DIPLOMATERVEZÉSI FELADAT

Ficsor Attila
Mérnökinformatikus hallgató részére

Nagyhatékonyságú predikátum kiértékelő illesztése gráfgenerátor algoritmushoz

Kritikus rendszerek tervezésére széles körben alkalmaznak modellezőeszközöket, amelyek gráf alapú modellek felhasználásával teszik automatizálhatóvá és ellenőrizhetővé a fejlesztés számos lépését. Azonban, mint minden szoftver, maguk a modellezőeszközök is tartalmazhatnak hibákat, melyek érvényteleníthetik az ellenőrzések eredményeit. Ezért kiemelten fontos a modellezőeszközök alapos tesztelése. Továbbá egy modern modellezőeszköznek akár több millió elemből álló modelleket is képesnek kell lennie feldolgozni, ezért egyre inkább elterjedté válik a modellezőeszközök teljesítményének szisztematikus mérése (benchmark).

Szisztematikus teszteléshez és teljesítményméréshez mindenekelőtt egy diverz és realisztikus modellkészletre van szükség, ahol a modellek szolgáltatják a tesztbemeneteket. A tanszékünkön fejlesztett VIATRA Solver keretrendszer célja épp ezért olyan szintetikus modellek automatikus előállítása, amelyek tesztbemenetként szolgálhatnak.

A dolgozat célja, hogy kiegészítse a VIATRA Solver keretrendszert olyan predikátum kiértékelő algoritmussal, amely közvetlenül a VIATRA Solver belső adatszerkezetei fölött végez logikai érvelést. Ezáltal nagymértékben javíthatóvá válik a VIATRA Solver skálázhatósága és hordozhatósága.

A hallgató feladatának a következőkre kell kiterjednie:

- Ismerje meg és mutassa be a predikátum kiértékelés folyamatát a VIATRA Solver keretrendszerben.
- Javasoljon egy olyan architektúrát, amely közvetlenül illeszt egy gráfminta-illesztő rendszert a VIATRA Solver belső adatszerkezeteihez.
- Készítsen eszköztámogatást, amely támogatja predikátumok fejlesztését és kiértékelését.
- Munkáját értékelje, és hasonlítsa össze az elkészült rendszer skálázhatóságát és hordozhatóságát a korábbi megvalósításával.

Tanszéki konzulens: Dr. Semeráth Oszkár, tudományos munkatárs
Budapest, 2021.03.16.

…………………………..
Dr. Dabóczi Tamás
tanszékvezető
egyetemi tanár, DSc

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

1117 Budapest, Magyar Tudósok krt. 2. I. ép. I.E.444.
Telefon: 463-2057, Fax: 463-4112
http://www.mit.bme.hu • e-mail: mitadm@mit.bme.hu

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

Integration of a High Performance Predicate Evaluator to a Graph Generator Algorithm

Master's Thesis

Author
Attila Ficsor

Advisor
dr. Oszkár Semeráth

December 10, 2021

Contents

Kivonat i

Abstract iii

1 Introduction 1

1.1 Graph generation . . . 1
1.2 Critical autonomous systems . . . 1
1.3 Testing the correctness of autonomous components . . . 2
1.4 Graph generation in testing . . . 2
1.5 Pattern matching in the past . . . 3
1.6 Structure of thesis . . . 3

2 Preliminaries 5

2.1 Advanced driver assistance systems . . . 5
2.1.1 Functional overview of autonomous components . . . 6
2.1.2 Fault hypotheses . . . 6
2.1.3 Testing of ADAS and self driving systems . . . 7
2.2 Graph generation . . . 7
2.2.1 Domain-specific languages . . . 7
2.2.1.1 Eclipse Modeling Framework . . . 7
2.2.1.2 New specification language for modeling . . . 8
2.2.2 Pattern matching . . . 10
2.2.2.1 VIATRA Query Language . . . 10
2.2.2.2 Syntax of predicates in the new specification language . . . 12

3 4-valued graph predicate evaluation 15

3.1 4-valued graphs . . . 15
3.2 Creating graph patterns . . . 16
3.2.1 Defining predicates using disjunctive normal form . . . 17
3.2.1.1 Building blocks of predicates . . . 18

3.2.1.2 Predicates referencing relations 18
3.2.1.3 Conjunction of expressions 19
3.2.1.4 Disjunction of conjunctions 20
3.2.1.5 Referencing predicates inside predicates 20
3.3 Pattern matching on custom datastructures 21
3.3.1 Initiating the Rete network 22
3.3.2 Applying changes to the model 22

4 Testing of autonomous vehicles using graph predicates 25
4.1 Scenes and Situations 25
4.2 Functional overview 26
4.2.1 Map generation 26
4.2.2 Situation generation 30
4.2.3 Scene generation 31
4.2.4 Test execution 32
4.2.5 Test evaluation 33
4.3 Testing approaches 33
4.3.1 Metamorphic testing 33
4.3.2 Coverage based testing 35
4.4 Summary of scenario building 36

5 Evaluation 37
5.1 Research questions 37
5.2 Selected domain 37
5.2.1 Patterns in VIATRA 38
5.2.2 Patterns with the new syntax 39
5.3 Measurement setup 40
5.4 Measurement results 41
5.5 Discussion of the results 42

6 Conclusions and future work 45

Acknowledgements 47

List of Figures 49

Bibliography 49

Appendix 57

A.1 Workflow in operation 57
A.1.1 Map generation 57
A.1.2 Abstract situation generation 57
A.1.3 Scene generation 58
A.2 Scenic 60
A.2.1 Supported Simulators 60
A.3 CARLA 61
A.3.1 The simulator 61
A.3.2 World and client 62
A.3.3 Actors and blueprints 62
A.3.4 Maps and navigation 62
A.3.5 Sensors and data 63

HALLGATÓI NYILATKOZAT

Alulírott *Ficsor Attila*, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2021. december 10.

Ficsor Attila
hallgató

Kivonat

Kritikus rendszerek tervezése és tesztelése során széles körben használnak modellezőeszközöket, amelyek gráf alapú modellek fehasználásával teszik automatizálhatóvá és ellenőrizhetővé a fejlesztés és tesztelés számos lépését. 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.

Szisztematikus teszteléshez mindenekelőtt egy diverz és realisztikus modellkészletre van szükség, ahol a modellek szolgáltatják a tesztbemeneteket. A tanszékünkön fejlesztett VIATRA Solver keretrendszer célja épp ezért olyan szintetikus modellek automatikus előállítása, amelyek tesztbemenetként szolgálhatnak.

A dolgozatom célja, hogy kiegészítsem a VIATRA Solver keretrendszert olyan predikátum kiértékelő algoritmussal, amely közvetlenül a VIATRA Solver belső adatszerkezetei fölött végez logikai érvelést. Ezáltal nagymértékben javíthatóvá válik a VIATRA Solver skálázhatósága és hordozhatósága. Az elkészült megoldás előnyeit és használatát egy esettanulmányon mutatom be.

Abstract

During the design and testing of critical systems, modeling tools are widely used, which help make it possible to automate several steps of development and testing with the usage of graph-based models. 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.

