional scenario consist of logic situations.

- Scene: Scene and other terms were inconsistent in the early literature, [25] introduced the standardized terms. According to [25], a scene describes a snapshot of the complete environment. In our simulation-based application it means, that the scene contains all information about the state of the simulation: the input data of all sensors are fully specified by the scene. A scene can be interpreted as a static snapshot of the simulation. Accordingly, concrete scenarios represent a sequence of fully specified scenes. The concrete scenarios are fully reproducible, and only concrete scenarios can be directly converted to executable test cases.
- Situation: a projection of a Scene excluding details that are not relevant to the input data of the ego vehicle. Therefore, information outside the situation should not have any impact on the behavior of the ego vehicle. The goal with the introduction of situations is to make the description of test cases more focused, and to group similar test cases into semantic equivalence classes. A sequence of partially specified scenes: parameters of a scene are given as a range of possible values. A logical scenario is a group of concrete scenarios, and by sampling from the parameter ranges, a concrete scenario can be constructed. A logical scenario describes usually infinite concrete scenarios, due to the common continuous values in the operational domain.

- Logic situation: [21] specifies a qualitative abstraction of a Situation, which extracts only the relevant information (with respect to the requirements). For example, instead of the concrete coordinates of vehicles, a logic situation represents the abstract relations between the actors (left-to, right-to, front-of, on-lane, close-distance, far-distance). Therefore, if two situations have the same logic situation, then the ego vehicle should have the same (or at least similar) behavior. For example, if a vehicle is close to, on the same lane as and in front of the ego vehicle, then the ego vehicle should start to slow down (regardless of the concrete coordinates of the actors). The most abstract level of scenarios can be formulated by domain experts in natural language. The building blocks of the sequence are semantically described logical situations.

3.2 Functional overview

Our testing framework consists of five main steps as illustrated in Figure 3.1. First, we create a map file that describes a road network. Then we generate the abstract situation, which determines the relations between the actors, and we also generate the actor's behavior. In the third step, concrete coordinates are calculated for the positions of the actors, and some elements in the surrounding area are varied for robustness testing. Next, we load the map and the starting scene into a simulator and execute the behaviors assigned to the actors during the simulation. In the final step we can evaluate the tests based on sensor data, video stream and execution trace provided by the simulator.

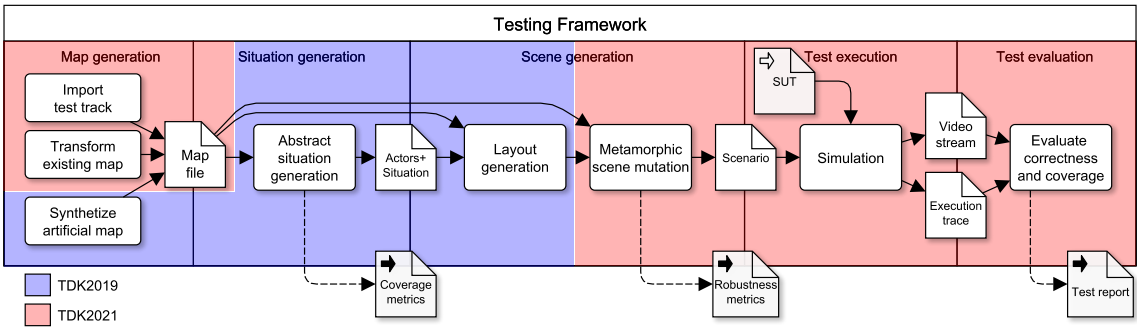


Figure 3.1: Functional overview

The presented testing framework is built upon and greatly improves our previous work from 2019 [14]. In Figure 3.1 our previous and current work phases are highlighted with different colors. While the aim of our previous work was to generate abstract situations on synthesized artificial maps, now we focus on importing real maps, generating concrete scenes and integrating a simulator into a complete workflow.

3.2.1 Map generation

While testing vehicles in real life is usually done on a test-track or sometimes on public roads, a simulation environment gives us more options. The easiest of them is if someone provides a premade map of an existing test-track, such as ZalaZONE[1], in OpenDrive format. This ensures that the test executed in the simulator can be reproduced in real life if it is needed.

[1] `https://avlzalazone.com/`

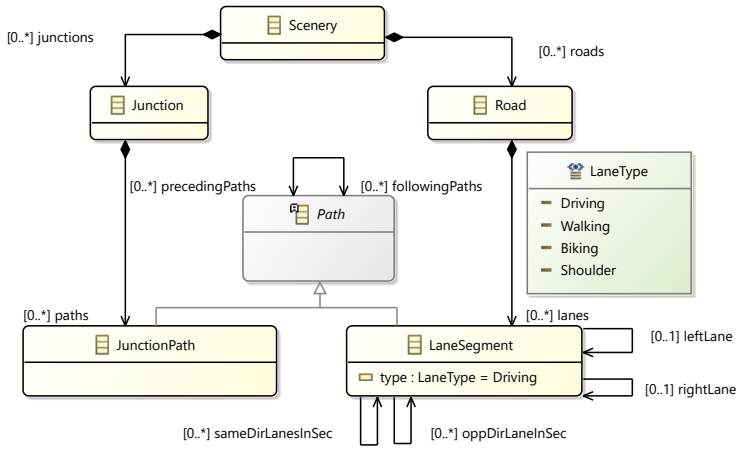
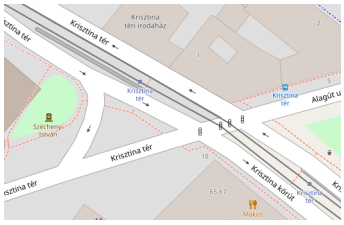


Figure 3.2: Metamodel of the road network

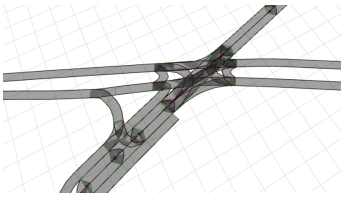
Figure 3.2 illustrates a the metamodel we used to represent road networks, which is generalized from the OpenDRIVE standard[2].

We can use OpenStreetMap to get the map data, transform it to OpenDRIVE and RoadRunner to create the 3D environment for the simulator (see 4.4 for more details). 3D building models are also available publicly, so there is no need to manually construct them. Common street objects and traffic infrastructure are also constructed here: lane marking, traffic signs, street lights, poles, vegetation and others.

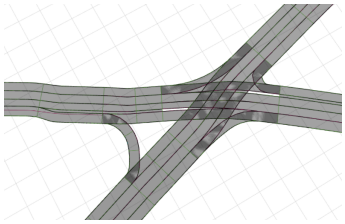
This map generation includes the first two layers of the four level scenario construction approach, presented in [28]. Road structure (OpenDRIVE) and infrastructure (Base 3D model of the location) belong here, these will not change, even during the perturbation generation.



(a) Original location

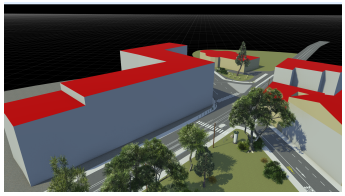


(b) OpenDRIVE from OpenStreetMap



(c) Manually modified OpenDRIVE

(d) Google Maps 3D



(e) Buildings from OpenStreetMap

(f) Buildings from Google Maps

Figure 3.3: Map generation methods

An other method available to us is to generate synthetic maps. Such generation method is described in [14].

Example 1. *Figure 4.4 provides insight into the environment building process.*

[2]Current implementation available at `https://github.com/ArenBabikian/CarlaScenarioGen`

- *Figure 3.3a: screenshot of the location, from* `openstreetmap.org`
- *Figure 3.3b: the OpenDRIVE file generated directly from the exported openstreetmap file. It contains many incorrect path and lane, the lanes of the trams are missing.*
- *Figure 3.3c: Manually modified OpenDRIVE file, based on the real lanes and paths in this area.*
- *Figure 3.3d: Screenshot of the birds eye view from Google Maps.*
- *Figure 3.3e: Virtually constructed environment based on the building database of openstreetmap:* `https://osmbuildings.org/`. *The vegetation is not part of the building database, it was added manually.*
- *Figure 3.3f: Virtually constructed environment based on the 3D building models of Google Maps*

3.2.2 Situation generation

Abstract situations describe only the relevant parts of a scene: relationships between objects, high-level states and behavior of the dynamic element are constructed here. All of this was generated by a graph generating method based on a scene metamodel with multiple level of abstraction available in Scenic [15][3]. At this point no concrete coordinates or maneuver paths are known, only relative positions as Figure 3.4 shows.