First and foremost, a diverse and realistic set of models is necessary for systematic testing, where the models provide the test input. The purpose of the VIATRA Solver framework developed in our department is generating synthetic models, which can serve as test inputs.

My goal is to extend the VIATRA Solver framework with a predicate evaluator algorithm, which executes logical reasoning directly on the internal data structures of VIATRA Solver. With this the scalability and portability of VIATRA Solver can be improved significantly. I will illustrate the benefits and the usage of my solution in a case study.

Chapter 1

Introduction

1.1 Graph generation

Graphs are the natural data structure to represent relational and structural information in many domains, such as knowledge bases, social networks, molecule structures and even the structure of probabilistic models [35]. The ability to generate graphs therefore has many applications; for example, a generative model of molecular graph structures can be employed for drug design [25, 34, 33, 52], generative models for computation graph structures can be useful in model architecture search [50], and graph generative models also play a significant role in network science [48, 5, 32].

The study of generative models for graphs dates back at least to the early work by Erdős and Rényi [20] in the 1960s. These traditional approaches to graph generation focus on various families of random graph models [53, 20, 26, 48, 9, 5], which typically formalize a simple stochastic generation process (e.g., random, preferential attachment) and have well-understood mathematical properties. However, due to their simplicity and hand-crafted nature, these random graph models generally have limited capacity to model complex dependencies and are only capable of modeling a few statistical properties of graphs. For example, Erdős–Rényi graphs do not have the heavy-tailed degree distribution that is typical for many real-world networks.

More recently, building graph generative models using neural networks has attracted increasing attention [40, 25, 34]. Compared to traditional random graph models, these deep generative models have a greater capacity to learn structural information from data and can model graphs with complicated topology and constrained structural properties, such as molecules.

1.2 Critical autonomous systems

In our everyday life we can already see critical autonomous systems, usually in the the forms of vehicles, trams, or forklifts. In vehicles, these systems can range from simple Advanced Driver-Assistance Systems (ADAS), like Lane Keeping Assist Systems, a further development of the modern Lane-Departure Warning System, to Autonomous Driving Systems (ADS), with full self-driving capabilities. In the near future we will see these solutions more and more [17], as the technology matures and gets adopted to more aspects of our lives.

1.3 Testing the correctness of autonomous components

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 [28], 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 [13, 31], but most of these datasets contain concrete scenarios or trajectories, which means limited mutation/perturbation possibilities compared to functional or logical scenarios.

1.4 Graph generation in testing

During the design and testing phases of critical systems, modeling tools are widely used, which help make it possible to automate several steps of development and testing with the usage of graph-based models. First and foremost, a diverse and realistic set of models is necessary for systematic testing, where the models provide the test input. The purpose of the VIATRA Solver framework developed in our department is generating synthetic models, which can serve as test inputs.

In my thesis I describe how I integrated a high performance predicate evaluator to the VIATRA Solver framework. This solution makes VIATRA Solver significantly more scalable and portable. I demonstrate the use of the graph generation using the newly integrated predicate evaluator in a case study, where I generate diverse and realistic scenarios for testing machine learning or deep learning components used in advanced driver-assistance systems.

Using high-level (functional) scenarios is a good way to create concrete test cases. From functional scenarios we can derive 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.

1.5 Pattern matching in the past

Until now, the only way to create patterns was in Viatra Query Language. To use these, we needed to work our way through a long list of steps setting up the integrated development environment (IDE). This included installing Eclipse with EMF and VIATRA, then creating a modeling project, where we could create a metamodel. From this we had to generate model code and editor code, which we had to use to start a Runtime Eclipse. In this instance of Eclipse we could create a Query Project, in it a VQL file, and in this file, we could write our pattern in VQL language. When we saved this file, some Java classes were generated, that we could use in our code.

This old method used EMF objects to store the data, and there was no other option. In the existing codebase in the VIATRA framework, there are two ways to create (partial) models used as the starting point of the generation. One is to create an EMF model either using the graphical user interface (GUI) editor, or programmatically using Java. We can then load this model, and VIATRA builds its own internal datastructure from the data stored in the model.

The other way we can create a partial model is using a tabular method, where we can create a table which we can use to write our data into. Then based on this table, VIATRA creates its own internal datastructure, similar to the previous method. Unfortunately this also only accepts data types provided by EMF.

There are several challenges resulting from these two methods:

- Setting up the IDE is not user-friendly, it has complicated software requirements and necessary settings, that are difficult to find.
- Portability is limited, since one version has Eclipse dependency, while the other version without this dependency is unstable.
- Originally the pattern matching operates on datastructures provided by EMF, which imposes severe performance limitations.

1.6 Structure of thesis

The rest of this thesis has the following structure:

- First, in Chapter 2 I will discuss the most import technological and theoretical background knowledge used in my work. This includes an overview of advanced driver assistance systems,as well as modeling and model generation technologies.
- Next, I will provide a detailed overview of the structure and usage my solution in Chapter 3.
- Then in Chapter 4 I demonstrate my solution in a case study.
- In Chapter 5 I will evaluate the scalability and the performance of my work.

- Finally, I will draw conclusion on my work and describe possible future work in Chapter 6.
- Chapter 6 contains more information about the case study and the technologies used in it.

The case study of my thesis is based on a Student Research Societies Report [21], in which we present a toolchain, which aims to help the testing of system level behaviour of AI-based systems. We introduce an approach to synthesize a test set, that targets critical behaviour of AI-based systems.

Chapter 2

Preliminaries

There are some technologies that are necessary to have a good understanding of in order to fully comprehend the motivation behind the task and the implementation of my solution. In this chapter I explain in detail what ADASs are and how their testing is a strong motivation for my work and also how graph generation works.

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 ADASs 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 on Level 1 and Level 2 of the six levels of driving automation defined in [15], 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 Machine Learning (ML)/Deep Learning(DL) components is challenging, but in my work, I focus on systems with ML/DL components.

2.1.1 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 [16] and autonomous driving standards like [4, 15].

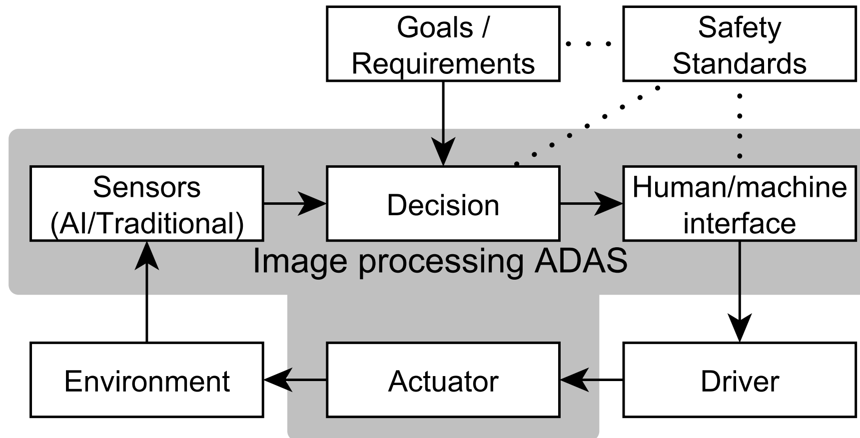


Figure 2.1: Functional overview of a generic ADAS component.

- 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.

2.1.2 Fault hypotheses

With this functional architecture, we can identify 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 [27] with clearly visible geometric objects the state-of-the-art accuracy is still only 99.8% [51][1]. Therefore, a primary goal is the systematic evaluation of object detection components in order to compare the accuracy with an expected reliability.

[1] State-of-the-art accuracy for benchmark [27]: `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 [30]) 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 [46].)

- **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 [41], which requires specialized logic reasoners [6].

2.1.3 Testing of ADAS and self driving systems

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 [13, 31], but these only list a limited number of vaguely defined requirements that the vehicles must meet. Satisfying these requirements is necessary, but not sufficient to guarantee the safe operation of the vehicles. For a comprehensive testing framework we need to provide a way to test the AI components in diverse scenarios, while also ensuring the robustness of the system.

2.2 Graph generation

2.2.1 Domain-specific languages

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) fro example, which is used by VIATRA. 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.

2.2.1.1 Eclipse Modeling Framework

I 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 traffic situation as seen in Figure 2.2.

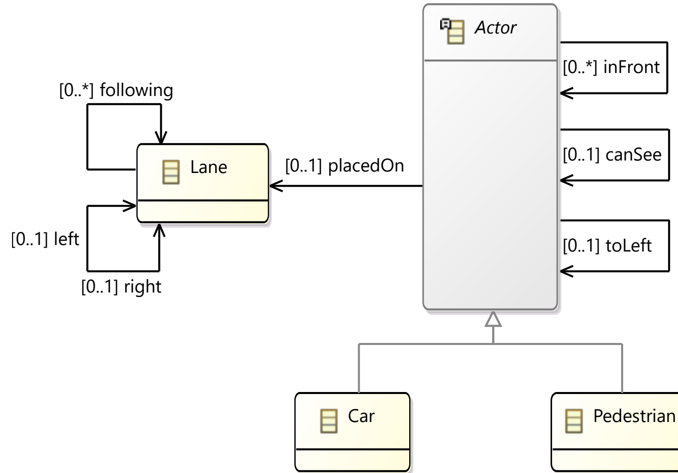


Figure 2.2: Metamodel of a simple traffic situation

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 `Actor` class in my example. This means they cannot be instantiated, but their non-abstract specializations can be. In the example, `Actor` has two specializations, `Car` and `Pedestrian`. This relation is represented in Figure 2.2 with an arrow pointing toward `Actor`.

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 interval. 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 `right` and `left` references of the `Lane` class. EReference may also represent a composition. In the metamodels, usually there is a single class that contains all the other classes. Here I omitted this, to make the model simpler.

2.2.1.2 New specification language for modeling

The new extended version of VIATRA that I worked on uses a different language for describing metamodels and representing partial models instead of EMF. The new specification language is introduced in [37].

The data structures consist of classes, which can have references and attributes. The references represent associations between two classes, and they have a multiplicity, with a lower and an upper bound to specify the minimum and maximum number of relation an instance may have of each of each type.

Example 1. A class representing a Lane section on a road. It may have multiple following Lanes, where a Car can go. These include the trivial continuation of the road, as well as lanes on other roads, where a vehicle is allowed to go from this Lane. E.g. entering an intersection, there are multiple possible directions a car can proceed. A Lane may also have at most one Lane on its left side and at most one Lane on its right side. The left and right references form a bidirectional relation, which is denoted by the opposite keyword.

```
class Lane { Lane[0..*] following
             Lane[0..1] left opposite right
             Lane[0..1] right opposite left }
```

When specifying attributes, we need to state the data type followed by the name of the attribute. Attributes without a specified multiplicity have an implied multiplicity of one.

Example 2. A class for representing four parameters of the weather. We use integers for the attribute values.

```
class Weather {
    int cloud
    int rain
    int fog
    int sun
}
```

Example 3. An extract of an abstract situation metamodel. An Actor may have positional relations relative to other Actors, and it can be placed on a Lane. Car and Pedestrian are subclasses of Actor, which is denoted by the extends keyword.

```
abstract class Actor {
    Actor[0..*] inFront, toLeft, canSee,
    Lane[0..1] placedOn }
class Car, Pedestrian extends Actor.
```

We can create (partial) models by specifying the types of variables and the relations they are a part of.

Example 4. Topological model of two lanes in opposite directions.

```
Lane(l1). Lane(r1). right(l1,r1). left(r1,l1).
Lane(l2). Lane(r2). right(l2,r2). left(r2,l2).
following(l1,l2). following(r2,r1).
```

Example 5. Model with three actors, their positions and visibility relative to each other.

```
Car(ego). Car(green). Car(grey).         // Actors
inFront(ego,green). toLeft(green,grey). // Positions
canSee(ego,green).  canSee(ego,grey).   // Visibility
```

2.2.2 Pattern matching

Based on the metamodel seen previously (Figure 2.2), we can create models of road networks, with cars and pedestrians placed on them. Given such models, we might want to find ones that fulfil some criteria. We need this type of search when we would like to filter and discard models, which do not meet our expectations in some aspects. This means we need to be able to find specific patterns inside these models. We can think of the models as graphs, where the elements of the models are the nodes of the graph and the relations between them are different types of edges in the graph. We want to find specific relations or patterns in these graphs. Give a pattern, we can assign graphs to it, which contains this pattern. This process is called graph pattern matching, and it is frequently used in graph generator algorithms, such as the one used in the VIATRA Solver framework[42].

For this we need a language, that can be used to describe these patterns, and which the graph pattern matcher can interpret. There are several of these, such as Object Constraint Language (OCL) [39], or VIATRA Query Language (VQL) [12]. While creating the patterns using these languages, we can use the types, references, etc. defined in the metamodel. The old version of VIATRA Solver uses VQL, so I will use that in some of my examples.

2.2.2.1 VIATRA Query Language

Using VQL we can define a pattern in the following way. We start the pattern with the **pattern** keyword, which is followed by the name of the pattern, an then the parameters in parenthesis. Finally, in braces we need to list the constraints, which need to be fulfilled for the pattern to match the graph. It is recommended to define the types of the parameters, but this can be done later in the body as constraint.

Example 6. The basic syntax of a pattern in VQL.

```
pattern myPattern(a,b : MyClass1, c : MyClass2) {
    ...
    constraints;
    ...
}
```

The simplest type of constraint is the *type constraint*. With this we can determine the type of a variable. This constraint is satisfied, if the type of the given variable matches the one determined by the constraint, or one of its descendants.

Example 7. Syntax of a type constraint, which matches, if entityVariable is of type Car.

```
Car(entityVariable);
```

Example 8.

The type constraint with type Lane matches even if entityVariable is of types Car or Pedestrian.

```
Actor(entityVariable);
```

We can also add relations (reference, composition, attribute) as constraints. This matches, if the given relation exists in the model.