Abstraction is important, when it comes to generating dynamic objects, as we want to cover many semantically different scenarios with as few concrete scenarios as we can. This way coverage metrics can be established for abstract situations [21], but this is beyond our work.

Example 2. *Figure 3.5a and Figure 3.5b show two different scenes in Carla. However, both of them can be described by the same abstract situation depicted in Figure 3.5c, where we have an Ego actor, another actor, v1 in front of the Ego, and a third actor, v2 to the left of v1.*

3.2.3 Scene generation

In this step, concrete scenes are generated based on the abstract situation created previously. Every dynamic object has to satisfy the relations and constrains of the abstract situation, considering the base map. There are tools to generate such layout (Scenic, [14, 7]).

Static objects and their perturbation is also generated here, based on the map file. Figure 3.6 shows the metamodel of the perturbation. We used 7 different types of objects[4], and we generated them, around the map. The number of each object type was random, but below 9 in a scene. Also, the weather is also present in the perturbation: fog, rain, sun and sky parameters can be configured for each scene.

Example 3. *Figure 3.7a shows the layout of the dynamic objects, Figure 3.7b shows an example layout including static objects.*

[3]Current implementation available at `https://github.com/ArenBabikian/CarlaScenarioGen`

[4]Each object types can have multiple blueprints: e.g: there are 4 different bench blueprint

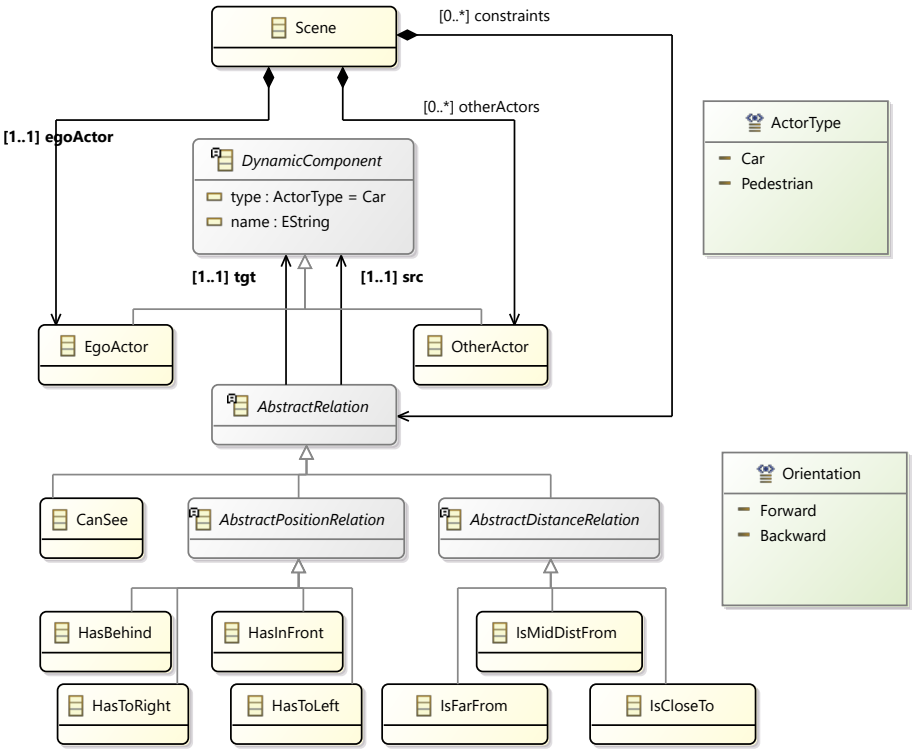


Figure 3.4: Metamodel of the abstract situation

Example 4. *Figure 3.8 shows the generated static objects, in the background. Pictures in the same row were taken from the same location, with different object layout.*

Example 5. *We constructed multiple weather blueprint, with different fog, rain, cloud and sun parameters, Figure 3.9 shows the same scenario with different weather.*

Combining the concrete static and dynamic objects on the map gives a scenario. The result of the layout generation looks like 3.7a. (excluding static objects, due to the visibility.)

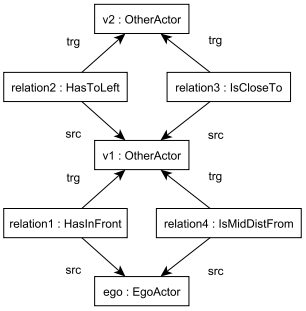
The 4th level of the 4 level model [28] is the environment conditions. Testing at different environment conditions is really vital, as proven in [24] small changes of the light or weather conditions can result in a bad (and dangerous) path prediction. At the perturbation we also generated weather types, with different fog, rain, sun and sky parameters. These are separated from the scene file, but at the execution it can be configured: this provides more portability.

3.2.4 Test execution

Given a scenario, we can load it into a simulator. For this purpose we use the open-source Carla running on Unreal Engine, but there are other options as well, like LGSVL or the popular video game Grand Theft Auto V. The scenario can contain behavior for all the actors which are executed in this step. Configuring the environment conditions is also executed here, we can change the weather even in run time. The SUT may be implemented separately, so we are able to test different versions of the system in the same scenario. From the simulator we get a video stream, an execution trace and all the sensor data necessary for evaluating the results. In our work we captured images from the position of a dashcam, with the corresponding semantic segmentation images as ground truth.

(a) Situation 1 in Carla

(b) Situation 2 in Carla



(c) Abstract model

Figure 3.5: A model of an abstract situations and two examples of it in the simulator

3.2.5 Test evaluation

Using the sensor data and ground truth received from the simulator we are able to train and test components or entire systems. For our demonstration we created training and test data sets for object recognition AIs used in a vision based benchmark [29].

3.3 Testing approaches

Proving the correct behavior of an ML/DL-based component or system is a challenging task: Verification is not possible on large scale, and testing requires a lot of resources (either data, computing capacity and/or time).

With our testing workflow multiple testing approach is realizable:

- Metamorphic testing
- Coverage based

3.3.1 Metamorphic testing

Metamorphic testing methods eliminate one of the hardest parts of the testing: getting a good oracle. For metamorphic testing knowing the ground truth is not required in advance. The main idea of metamorphic testing is to use the metamorphic relations between inputs, and check the results of the component/system for those inputs. The outputs for similar[5] inputs should be the same, or close to each other (e.g.: a picture of a giraffe should be recognized, even after some noise is added). This can reveal many errors of the system, when the results are (too) different there must be an error, but it can not prove the lack of failures: the output can be the same for the mutated inputs, but both can be wrong. Although this is a serious problem, metamorphic testing is still really useful, to prove the robustness of ML/DL-based components/systems. For computer vision components metamorphic relations can be affine transformations (e.g. rotation, scaling, etc..), but we can create equivalence classes for the input, based on semantic data: adding, changing or removing objects in the background should not change the object detection capabilities of a computer vision system of the system, but as described in [13, 27] in many cases it does.

[5]two inputs are similar, when they have a metamorphic relationship, meaning that one can be transformed to the other using metamorphic perturbation methods (e.g.: affine transformation, adding or removing some noise, semantic mutation)