Example 9. To find the lanes that are next to each other, we can search for pairs of Lanes, which have right reference between them.

```
pattern rightLane(left : Lane, right : Lane) {
    Lane.right(left, right);
}
```

Example 10. If we generate models only based on the metamodel, we might get some models, where the left or the right reference of a Lane is pointing to itself. Of course this cannot happen in real life, so we would like to find the models, which have this error. This can be done with the following pattern.

```
pattern laneNextToItself(lane : Lane) {
    Lane.right(lane, lane);
}
```

We can also search for transitive closure in the graphs. The transitive closure of a $G = (V, E)$ directed graph is the $G' = (V, E')$ graph, in which there is an edge from u to v if and only if there is a directed path from u to v in the original G graph. In our case, since there are different types of edges in the graph, we need to provide the type of edge that the directed path needs to consist of from u to v. In VQL these edges must be provided in the forms of patterns. We can use the `find` keyword to use previously defined patterns as part a new pattern. To search for transitive closure, we need to use the + symbol.

Example 11. We can find all the lanes to the right of a lane, if we search for all the lanes which are reachable through right edges. For this we can use the right pattern defined previously.

```
pattern rightLanes(left : Lane, right : Lane) {
    find right+(left, right);
}
```

For negating a constraint, we can use the `neg find` keywords. With this we can negatively match patterns, meaning it will not give a match if the original pattern matches, and it will give a match if the original pattern is not found in the graph.

Example 12. We can search for lanes, which are not directly next to each other.

```
pattern notRightLane(lane1 : Lane, lane2 : Lane) {
    neg find rightLane(lane1, lane2);
}
```

Checking equality and the inequality of two variables can be done using the `==` and `!=` operators.

Example 13. The previous example does not guarantee, that the two lane, lane1 and lane2 are different lanes. We need to give a separate constraint for this.

```
pattern notRightLane(lane1 : Lane, lane2 : Lane) {
    lane1 != lane2;
    neg find rightLane(lane1, lane2);
}
```

Example 14. In case we would like find the lanes, that have at least one lane to its right, but we don't need to know which lanes are to the right, we just need to leave lane2 out of the parameter list. If the variable is only used once, prefix or replace the name of a variable using the _ symbol.

```
pattern hasRightLane(lane1 : Lane) {
    find rightLane(lane1, _);
}
```

A pattern can have multiple bodies separated by the `or` keyword. This means the pattern will match, if at least one of the bodies matches.

Example 15. A pattern with multiple bodies.

```
pattern hasLeftOrRightLane(lane: Lane) {
    Lane.left(lane, leftLane);
} or {
    Lane.right(lane, rightLane);
}
```

Although VQL is capable of more than described here, I only showed the parts needed to understand the examples in the following chapters.

2.2.2.2 Syntax of predicates in the new specification language

Using the new specification language we can define a predicate in the following way. We start the pattern with the `pred` keyword, which is followed by the name of the pattern, an then the parameters in parenthesis. Finally, in braces we need to list the constraints, which need to be fulfilled for the pattern to match the graph. It is recommended to define the types of the parameters, but this can be done later in the body as constraint.

Example 16. The basic syntax of a predicate in the new description language.

```
pred myPattern(MyClass1 a, MyClass1 b, MyClass2 c) <->
    ...
    constraints,
    ... .
```

Using this new description language we are able to define the same types of constraints as with VQL. These are demonstrated below.

Example 17. Syntax of a type constraint, which matches, if entityVariable is of type Car.

```
Car(entityVariable).
```

Example 18. The type constraint with type Lane matches even if entityVariable is of types Car or Pedestrian.

```
Actor(entityVariable).
```

Example 19. To find the lanes that are next to each other, we can search for pairs of Lanes, which have rightLane reference between them.

```
pred rightLane(Lane left, Lane right) <->
    right(left, right).
```

Example 20. Transitive closure in the new description language.

```
pred rightLanes(Lane left, Lane right) <->
    right+(left, right).
```

Negating a constraint, is done using the exclamation mark (`!`). With this we can negatively match patterns, meaning it will not give a match if the original pattern matches, and it will give a match if the original pattern is not found in the graph.

Example 21. Negation in the new description language.

```
pred notRightLane(Lane left, Lane right) <->
    !right(lane1, lane2).
```

Checking equality two variables can be done using the `equals` operator, and inequality can be checked by negating the equality.

Example 22. Inequality in the new description language.

```
pred notRightLane(Lane left, Lane right) <->
    !equals(lane1, lane2),
    neg find rightLane(lane1, lane2).
```

Example 23. Similarly to VQL, we can prefix or replace the name of a variable with the _ symbol if is only used once.

```
pred hasRightLane(Lane lane1) <->
    right(lane1, _).
```

A pattern can have multiple bodies separated by the semicolon (`;`) character. This means the pattern will match, if at least one of the bodies matches.

Example 24. A pattern with multiple bodies.

```
pred hasLeftOrRightLane(Lane lane) <->
    left(lane, leftLane)
  ; right(lane, rightLane).
```

If we would like to specify what truth values we want the predicated to evaluate to, we can use direct pred instead of pred at the beginning. Then, during the pattern matching it will only match a model, if the truth value of the predicate matches the one we specified. For a summary of the four valued logic see Section 3.1.

Example 25. A direct predicate, which matches a Lane, if it must have a following Lane, or if it may have one.

```
direct pred hasFollowing(from) <->
    following(from, _to) = true|unknown.
```

Chapter 3

4-valued graph predicate evaluation

3.1 4-valued graphs

The graph generator algorithm uses Belnap-Dunn 4-valued logic [10, 29], which allows us to represent unfinished, partial models, as well as errors and inconsistencies arising during the evaluation of computations over such models. The semantic foundations for the specification language are described in [37]. This section provides a brief introduction of Belnap-Dunn 4-valued logic based on [37].

Belnap-Dunn 4-valued logic contains the usual false `false` and true `true` truth values, the unknown `unknown` value introduced for unspecified or unknown properties, and the inconsistent `error` value that signals inconsistencies and computation failures. The subset {`false`, `true`, `unknown`} of logic values can express partial, but consistent information. Conversely, the subset {`false`, `true`, `error`} expresses possibly inconsistent, but complete information.

Two partial orders can be defined over 4-valued logic values (Figure 3.1). Information order (denoted by $\sqsubseteq$) expresses the gathering of information as new facts are learned during the refinement of partial models. Facts with `unknown` logical value can be set to either `true` or `false`, while a change to `error` signifies an inconsistency or failure. This order is defined as

$$(X \sqsubseteq Y) \Leftrightarrow [(X = \texttt{unknown}) \vee (X = Y) \vee (Y = \texttt{error})]. \tag{3.1}$$

The second partial order is implication order, which defined as

$$(X \leq Y) \Leftrightarrow [(X = \texttt{false}) \vee (X = Y) \vee (Y = \texttt{true})] \tag{3.2}$$

and serves as a generalization of logical implication. We will write X $\sqsubset$ Y and (resp. X $<$ Y) when X $\sqsubseteq$ Y (resp. X $\leq$ Y) and X $\neq$ Y hold.