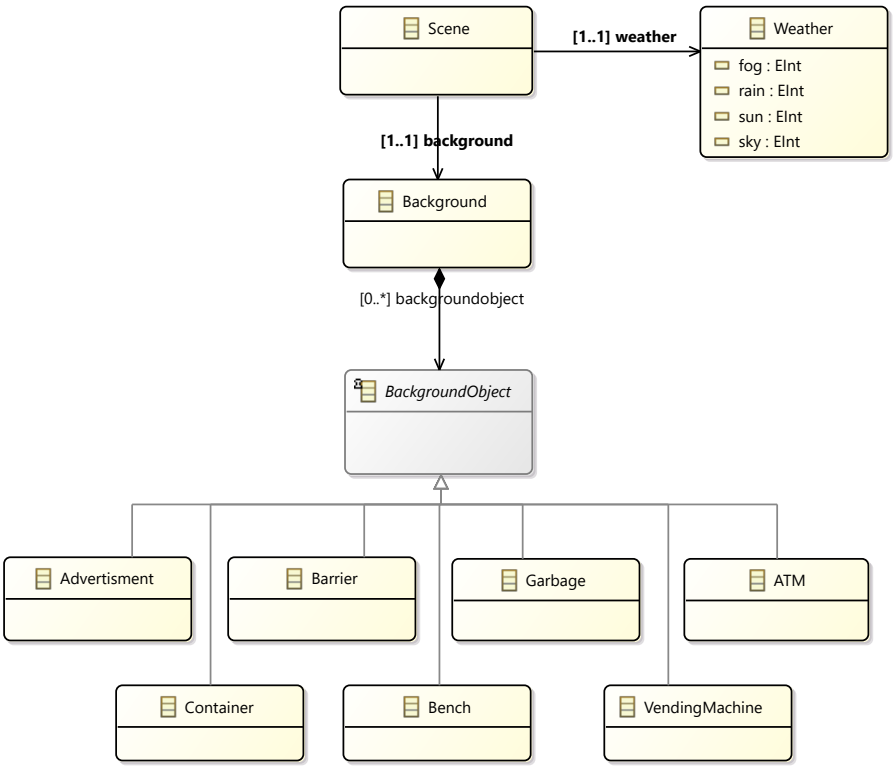


Figure 3.6: Metamodel of the perturbations

Different filters on an image can also make computer vision components fail, as presented in [24]. In [24] (and in our case too), the filters were weather conditions: rain, fog, cloudiness, and sun parameters.

For metamorphic testing we generated many metamorphic perturbations to each equivalence class. This way we generated plenty semantically similar, but perturbed test case.

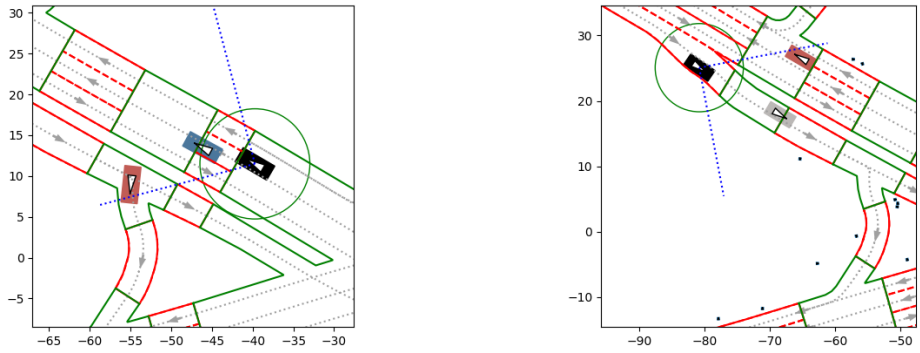
3.3.2 Coverage based testing

Based on the metamodel of the abstract situation multiple, situations can be generated with graph generating methods. As the abstract situation only contains a few different relations and elements due to the qualitative abstraction, the number of possible abstract situations are finite (with a certain number of actors), and even the similarity of the abstract situations can be measured by shape analysis techniques.

For coverage based testing we generate as many different abstract situations as possible, while introducing only a limited number of metamorphic mutations.

3.4 Summary of scenario building

During the testing workflow in 3.1, we followed the 4 level scenario hierarchy, mentioned in [28]. This allowed us, to handle the different levels of a scenario in each abstraction layer differently, making the process more transparent and portable.



(a) Vehicles **(b)** Layout with static objects

Figure 3.7: Layout of an abstract situation

Figure 3.8: Static object comparison

- *Level 1: Road network*

  Road structure is described with roads, lanes, junctions with the corresponding geometry and properties.

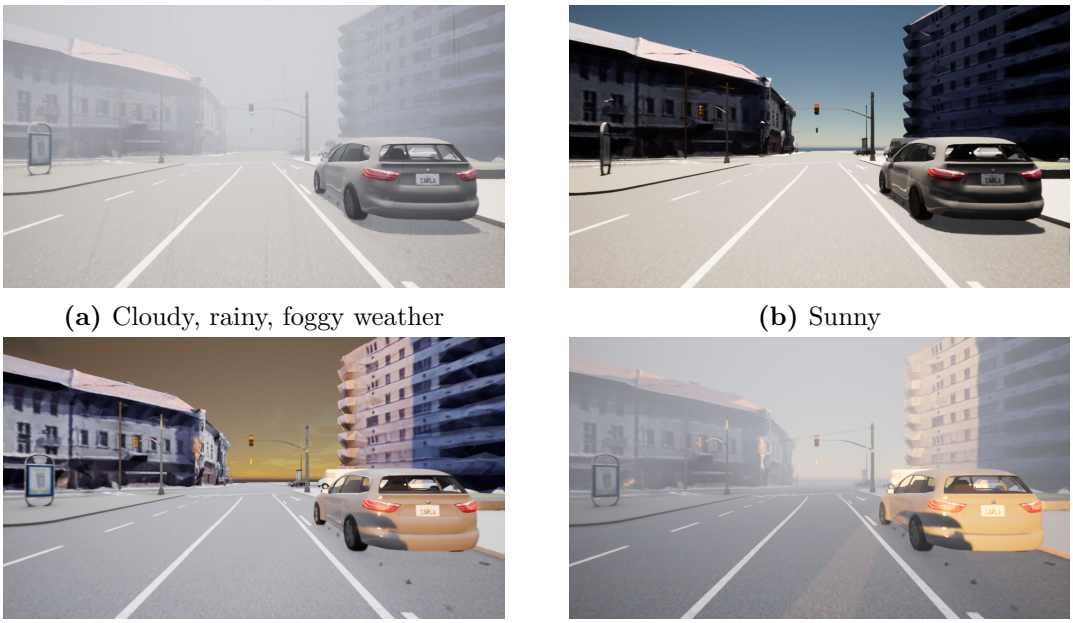
- *Level 2: Infrastructure*

  At this level, situation-dependent adaptions are added to the basic road: not only the traffic guidance infrastructure but the static objects (buildings, trees, and other street objects) are defined here.

- *Level 3: Dynamic objects*

  The quantity and the behavior of the dynamic elements are defined at this level. The behavior and the property can be defined on various abstraction levels, as we described earlier, our workflow contains abstract situations and concrete scenarios too.

- *Level 4: Environment conditions*



(a) Cloudy, rainy, foggy weather

(b) Sunny

(c) Cloudy sunset

(d) Foggy sunset

Figure 3.9: Weather conditions in Carla

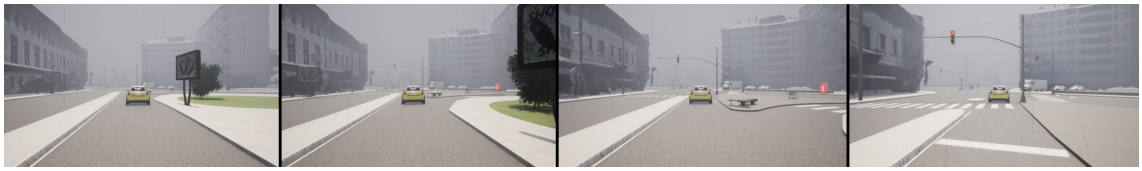


Figure 3.10: Execution

Environment conditions can make big differences, even when the first 3 level is the same, driving in a dark, rainy weather is completely different from a sunny day driving, not only visually, but the physics of the track can also change (e.g. less traction in rain)

- *Extra level*

  Some publications [9, 2, 8] added a new level (2.5 in our context) between infrastructure and dynamic object levels, as the temporary modification of the first two levels. We only mutated the static objects: the mutations could be considered as an extra level, but this covers less than the additional level in [9] or in [2]

Chapter 4

Technological overview

4.1 Workflow in operation

In this section we present the steps of our workflow with the corresponding technologies and tools.

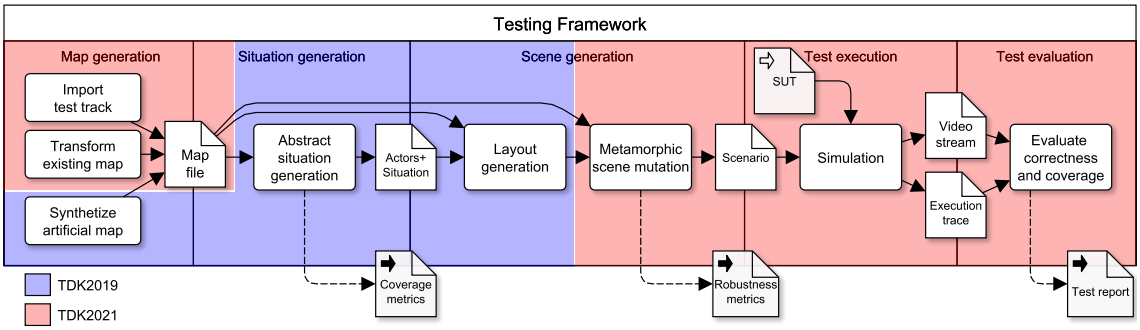


Figure 4.1: Testing workflow

4.1.1 Map generation

With RoadRunner and the further presented tools, it is possible to create an accurate representation of a real world location with little effort. There are solutions for end-to-end generation of virtual environment based on video footage of a real location, but none of them are available for the public.