The information merge operator $\bigoplus$ merges 4-valued truth values where contradictory information results in `error`. Other operations on 4-valued truth values $\neg^4$, $\vee^4$, and $\wedge^4$ are extensions of the usual logic operators $\neg$, $\vee$, and $\wedge$. Their truth tables (see FFigure 3.2) correspond with their classical counterparts for {`false`, `true`} inputs.

Semantically, `unknown` truth value represents potential `true` or `false` (or `error`) values, and the semantic is chosen to cover all of those options. On the other hand, `error` is often unintuitive, but it allows the precise and explicit localization of inconsistencies within models [10, 14]. For example, we may see that if X =`error` and Y =`unknown`, then X $\vee^4$ 4 Y = `true`, because the only way for our logical inference to result in a consistent truth value is to eventually learn that Y is `true`. Should Y turn out to be `false`, the inconsistent `error` value will be propagated.

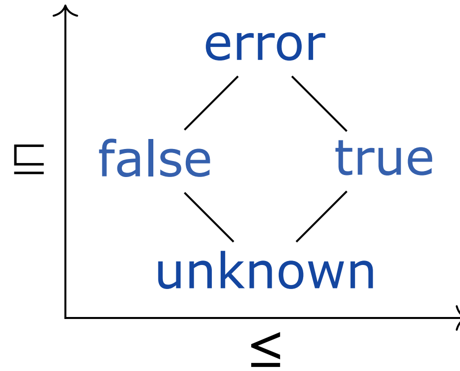


Figure 3.1: Logic values

$\bigoplus$	u	f	t	e
u	u	f	t	e
f	f	f	e	e
t	t	e	t	e
e	e	e	e	e

$\neg^4$	
u	u
f	t
t	f
e	e

$\vee^4$	u	f	t	e
u	u	u	t	t
f	u	f	t	e
t	t	t	t	t
e	t	e	t	e

$\wedge^4$	u	f	t	e
u	u	f	u	f
f	f	f	f	f
t	u	f	t	e
e	f	f	e	e

Figure 3.2: Truth tables of information merge and logical connectives

3.2 Creating graph patterns

To illustrate how different aspects of my work function, I will use a simplified version of a road network and vehicle placements, shown in Figure 2.2.

The first part of my solution allows us to use our own datastructures instead of EMF models or tables filled with EMF types, which gives us more flexibility.

With the improvements, creating a pattern has become a much simpler process. We can define them in Java code, using disjunctive normal form (DNF), as DNFPredicate objects. The classes used for this are shown in Figure 3.3. We can also use the new syntax described in Section 2.2.2.2. Predicates defined using this method are then parsed and converted to DNFPredicate objects.

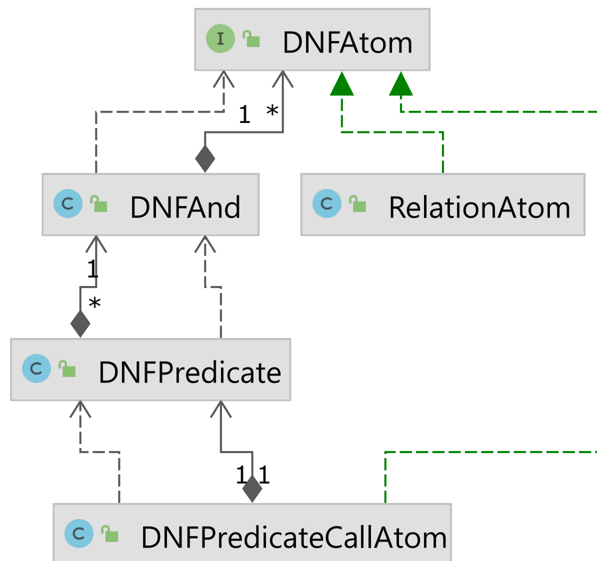


Figure 3.3: Class diagram of DNF classes

My solution allow us to transform the DNFPredicate objects created in the previous step to PQuery objects, which is the representation that VIATRA Solver can directly work with. This is done using the following mapping:

- DNFPredicate→SimplePQuery (my own implementation of PQuery)
- DNFAnd→PBody
- DNFAtom→PConstraint
  - RelationAtom→TypeConstraint
  - DNFPredicateCallAtom→PositivePatternCall, NegativePatternCall or BinaryTransitiveClosure, depending on attributes

3.2.1 Defining predicates using disjunctive normal form

Graph patterns or predicate definitions can be written by arranging predicates in disjunctive normal form (DNF). Both the syntax of the specification language and the Java implementation follow this canonical normal form to represent predicates. A logical formula is always evaluated on a (partial) model. In the sections below I will demonstrate how these can be created and the way they relate to the VQL patterns, as well as the PQuery objects used by VIATRA Solver. I will also show how these are evaluated using Belnap-Dunn 4-valued logic.

During predicate evaluation an `error` value occurs, when a value must be `true` and `false` at the same time. This can happen if different constraint require different values. Since during generation we discard models where an `error` is found, I will not describe evaluations involving `error` values in detail.

3.2.1.1 Building blocks of predicates

A DNFAtom is an expression that can be evaluated to a truth value using Belnap-Dunn 4-valued logic described in Section 3.1. This is interface for the specific types of atoms one may use. They are transformed to implementations of the PConstraint interface, as expected by VIATRA Solver.

3.2.1.2 Predicates referencing relations

A RelationAtom evaluates to `true`, if the specified type of relation exists between the two variables. When specifying this atom, we have to first state the type of relation the relation, then the source and target of the relation. This atom is transformed into a TypeConstraint object, implementing the PConstraint interface.

> **Example 26.** Relation expressed in the new specification language.
>
> ```
> following(lane1, lane2).
> ```

> **Example 27.** Relation expressed in VQL.
>
> ```
> Lane.following(lane1, lane2);
> ```

The type of a variable is a special type of relation. Whereas for a binary relation a TypeConstraint means that the given type of relation exists between the two variables, here it will evaluate to `true`, if the variable is an instance of the specified class, or one of its subclasses.

> **Example 28.** Type expressed in the new specification language.
>
> ```
> Lane(lane1).
> ```

> **Example 29.** Relation expressed in VQL.
>
> ```
> Lane(lane1);
> ```

Another special relation is the interpreted equivalence, which allows us to represent abstractions on the number of nodes explicitly.

> **Example 30.** A positive and a negative equivalence expressed in the new specification language.
>
> ```
> equals(lane1, lane2),
> !equals(lane1, lane3).
> ```

> **Example 31.** A positive and a negative equivalence expressed in VQL.
>
> ```
> lane1 == lane2;
> lane1 != lane3;
> ```

> **Example 32.** To see when a relation is evaluated to `unknown`, look at the following example, illustrated in Figure 3.4. This includes five lanes named l1-l5, as well as three cars named c1-c3. The lanes follow each other in a straight line, and the three

cars are placed on the first three lanes. The equality is represented by the character. The arrows with solid lines represent the **true** value, while the arrows with dashed lines are **unknown**. There is a Car::new node, which represents all the cars that could exist in the model. This has an **unknown** existence, since we don't know if more cars will be created, and the possibly existing cars might be placed on the empty lanes, so these relations are also **unknown**.

The twoFullLanes predicate below, will evaluate to **true** with the l1-l2, and the l2-l3 lane pairs, and it will evaluate to **unknown** with the l3-l4 and the l4-l5 ane pairs.

```
pred twoFullLanes(Lane lane1, Lane lane2) <->
    placedOn(_, lane1),
    placedOn(_, lane2),
    following(lane1, lane2).
```

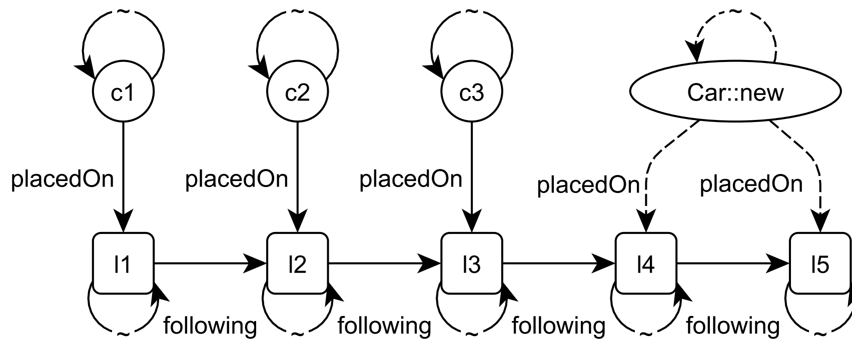


Figure 3.4: Partial model with unknown relation values

3.2.1.3 Conjunction of expressions

DNFAnd is the conjunction of expressions. This implements the $\wedge^4$ (and) operation. It evaluates to **true**, if all included expressions evaluate to **true**, evaluates to **false**, if at least one of the expressions evaluate to **false**. The cases involving **unknown** or **error** values can be seen in Figure 3.2. When transforming it for use with VIATRA Solver, it is turned into a PBody object.

Example 33. Conjunction expressed in the new specification language. A list of expressions, separated by commas.

```
Car(car),
Lane(currLane),
placedOn(car, currLane).
```

Example 34. Conjunction expressed in VQL. A list of expressions, separated by semicolons and surrounded by braces. In VQL this is called the body of the pattern.

```
{
    Car(car);
    Lane(currLane);
```

```
    Car.placedOn(car, currLane);
}
```

3.2.1.4 Disjunction of conjunctions

A representation of a predicate definition, consisting of a disjunction of conjunctions. This implements the $\vee^4$ (or) operation. It evaluates to **true**, if at least one of the included conjunctions evaluates to **true**. The other cases can be seen in Figure 3.2. When transformed into a form required by VIATRA Solver, it is turned into a SimplePQuery object, an implementation of the PQuery interface.

> **Example 35.** Predicate expressed in the new specification language. It starts with the keyword **pred**, followed by the name of the predicate. Then the parameters are listen in parenthesis, together with their type. The header and the body are separated by the <-> characters. This is followed by one or more bodies, a list of conjunctions, separated by semicolons. The predicate is ended with a period. The semicolon is interpreted as the or operation between the conjunctions.
>
> ```
> pred hasLeftOrRightLane(Lane lane) <->
>     left(lane, leftLane)
>   ; right(lane, leftLane).
> ```

> **Example 36.** Predicate expressed in VQL. It starts with the keyword **pattern**, followed by the name of the predicate. Then the parameters are listen in parenthesis, together with their type. This is followed by one or bodies, a list of conjunction surrounded by braces. These bodies are separated by the keyword **or**.
>
> ```
> pattern hasLeftOrRightLane(lane: Lane) {
>     Lane.left(lane, leftLane);
> } or {
>     Lane.right(lane, rightLane);
> }
> ```

3.2.1.5 Referencing predicates inside predicates

This atom represents a special relation between variables, where the predicate given as parameter is used instead of a simple relation specified in the metamodel. It evaluates to the same value, as the referenced predicate.

> **Example 37.** DNFPredicateCallAtom expressed in the new specification language. Referencing a previously defined predicate hasLeftOrRightLane.
>
> ```
> pred hasEmptyLeftOrRightLane(Lane lane) <->
>     hasLeftOrRightLane(lane, otherLane),
>     !placedOn(_actor, otherLane).
> ```

> **Example 38.** DNFPredicateCallAtom equivalent expressed in VQL. Referencing a previously defined predicate hasLeftOrRightLane.
>
> ```
> pattern hasEmptyLeftOrRightLane(lane: Lane) {
> ```

```
    find hasLeftOrRightLane(lane, otherLane);
    neg find placedOn(_actor, otherLane);
}
```

The DNFPredicateCallAtom has two Boolean attributes, positive and transitive, The positive attribute marks whether it needs to be negated or not, and if the transitive attribute is true, we are looking for the transitive closure instead of the simple relation. A transitive closure currently cannot be negated.

Example 39. A transitive closure expressed in the new specification language, using the following relation.

```
pred reachableFollowing(Lane lane1, Lane lane2) <->
    following+(lane1, lane2).
```

Example 40. A transitive closure expressed in VQL, using the following relation. In VQL transitive closure can only be applied to patterns we define, not to simple relation. This means, we need to wrap the following relation with a pattern.

```
pattern following(lane1: Lane, lane2: Lane) {
    Lane.following(lane1, lane2);
}

pattern reachableFollowing(lane1: Lane, lane2: Lane) {
    find following+(lane1, lane2);
}
```

3.3 Pattern matching on custom datastructures

If we want to perform pattern matching on our own datastructures instead of the datatypes and data structures provided by EMF, we have two options. The easier method is to create a tabular context, similar to the one mentioned in Section 1.5. Our own implementation could accept any datatypes instead of only the EMF types. VIATRA already provides the necessary interfaces we would need to implement. Unfortunately, this has the downside of containing redundant steps, which cannot be circumvented. The data in the partial models are created twice. Once when we write them into the tables provided by the interface, and once when they are copied into an internal datastructure used by VIATRA Solver. This redundant step has a negative impact on the performance.

The only way to circumvent this redundancy, is to create a datastructure, which implements the interfaces expected by the solver, but can also be accessed by the user. This way the data will be written directly into this datastructure, and the solver can also use it, without having to copy and transform it. This also means, that some of the classes provided for the existing solutions need to be rewritten or adapted for this new solution. Luckily, we can look at the existing solutions to see what interfaces we need to implement.

Although there were many classes created, I will only describe the most important ones below. These are shown in Figure 3.5, along with some others, which help understand the connection between the described classes.