For the presented solution we used OpenDRIVE map, generated from OpenStreetMap, but customized later. The 3D model of buildings was extracted from Google Maps. We pruned the redundant elements of the model in Blender, to get only the buildings, separately. We constructed the map and the environment in RoadRunner. We used fence, pole, traffic lights and other assets too, to enrich the base environment. After the map ingestion in Carla, in Unreal Editor, we also added some extra feature to the environment: vegetation, and some minor modification, to run the simulation smoothly. This is only the base environment, after the mutation more static objects are in the map, but for the abstract situation and scenario generation it is irrelevant, as it only needs the OpenDRIVE map. The buildings and other static objects are only for the simulator.

4.1.2 Abstract situation generation

The generation of abstract situation is based on the metamodel in Figure 3.4. In our case, it only contains vehicles, but it can handle other dynamic objects (e.g.: pedestrians) or static objects. For the behavior we only used a lane following behavior, for the sake of simplicity.

4.1.3 Scene generation

Layout An instance of the abstract situation, in the OpenDRIVE map gives the layout of the vehicles. This is described in a scenic file, but each vehicle has it's concrete position, and blueprint.

Mutation Robustness test requires multiple similar inputs with small changes, so we created many environment mutations of our map, using Scenic and Carla's python API. The idea was, that the base of the map remains the same (road structure, buildings, some vegetation), but we place random objects at non-disturbing places (at least it is not supposed to disturb the ADAS module). We generated many Scenic code, which spawned a various number of different objects in the map. The spawning region was everywhere except the roads (sidewalks, medians, etc..). The spawning inside a region is based on Scenic's random sampling, after each execution the same code places the same number of objects at different locations. To make each execution reproducible we saved the location and the blueprint of the objects, also in a Scenic code.

We also created different environment conditions, with different lighting, fog, rain and sky parameters. This can be configured at the start of every simulation.

Example 6. *Generated scenic snippet: Spawns 3 ATM next to the road by 0.5 to 1.5 meter.*

```
for i in range(3):
  roads = Uniform(*network.roads)
  if i < 1:
    spawnPoint = OrientedPoint in roads.rightEdge
  else:
    spawnPoint = OrientedPoint in roads.leftEdge
  obj = ATM right of spawnPoint by Range(0.5,1.5) #right of: relative to orientation of OrentedPoint
     in the edge of roads
  props.append(obj)
```

Scene After the initial layout of the vehicles, and the mutation of static objects were constructed, we combined them in every possible way. Note, that in our work the generation of the static and dynamic objects of the scene was independent from each other, both methods required only the OpenDRIVE layout of the map.

Example 7. *Concrete scene: contains vehicles and static objects as well. The orientation is not specified in code, but it is determined by location.*

```
ego = Car at (-80.76853338265082 @ 25.08448837624833),
  with blueprint "vehicle.nissan.patrol"
id0 = Garbage at (-56.713260650634766 @ -0.6789436936378479 ),
  with blueprint "static.prop.garbage02"
id1 = Bench at (64.298583984375 @ -57.030330657958984 ),
  with blueprint "static.prop.bench03"
```

Executing the concrete scene above would result in all vehicles being stationary, since we have not defined their behavior. This can be as simple or complex as we would like, and

can also be generated as mentioned in Section 3.2.2. However, in our tests we used a simple behavior, where all vehicles follow the lane they are placed on, at a speed specified as a parameter.

Example 8. *Concrete scene extended with vehicle behaviors: the vehicle has a behavior defined, which will be executed during the simulation.*

```
ego = Car at (-80.76853338265082 @ 25.08448837624833),
  with blueprint "vehicle.nissan.patrol",
  with behavior FollowLaneBehavior(30)
id0 = Garbage at (-56.713260650634766 @ -0.6789436936378479 ),
  with blueprint "static.prop.garbage02"
id1 = Bench at (64.298583984375 @ -57.030330657958984 ),
  with blueprint "static.prop.bench03"
```

Test execution Scenic can connect to Carla as a client, can load scenes into the simulator and can also execute the behavior assigned to the vehicles. However, it is not capable of everything we needed, e.g. it cannot save sensor data by default, so we had to create our own client to do so. Since Scenic is open-source, we used it as the base of our program, and extended it with further functionality. This includes the ability to save images from virtual dash cams, to change the elapsed time between simulation steps, and to change the weather.

Example 9. *A typical command with arguments to run our modified Scenic.*

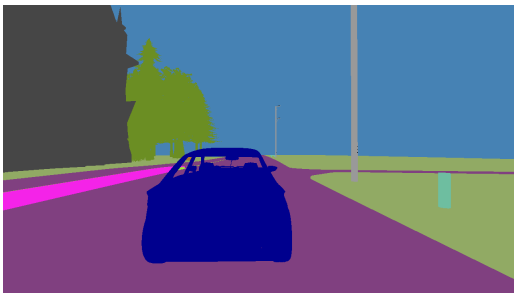
```
python .\scenic.py .\input\scene.scenic --simulate --count 1 --timestep 0.1 --samplingrate 10
    --time 50 --skip 20 --weather 0 60 30 55 --output .\output -p render 0
```

Let's see what all the arguments mean in Example 9:

- `.\input\scene.scenic`: path to the input file
- `--simulate`: execute the simulation in the simulator specified in the input file
- `--count 1`: only run the simulation once
- `--timestep 0.1`: set the delta time between steps to 0.1 seconds
- `--samplingrate 10`: only save the images in every 10th step
- `--time 50`: run the simulation for 50 steps
- `--skip 20`: in the first 20 steps no images will be saved
- `--weather 0 60 30 55`: set the following weather parameters: fog=0%, cloud=60%, rain=30%, sun-angle=55°
- `--output .\output`: path to the output directory for the images
- `-p render 0`: do not show a live image of the simulation

Once Scenic loaded and interpreted the input file, it connects to Carla. At this point we set the simulator to synchronous mode, which means now it will only calculate the next step in the simulation once we tell it to. This allows us to retrieve all the necessary sensor data, like the camera images between each step. We found that with relatively short executions, where we don't need to save more than a few thousand images it is best to keep them in memory at this point. This is also the time when we can execute the

(a) RGB image from Carla



(b) Semantic segmentation image from Carla

behavior of the actors. Scenic checks if any conditions meet the criteria for intervention in any of the defined behaviors, and if necessary, it tells the affected actors what action to take in the next step.

Once the simulation reaches the desired number of steps, we can access all the saved images and write them to disk. In our case, we have two different cameras set up in the position of a dashcam on a vehicle. One of them is a standard RGB camera with a horizontal field of view of 90°, shutter speed of 1/200, and image size of 1280x720 pixels. The other camera is a semantic segmentation camera, which with the same parameters. This camera classifies every object in sight by displaying it in a different color according to its tags, as seen in Figure 4.2b (e.g., building in a different color than vehicles). This gives us the ground truth needed for the evaluation of an object detection AI.

4.2 OpenStreetMap

OpenStreetMap is a free and open source world map by OpenStreetMap contributors. OpenStreetMap contains many and detailed information about roads, it is a really good starting point for the simulation road network, even though due to the crowdsourcing data collection, it contains some incorrectness. OpenStreetMap also contains simple 3D models of the buildings, it is good for the environment building.

It is easy to export the desired area and the exported file can be converted to OpenDrive format using CARLA's python package. With some georeference modification, it can be imported in RoadRunner too. 3D models can be extracted as well, they need an additional software[1]. It will generate an .obj file, which can be modified by Blender[2]

4.3 OpenDRIVE

OpenDRIVE is a road network description standard by ASAM, it describes the structure of the roads in an XML format. The standard can describe roads, lanes, junctions, traffic signs, traffic lights and even railways, switches and many more. It is supported (at least partially) by the tools we are working with: Roadrunner, Scenic, CARLA.

Usage OpenDRIVE is able to describe complex maps, it can be generated or modified by many tools, including RoadRunner, Carlas python package or manual modification of the xml file is also possible. We used OpenDRIVE files in every step of our workflow.

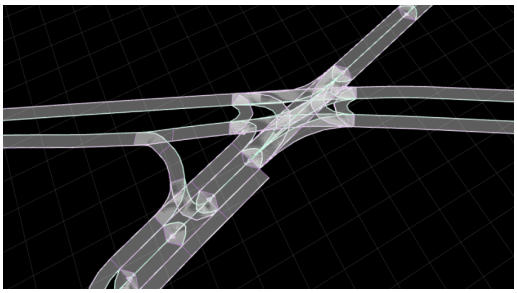
[1] OpenStreetMap to 3D .obj tool and docmentation: `http://osm2world.org/`

[2] `https://www.blender.org/`

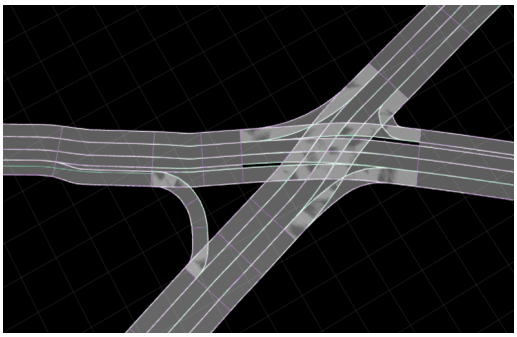
4.4 RoadRunner

RoadRunner is a road network and environment design software by VectorZero, with a powerful building and editing toolset, widely supported import and export formats and many good customizable assets.

Road network To build a real world environment for the scenario generation and for the simulation in Carla, the OpenDRIVE file generated from OpenStreetMap must be edited and for the realistic appearance the static objects also need to be placed down. As figure 4.3 shows, the OpenDRIVE file generated from OpenStreetMap significantly differs from the reality: lanes are missing, road annotations and properties are not correct and junction paths are sometimes faulty, but these are easy to fix with RoadRunner. The conversion from .osm to .xodr format is far from lossles, the generated OpenDRIVE file does not contain railway lanes, road markings and signs[3]. Scenic only requires the OpenDRIVE file, but the simulation environment CARLA also needs a triangle-mesh file (.fbx extension) for Unreal engine, to make the simulation photorealistic. These can be easily exported in the required formats. Other export formats are also available for Unreal, Unity and others.



(a) Original OpenDRIVE file, generated directly from OpenStreetMap



(b) Manually modified, realistic OpenDRIVE file

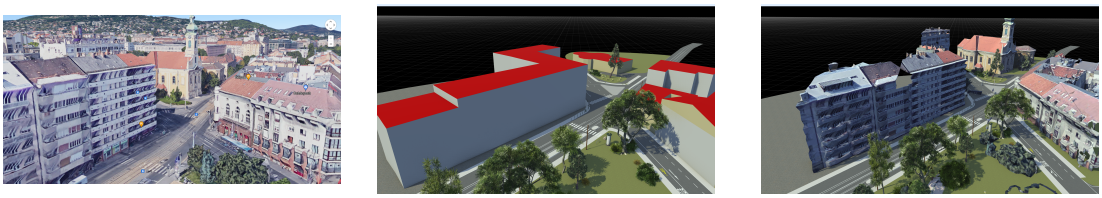
Figure 4.3: Visualized by OpenDRIVE viewer `https://sebastian-pagel.net/`

With RoadRunner everyone can build complex environment including buildings, trees, signs and other static objects. To make the scenes realistic in the simulation these objects are important, a lot of common street objects are already implemented, but everyone can make new ones or customize the existing ones, even importing 3D objects or pictures and using them is possible.

3D buildings There are two ways to import real word buildings into RoadRunner: from OpenStreetMap we can extract[4] the ground layout and the height of the buildings. 3D buildings from google maps also can be extracted[5]. In the 3D model extracted from Google Maps the buildings are geometric accurate, they also have texture, but these 3D models have some disadvantages even though it is more realistic: it can not be used for further environment mutations as it is too detailed. Furthermore, the extracted 3D object does not separate the buildings, the model contains the trees, bushes, even the traffic signs and they are distracting, because they bend into the roads and the texture is too blurry. Although RoadRunner can handle .obj files, both importing methods requires manual afterwork in

[3]The supported OpenDRIVE elements in Carla: OpenDRIVE 1.4
[4]OpenStreetMap to 3D .obj tool: `http://osm2world.org/`
[5]Tool and description: `https://github.com/eliemichel/MapsModelsImporter`

Blender[6], before it can be imported to RoadRunner. For the OpenStreetMap approach to separate and export only the interesting buildings. The google maps approach needs a Blender extension, to transform the extracted google maps capture file (.rdc) to .fbx, and also the building separation, and the pruning of redundant parts needs to be done in Blender.



(a) Original picture from google maps

(b) Buildings imported from OpenStreetMap

(c) Buildings imported from google maps

Figure 4.4: Comparasion of the buildings

4.5 Scenic

Scenic [3, 15] is a probabilistic programming language designed for modeling the environments of cyber-physical systems such as robots and autonomous cars. A Scenic program specifies a distribution of scenes, configurations of physical objects and agents; sampling from this distribution produces concrete scenes that can be simulated to generate training or testing data. An open-source compiler and scenario generator for the Scenic scenario description language is available under BSD 3-Clause License.

4.5.1 Supported Simulators

Scenic is designed to be easily interfaced to any simulator. To interface Scenic to a new simulator, there are two steps: using the Scenic API to compile scenarios and generate scenes, and writing a Scenic library defining the virtual world provided by the simulator. Each of the simulators natively supported by Scenic has a corresponding model.scenic file containing its world model.

CARLA The CARLA simulator's interface allows Scenic to be used to define autonomous driving scenarios. Dynamic scenarios written with the CARLA world model as well as scenarios written with the cross-platform Driving Domain are supported by the interface. In this report, we used this simulator.

Grand Theft Auto V The interface to Grand Theft Auto V allows Scenic to position cars within the game as well as to control the time of day and weather conditions. Importing scenes into GTA V and capturing rendered images requires a GTA V plugin.

LGSVL The LGSVL interface supports dynamic scenarios written using the LGSVL world model as well as scenarios using the cross-platform Driving Domain.

[6] `https://www.blender.org/`

Webots There are several interfaces to the Webots robotics simulator, for different use cases.

An interface for a Mars rover example. This interface is extremely simple and might be a good baseline for developing our own interface. A general interface for traffic scenarios. A more specific interface for traffic scenarios at intersections, using guideways from the Intelligent Intersections Toolkit.

X-Plane The interface to the X-Plane flight simulator enables using Scenic to describe aircraft taxiing scenarios. This interface is part of the VerifAI toolkit.

4.6 CARLA

CARLA [1, 12] is a free and open-source self-driving simulator licensed under the MIT License. It was created from the ground up to serve as a modular and adaptable API for addressing a variety of tasks related to autonomous driving. One of CARLA's major aims is to assist democratize autonomous driving research and development by providing a platform that anyone can readily use and configure.

4.6.1 The simulator

A scalable client-server architecture underpins the CARLA simulator.

The server is in charge of everything connected to the simulation itself, including sensor rendering, physics calculation, world-state and actor updates, and much more. Because it aims for realistic results, running the server with a dedicated GPU is the best option, especially when working with machine learning.

The client side is made up of a collection of client modules that govern the logic of actors in a scene and define world conditions. This is accomplished by utilizing the CARLA API (in Python or C++), a layer that acts as a middleman between the server and the client and is constantly growing to include new features.

The simulator's core structure is summarized in this way. CARLA, on the other hand, is much more than that, as it contains many various characteristics and elements. Some of these are listed here to give you an idea of what CARLA is capable of.

- **Traffic manager.** A built-in system that controls all cars other than the one being used for learning. It serves as a conductor provided by CARLA to recreate urban-like environments with realistic behavior.
- **Sensors.** Vehicles rely on them to provide information about their surroundings. They are a specific type of actor in CARLA attached to the vehicle, and the data they receive can be collected and stored to make the process easier. Various sorts of these are currently supported by the project, including cameras, radars, lidar, and many others.
- **Recorder.** This feature is used to reenact a simulation step by step for every actor in the world. It grants access to any moment in the timeline anywhere in the world, making for a great tracing tool.