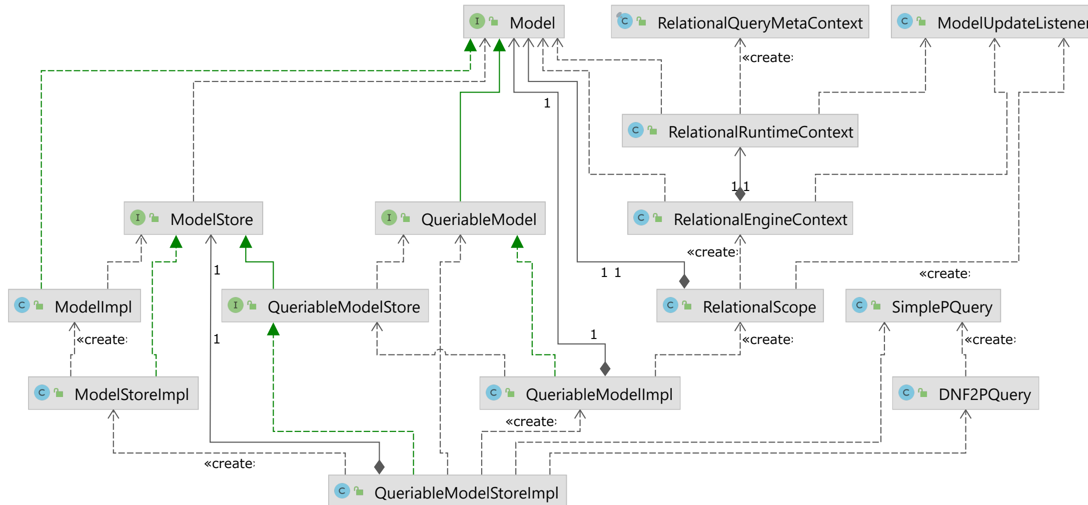


Figure 3.5: Class diagram of some of the classes created for pattern matching on custom datastructures.

```
@Override
public Iterable<Tuple> enumerateTuples(IInputKey key, TupleMask seedMask, ITuple
    seed) {
    RelationView<?> relationalViewKey = checkKey(key);
    Iterable<Object[]> allObjects = relationalViewKey.getAll(model);
    Iterable<Object[]> filteredBySeed = filter(allObjects,objectArray ->
        isMatching(objectArray,seedMask,seed));
    return map(filteredBySeed,Tuples::flatTupleOf);
}
```

Listing 3.1: Method used for initializing the Rete network.

3.3.1 Initiating the Rete network

One of the first steps in incremental pattern matching [11] is initiating the Rete network [23]. This is done in the enumerateTuples() method of the RelationalRuntimeContext class, shown in Listing 3.1, which explicitly lists all of the content of the relations in order to index them. This is only executed once, at the beginning.

3.3.2 Applying changes to the model

The goal of the second component is to delegate changes. This component collects the changes to the model and stores them.

The processing of changes applied to the models is shown in Figure 3.6. This process is conducted by the ModelUpdateListener class. After the model is created and the Rete network is initialized, they are synchronized. When we change the model using the put() method, the change is stored in a buffer, not yet applied to the Rete network. Every new change is placed in the buffer until the flush() method is called. This replays the changes one by one, and delegates them to the Rete network. Once all changes have been replayed, the model and the Rete network are once again synchronized.

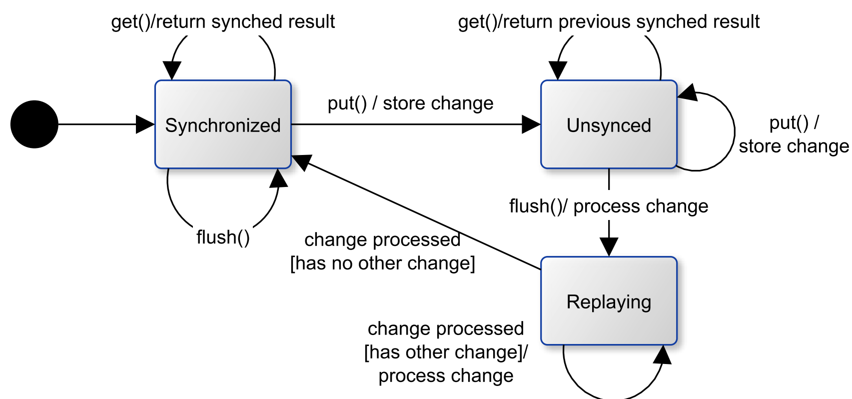


Figure 3.6: Processing changes applied to the model

Chapter 4

Testing of autonomous vehicles using graph predicates

For demonstrating a use case for graph generation, I will use my former work for this year's Students' Scientific Conference [21], which I wrote with Balázs Pintér. In this we presented a toolchain, which is able to synthesize a test set, that targets critical behaviour of AI-based systems. Our approach relies on advanced simulator and graph generation technologies. From the test 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. I will show the steps which already use VIATRA, that can easily be replaced with the description language and predicate evaluator. I will also show some pints in the toolchain where we could introduce new uses for pattern matching.

4.1 Scenes and Situations

During our test generation, at the scenarios we followed the three abstraction levels described in [38]: 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, [45] introduced the standardized terms. According to [45], 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: [36] 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.

4.2 Functional overview

Our testing framework consists of five main steps as illustrated in Figure 4.1. 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.

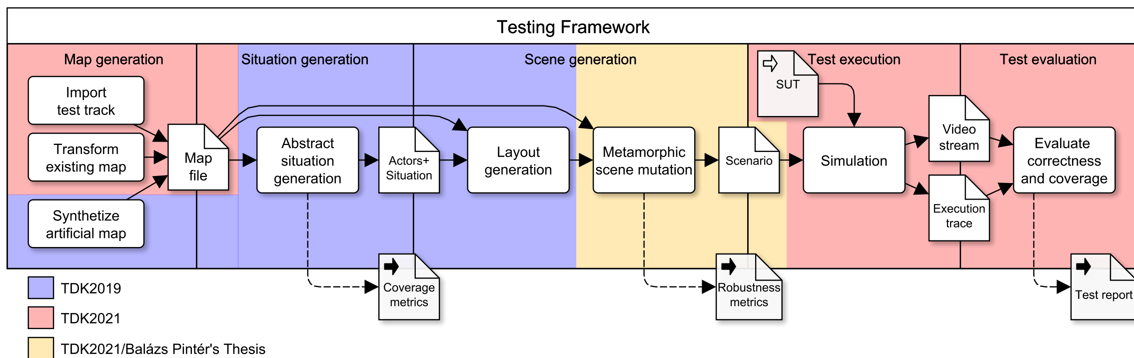


Figure 4.1: Functional overview

The presented testing framework is built upon and greatly improves our previous work from 2019 [22]. In Figure 4.1 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.

4.2.1 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

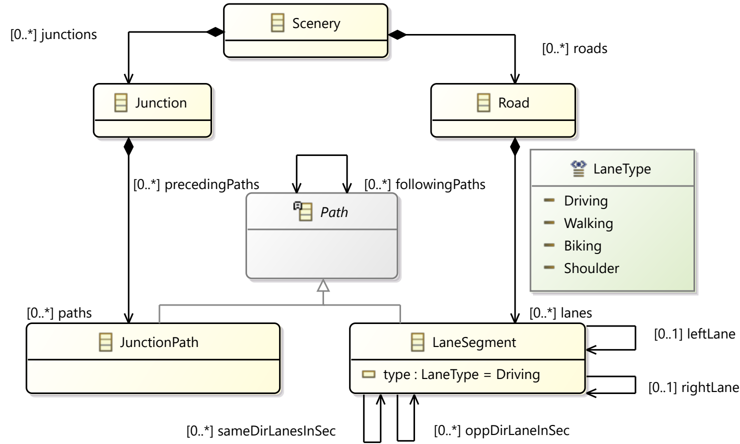


Figure 4.2: Metamodel of the road network

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.