- **ROS bridge and Autoware implementation.** The CARLA project ties knots and seeks to integrate the simulator into different learning environments as a matter of universalization.
- **Open assets.** CARLA offers a variety of maps for urban settings, as well as weather control and a blueprint library with a large number of actors to choose from. However, these elements can be modified and new ones may be created using simple guidelines.
- **Scenario runner.** CARLA provides a variety of routes outlining distinct conditions to iterate on to make the learning process for vehicles easier.

4.6.2 World and client

The user runs the client module to request information or changes in the simulation. A client is identified by an IP address and a port number. It uses a terminal to talk with the server. Many clients may be active at the same time. Advanced multiclient management requires thorough understanding of CARLA and synchrony.

The world is an object representing the simulation. It serves as an abstract layer that contains the primary methods for spawning actors, changing the weather, obtaining the current state of the environment, and so on. Each simulation has just one world. When the map is changed, the world will be destroyed and replaced with a new one.

4.6.3 Actors and blueprints

Anything that takes part in the simulation is referred to as an actor.

- Vehicles
- Pedestrians
- Sensors
- The spectator
- Traffic signs and traffic lights

Blueprints are pre-made actor layouts that are required to spawn an actor. Models with animations and a set of properties, in a nutshell. Some of these attributes are customizable by the user, while others are not. A blueprint library is provided, which contains all the blueprints as well as information about them.

4.6.4 Maps and navigation

A map is an object that represents the simulated world, primarily the town. By default, there are eight maps available. To describe the roadways, they all use the OpenDRIVE 1.4 standard.

Roads, lanes, and junctions are managed by the Python API to be accessed from the client. These are used along with the waypoint class to provide vehicles with a navigation path.

Traffic signs and traffic lights are accessible as carla.Landmark objects that include information about their OpenDRIVE definition. When running using the information in the OpenDRIVE file, the simulator also creates stops, yields, and traffic light objects automatically. These have bounding boxes placed on the road. Vehicles become aware of them once inside their bounding box.

4.6.5 Sensors and data

Sensors wait for some event to happen, and then gather data from the simulation. They require a function that specifies how to manage the data. Sensors get different forms of sensor data depending on their type.

A sensor is an actor attached to a parent vehicle. It follows the car around collecting data about its surroundings. The sensors available are defined by their blueprints in the Blueprint library:

- Cameras (RGB, depth, semantic segmentation, and optical flow)
- Collision detector
- Gnss sensor
- IMU sensor
- Lidar raycast
- Lane invasion detector
- Obstacle detector
- Radar
- RSS

Chapter 5

Evaluation

5.1 Resarch questions

We evaluated our approach by formulating various research questions and answering them by measuring execution times as well as by carrying out benchmarks. Our questions can be grouped into two categories. The first one focuses on the performance of the tools used in our workflow, while the second one targets the result of our work, i.e. whether or not our approach to robustness testing of ADAS modules is viable. These are the research questions we aim to answer:

RQ1 What is the performance of the proposed test generation tool?

- **RQ1.1** What is performance of the proposed scene generation approach?
- **RQ1.2** What is the performance of the image synthesis tool?

RQ2 What is the quality of the generated test cases?

5.2 Selected domain

Our goal was to create a realistic testing environment, so we chose an intersection in Budapest, with diverse traffic opportunities. The area is 230*170m, and contains 3 real road. The OpenDrive representation contains 27 roads, with 163 lanes and 4 junctions total. The surrounding environment is also diverse: there are classic, and modern buildings and also a wide variety of vegetation, around the roads.

5.3 Scene generation performance

5.3.1 Measurement setup

During the static object generation, we generated 10 different high-level Scenic files, these files only described the number and the type of the static objects. It placed the objects next to the roads, all around the map. The number of objects was between 33 and 40. We also generated the layout of the dynamic objects, then combined the layout of the static and dynamic objects in every possible way. We measured the runtime of the layout generation on an average personal computer (GPU: Nvidia GeForce gtx 1660 super, CPU: AMD Ryzen 5 3600x, Memory: 16 GB, OS: Windows 10).

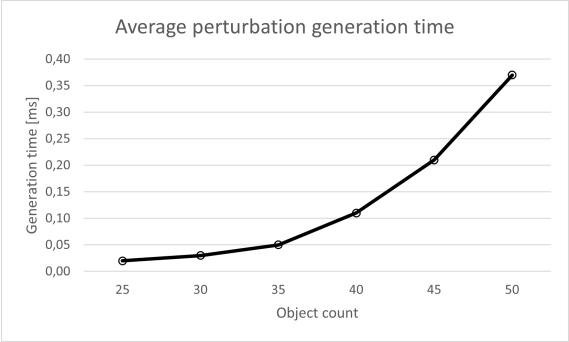


Figure 5.1: Perturbation generation runtime

5.3.2 Measurement results

In Figure 5.1 Y axis shows the average runtime of 25 different perturbation generation. X axis shows the number of static objects to spawn. The average execution (and capture) time of the high-level Scenic files was 80 ms. The fastest was 13 ms, the slowest 174 ms, due to the random sampling of Scenic. We only measured the performance of the concrete scene generation, the performance of the abstract scene generation can be found in [14].

5.3.3 Discussion of the results

Generating the scenes, including the perturbations of the static objects, is a relatively quick process, until the number of objects is limited. Typically, the number of static objects was between 33 and 40 in our case.

The runtime looks like a quadratic function, because of the Scenic's sampling: it has to allocate X objects in a fixed size map. If a randomly chosen spawnpoint is already taken (the hitbox of objects can't collide), it has to pick another random point.

RQ1.1: The scene generation process is relatively quick, but the runtime of perturbation generation grows quadratically based on the number of static objects.

Over 50 objects the map was too crowded, the ideal number of objects was below 40, therefore we continued the measurements with this many objects.

5.4 Image synthesis performance

5.4.1 Measurement setup

The measurements for the simulation phase were executed on a desktop PC with an AMD FX(tm)-8350 8-Core 4GHz processor, an NVIDIA GeForce GTX 750 Ti GPU and 16 GB of memory, running Windows 10 Pro. The measurements involving writing data to disk were executed using a Samsung 850 EVO 250GB SSD.

For all measurements relating to **RQ1.2**, we used scenes with three actors, all of them executing a simple behavior following the lane they are initially placed on, and 33 additional static objects, since this provided the most realistic scenery. The dimensions of all images created and saved during the measurements were 1280x720 pixels.

To answer **RQ1.2**, we measured multiple aspects, which required two different setups.

5.4.1.1 Measurement setup I

In the first setup we simulated 50 different scenes, first for 100 steps, then for 500 steps and lastly for 1000 steps, and measured the runtime. In order to eliminate noise, we then took the median of the results from the 50 different runs. During all these measurements there were two cameras attached to one of the three vehicles.

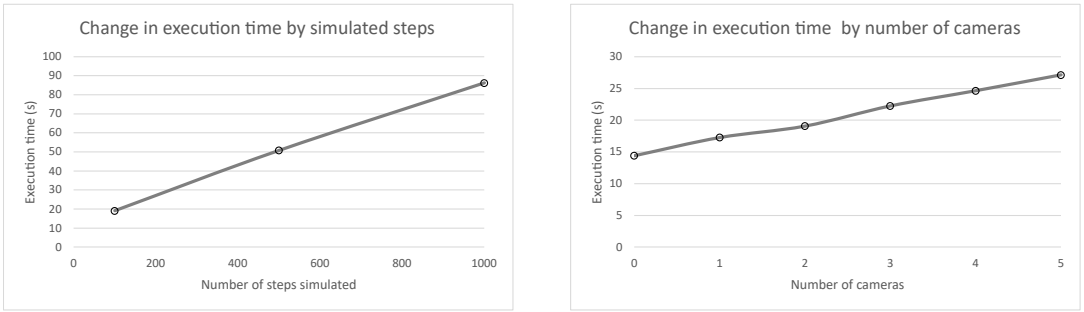
5.4.1.2 Measurement setup II

In the second setup we used the same representative 50 scenes as before, but always simulating them for 100 steps. We measured the runtime of the simulation during the executions of the scenes, and took the median to eliminate noise.

5.4.2 Measurement results

For **RQ1.2**, first we measured how the number of steps calculated in the simulation affects the execution time of the simulator using **Measurement setup I.** These results are shown in Figure 5.2a, and it is clear, that as the number of steps increases, the execution time increases linearly. At 100 steps it takes 19.07 seconds, which is 5.24 steps every second. At 500 steps the execution takes 50.74 seconds, which means 9.85 steps per second, and finally, at 1000 simulated steps, the runtime is 86.23 seconds, which is 11.6 steps calculated every second.

We also measured how the execution time changes when we increase the number of cameras attached to vehicles. These were measured with **Measurement setup II.** These result can be seen in Figure 5.2b, and this also shows a linear increase in runtime. With zero cameras attached to vehicles the simulation took 14.41 seconds, and with five cameras this time nearly doubled to 27.14 seconds.



(a) **(b)**

Figure 5.2: Runtime of the simulation

Lastly, we measured how long it takes to save the images from the virtual cameras to disk. These results are shown in Figure 5.3, with the number of saved images on the horizontal axis and the time it took to save them on the vertical axis. There are two types of images we saved, one is images from a regular RGB camera, and the other is a semantic segmentation image. This time includes not just writing the images to the disk, but also some processing done by Carla. Saving a single RGB image took 0.66 seconds, while saving a semantic segmentation image took 0.56 seconds. At 1000 images the times are almost 1000 times that, 641.84 seconds and 555.89 seconds respectively.

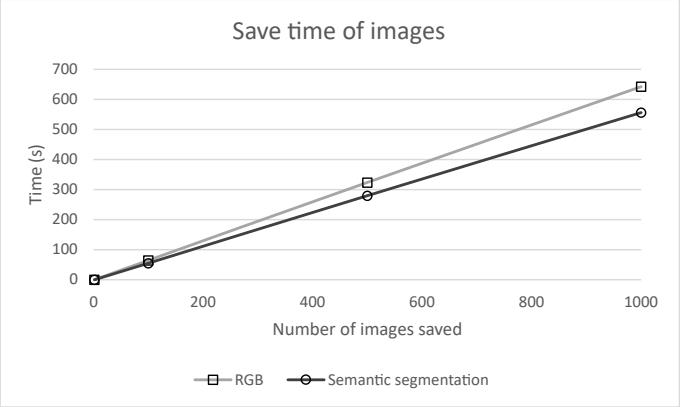


Figure 5.3: Save time of images

5.4.3 Discussion of the results

All three measured aspects affect the runtime of the simulation linearly. This means that the required time for a simulation is easily predictable. However, it is clear, that in our setup the bottleneck was the final step, processing and saving the images provided by Carla. Here the difference between the RGB and semantic segmentation images may be explained by the size difference. While the average size of the RGB images was around 1.2 MB, the semantic segmentation images averaged at around 34 KB, since they can be compressed much more efficiently thanks to only containing large blocks of just a few colors.

Moreover, the image generation can be executed parallelly on multiple computers.

RQ1.2: The image synthesis takes a reasonable time, but saving the images to disk can be a bottleneck.

5.5 Test quality evaluation

5.5.1 Measurement setup

To evaluate the qualitiy of the generated images as test cases, we used the generated test suite to measure the accuracy of an existing well-known image recognition AI model. For this purpose, we downloaded used the Mask R-CNN model [16], and we used the most suiting COCO-InstanceSegmentation/mask_rcnn_R_50_FPN_3x configuration. The AI model was trained on annotated real-world images, but it can be tested with generated images as well. The task of the AI was to determine whether the image contained a *vehicle* or a *traffic light*.

We generated two datasets, one which is intended to be robust, and one which can be described as random. The robust dataset was created in the following way. First, we generated five different abstract situations involving three or four vehicles. Then all actors were assigned their coordinates. In the next step each scene was combined with ten different layout of 33 static objects, resulting in 50 different scenes. All of them were then executed in Carla with 16 different weather conditions. We take the following possible values for the fog, cloud, rain and sun settings (fog, cloud, rain can be any float between 0 and 100, sun angle ranges from -90° to 90°), and use all possible combination of them:

- Fog: 0, 30

- Cloud: 0, 90
- Rain: 0, 90
- Sun: 5°, 90°

This resulted in 800 simulations, and from each of them we extracted seven RGB images from the location of a dash cam placed in the Ego vehicle, and seven corresponding semantic segmentation images. The time difference between the images was 1.4 seconds. In total we got 5600 pairs of images.

For the random dataset we ran 800 simulations, in each of them we placed 10-15 vehicles in random places on the road, changed all four weather parameters to a random value between the previously used values. From each simulation we extracted seven images, with 0.2, 0.5 or 2 seconds between them. This resulted in a diverse set of 5600 images.

5.5.2 Measurement results

The results of the object recognition tests are in Table 5.1. The AI correctly recognized the vehicles in 96.38% of the random dataset, and in 97.89% of the robust dataset. In terms of recognizing traffic lights, we can see a larger difference. While the AI achieved a 96.9% accuracy on the random dataset, it only reached 73.28% accuracy on the robust dataset.

Data set	Vehicle	Traffic light
Random	0.9638	0.9690
Robust	0.9789	0.7328

Table 5.1: Accuracy of object detection on generated images

5.5.3 Discussion of the results

As we can see, there is a significant difference between the accuracy achieved on the two datasets in terms of recognizing traffic lights, but not for the vehicles. After taking a closer look at the images where the AI was incorrect, we did not find a clear correlation between the errors and the weather in the images. This suggests, that simply introducing small changes into the images can alter the results of the image recognition AI.

RQ2 As expected, the pretrained model performed better traffic light detection in the random dataset, but for the vehicles the accuracy was almost the same, compared to the robust dataset.

5.5.4 Threats to validity

Even though the images are not the best quality, due to the buildings with blurry, low resolution texture, the object detection accuracy is similar to real world based image datasets. We used a publicly available pre-trained Mask R-CNN model, with models trained on our own dataset, the results could be other.

Chapter 6

Related works

6.1 Testing approaches

Testing deep or machine learning based systems is a heavily researched field: the internal working of these systems are not interpretable for humans, and formal verification is practically impossible, especially for models with many parameters.

A concerning property of ML/DL based systems is the lack of robustness: small changes at the input can drastically change the systems behavior. For detecting these kinds of failures, metamorphic testing is a good method, as it does not need an oracle, just similar inputs and the model, to compare the results.

Finding the challenging inputs is possible by not only randomly mutate or transform the inputs. Some metrics can describe the difficulty of the decision-making for neural networks (e.g.: neuron coverage: higher neuron coverage usually means that the prediction is less reliable.). Optimizing the input for larger neuron coverage could fool a DL-based systems.

- DeepTest [24] adds adversarial noise (e.g. rain, fog) to a picture:

Figure 6.1: System fails to detect the correct path [24]

- The approach to vision-based ADAS testing by Lionel Briand in [6] is similar to ours: they used abstract metamodel of the scenario to generate test cases. They constructed the test scenarios for AEB testing, but the method can be generalized for every other ADAS. The test generation method is not random, it is guided by classification decision trees, this way it can generate more threatening scenarios, and show critical input feature groups.

Semantic mutation of the input creates semantically similar inputs, so the system should behave the same.

- MetaOD [27] provides a domain specific background mutation method, which can trigger object detection failure. MetaOD extracts the features (object types) of the original picture, then based on these features it inserts an appealing object chosen from an object pool. The placement is not random, it is close to existing objects, this way it is more likely to trigger an error.

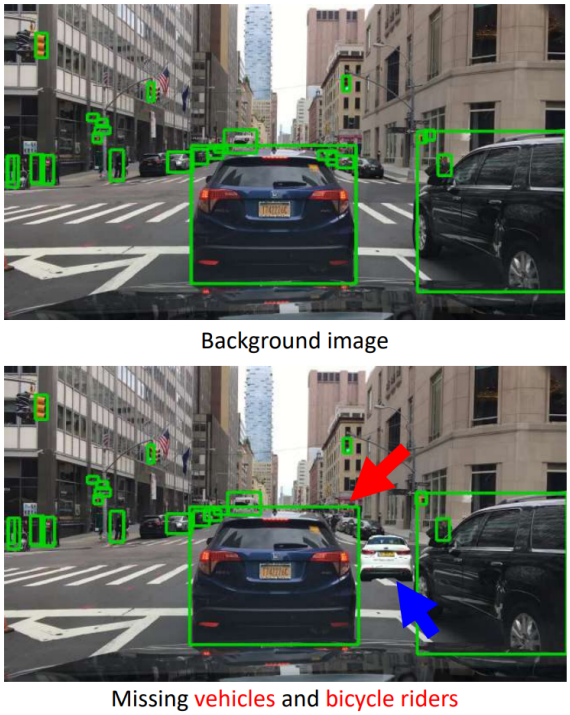


Figure 6.2: detection error at mutateted picture [27]

Chapter 7

Conclusion and Future Works

Testing AI-based components in autonomous vehicles is challenging since they must operate in an immensely complex and rapidly changing environment. Here, traditional software test generation and coverage-based approaches (e.g., symbolic execution or concolic testing) fail to provide useful test cases. On the other hand, test drives and tracks are expensive compared to computer-based testing and fail to cover dangerous situations (which might be the most important to check). Therefore, simulator-based testing approaches are getting wider use recently. In this report, we proposed a simulator-based testing environment for vision-based autonomous driving components. The key elements of our work are the following:

- We proposed a map generation approach that uses publicly available data of real world locations, this way we can reproduce any real urban environment, to run the simulations in a realistic environment.
- We developed a method to perturb the scenes by changing weather effects and location of static objects, to generate test cases for robustness testing.
- We integrated the simulator with our other tools in our workflow: We successfully ingested our maps in the simulator through Unreal Engine Editor, and we used Scenic to describe and run the scenes in CARLA.
- We evaluated the generated test cases on existing object-recognition AI models, which showed that such models perform worse in a perturbed environment.

Future works The semantic mutation of objects was based on random sampling, in the future more semantic constraints could be included to create a more realistic environment. In the future, we could also propose an *adversarial testing approach* by measuring threat metrics (based on the level of danger caused by the decision of an AI model under test) to optimize the perturbation, and to generate more threatening environments.

Additionally, we are planning to adapt and evaluate our approach to test real-life industrial AI applications developed for railway systems.

Acknowledgments. We would like to thank Aren A. Babikian for his contribution in the modeling and generation of situations, and Boqi Chen for his help in the evaluation of the test cases. We would also like to thank Arrowhead Tools project: This research was funded by the European Commission and the Hungarian Authorities (NKFIH) through the Arrowhead Tools project (EU grant agreement No. 826452, NKFIH grant 2019-2.1.3-NEMZ ECSEL-2019-00003).

Finally, we would like to thank our advisors, dr. András Vörös and dr. Oszkár Semeráth, for all the help and guidance in the past months.

Bibliography

[1] Carla documentation. https://carla.readthedocs.io/. Accessed: 2021-10-28.

[2] Safety First for Automated Driving. page 157.

[3] Scenic documentation. https://scenic-lang.readthedocs.io/. Accessed: 2021-10-28.

[4] Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles, 2018.

[5] Road vehicles – functional safety, 2018.

[6] Raja Ben Abdessalem, Shiva Nejati, Lionel C Briand, and Thomas Stifter. Testing vision-based control systems using learnable evolutionary algorithms. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 1016–1026. IEEE, 2018.

[7] Aren A Babikian, Oszkár Semeráth, Anqi Li, Kristóf Marussy, and Dániel Varró. Automated generation of consistent models using qualitative abstractions and exploration strategies. *Software and Systems Modeling*, pages 1–25, 2021.

[8] G Bagschik, T Menzel, C Korner, and M Maurer. Wissensbasierte Szenariengenerierung für Betriebsszenarien auf deutschen Autobahnen. page 14.

[9] Gerrit Bagschik, Till Menzel, and Markus Maurer. Ontology based scene creation for the development of automated vehicles. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 1813–1820. IEEE, 2018.

[10] Antonia Breuer, Jan-Aike Termöhlen, Silviu Homoceanu, and Tim Fingscheidt. opendd: A large-scale roundabout drone dataset. In *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*, pages 1–6. IEEE, 2020.

[11] Krzysztof Czarnecki. On-road safety of automated driving system - taxonomy and safety analysis methods. Technical report, U. of Waterloo, 2018.

[12] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. In Sergey Levine, Vincent Vanhoucke, and Ken Goldberg, editors, *Proceedings of the 1st Annual Conference on Robot Learning*, volume 78 of *Proceedings of Machine Learning Research*, pages 1–16. PMLR, 13–15 Nov 2017. URL https://proceedings.mlr.press/v78/dosovitskiy17a.html.

[13] Tommaso Dreossi, Daniel J Fremont, Shromona Ghosh, Edward Kim, Hadi Ravanbakhsh, Marcell Vazquez-Chanlatte, and Sanjit A Seshia. Verifai: A toolkit for the design and analysis of artificial intelligence-based systems. *arXiv preprint arXiv:1902.04245*, 2019.

[14] Attila Ficsor and Balázs Somorjai. Tesztelrendezések automatikus generálása autonóm járművek szisztematikus ellenőrzéséhez. In *Student Research Societies Report*, 2019.

[15] Daniel J. Fremont, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L. Sangiovanni-Vincentelli, and Sanjit A. Seshia. Scenic: a language for scenario specification and scene generation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 63–78, Phoenix AZ USA, June 2019. ACM. ISBN 978-1-4503-6712-7. DOI: 10.1145/3314221.3314633.

[16] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask R-CNN. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969, 2017.

[17] Justin Johnson, Bharath Hariharan, Laurens van der Maaten, Li Fei-Fei, C. Lawrence Zitnick, and Ross Girshick. CLEVR: A Diagnostic Dataset for Compositional Language and Elementary Visual Reasoning. *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, 2017-Janua:1988–1997, dec 2016. URL http://arxiv.org/abs/1612.06890.

[18] Nidhi Kalra and Susan M Paddock. Driving to Safety: How Many Miles of Driving Would It Take to Demonstrate Autonomous Vehicle Reliability? page 15.

[19] Nikhil Ketkar. Introduction to keras. In *Deep learning with Python*, pages 97–111. Springer, 2017.

[20] Robert Krajewski, Julian Bock, Laurent Kloeker, and Lutz Eckstein. The highd dataset: A drone dataset of naturalistic vehicle trajectories on german highways for validation of highly automated driving systems. In *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, pages 2118–2125. IEEE, 2018.

[21] István Majzik, Oszkár Semeráth, Csaba Hajdu, Kristóf Marussy, Zoltán Szatmári, Zoltán Micskei, András Vörös, Aren A Babikian, and Dániel Varró. Towards system-level testing with coverage guarantees for autonomous vehicles. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 89–94. IEEE, 2019.

[22] Till Menzel, Gerrit Bagschik, and Markus Maurer. Scenarios for development, test and validation of automated vehicles. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 1821–1827. IEEE, 2018.

[23] Oszkár Semeráth, András Vörös, and Dániel Varró. Iterative and incremental model generation by logic solvers. In Perdita Stevens and Andrzej Wasowski, editors, *Fundamental Approaches to Software Engineering*, pages 87–103, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. ISBN 978-3-662-49665-7.

[24] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. DeepTest: Automated Testing of Deep-Neural-Network-driven Autonomous Cars. *arXiv:1708.08559 [cs]*, March 2018. URL http://arxiv.org/abs/1708.08559. arXiv: 1708.08559.

[25] Simon Ulbrich, Till Menzel, Andreas Reschka, Fabian Schuldt, and Markus Maurer. Defining and substantiating the terms scene, situation, and scenario for automated driving. In *2015 IEEE 18th International Conference on Intelligent Transportation Systems*, pages 982–988. IEEE, 2015.

[26] Walther Wachenfeld and Hermann Winner. *The Release of Autonomous Vehicles*, pages 425–449. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016. ISBN 978-3-662-48847-8. DOI: 10.1007/978-3-662-48847-8_21.

[27] Shuai Wang and Zhendong Su. Metamorphic object insertion for testing object detection systems. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1053–1065. IEEE, 2020.

[28] Hermann Winner, Günther Prokop, and Markus Maurer, editors. *Automotive Systems Engineering II*. Springer International Publishing, Cham, 2018. ISBN 978-3-319-61605-6 978-3-319-61607-0. DOI: 10.1007/978-3-319-61607-0.

[29] Kexin Yi, Jiajun Wu, Chuang Gan, Antonio Torralba, Pushmeet Kohli, and Joshua B. Tenenbaum. Neural-symbolic vqa: Disentangling reasoning from vision and language understanding. In *NeurIPS*, 2018.