Figure 4.2 illustrates 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 [49]. Road structure (OpenDRIVE) and infrastructure (Base 3D model of the location) belong here, these will not change, even during the perturbation generation.

An other method available to us is to generate synthetic maps. Such generation method is described in [22]. This used a different metamodel, shown in Figure 4.3. This uses constraints during the generation, which originally were constructed using patterns written in VQL. Some of these can be seen in Example 41 and Example 42. These each contain a diagram of a graph pattern, which indicates that a model represents a road network that cannot exist in real life. They also include the original VQL implementations, as well as the new version of the patterns using the new syntax.

[1]https://avlzalazone.com/

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

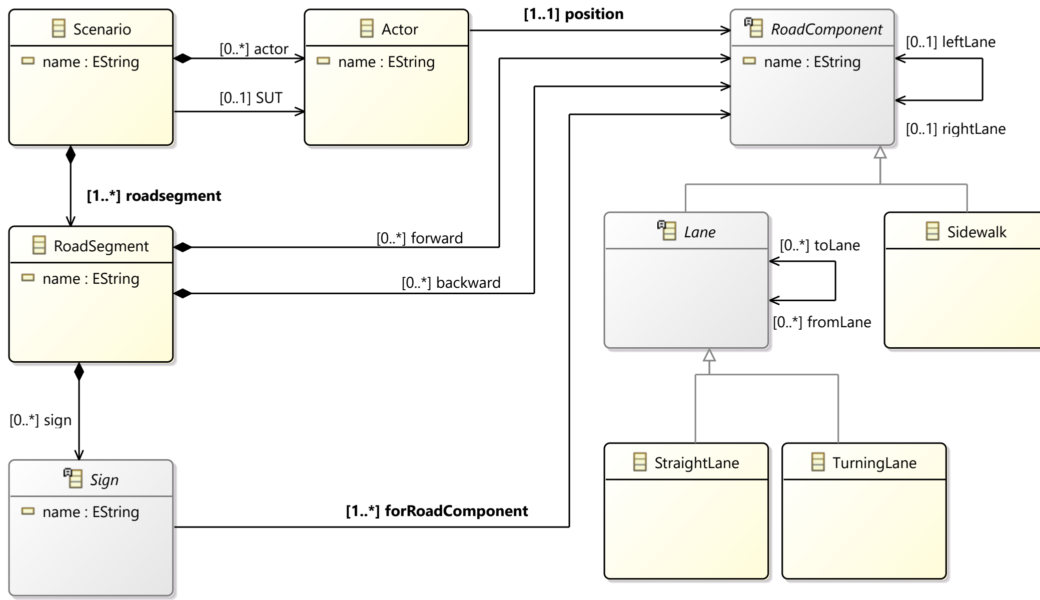
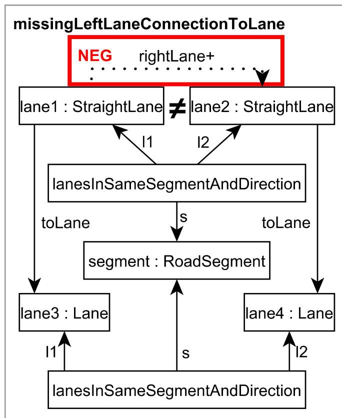


Figure 4.3: Metamodel of the road network used in [22]

Counter example	Graph pattern

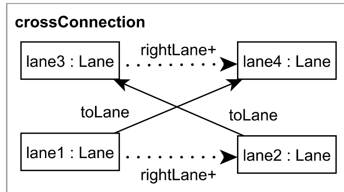
Example 41. Missing rightLane–leftLane relation between two parallel straight lanes going in the same direction in the same RoadSegment.



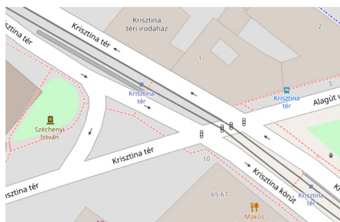

```
pattern missingLeftLaneConnectionToLane(
        lane1 : StraightLane,
        lane2 : StraightLane) {
    lane1 != lane2;
    find lanesInSameSegmentAndDirection(
        segment1, lane1, lane2);
    Lane.toLane(lane1, lane3);
    Lane.toLane(lane2, lane4);
    find lanesInSameSegmentAndDirection(
        segment1, lane3, lane4);
    neg find leftLaneOrRightLaneTransitive(
        lane1,lane2);
}
pred missingLeftLaneConnectionToLane(
        StraightLane lane1,
        StraightLane lane2) <->
    !equals(lane1, lane2),
    lanesInSameSegmentAndDirection(
        segment1, lane1, lane2),
    toLane(lane1, lane3),
    toLane(lane2, lane4),
    lanesInSameSegmentAndDirection(
        segment1, lane3, lane4),
    !leftLaneOrRightLaneTransitive(
        lane1, lane2).
```

Counter example	Graph pattern

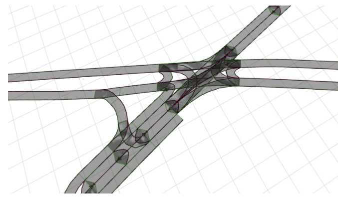
Example 42. Two parallel Lanes crossing each other.



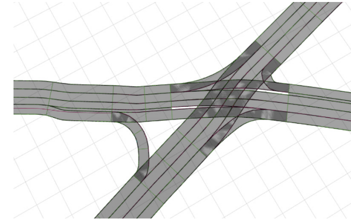

```
pattern crossConnection(lane1: Lane,
        lane2: Lane, lane3: Lane, lane4: Lane) {
    find rightLane+(lane1, lane2);
    find rightLane+(lane3, lane4);
    Lane.toLane(lane1, lane4);
    Lane.toLane(lane2, lane3);
}
pred crossConnection(Lane lane1, Lane lane2,
        Lane lane3, Lane lane4) <->
    rightLane+(lane1, lane2),
    rightLane+(lane3, lane4),
    toLane(lane1, lane4),
    toLane(lane2, lane3).
```



(a) Original location

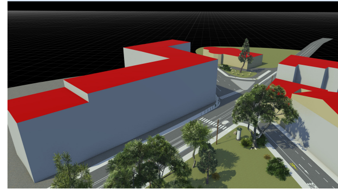


(b) OpenDRIVE from OpenStreetMap

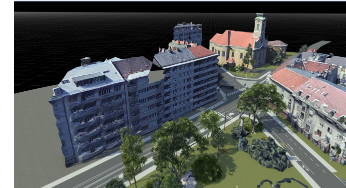


(c) Manually modified OpenDRIVE

(d) Google Maps 3D



(e) Buildings from OpenStreetMap



(f) Buildings from Google Maps

Figure 4.4: Map generation methods

Example 43. Figure 4.4 provides insight into the environment building process.

- Figure 4.4a: screenshot of the location, from `openstreetmap.org`
- Figure 4.4b: OpenDRIVE file generated directly from the exported openstreetmap file. It contains incorrect paths and lanes, tram tracks are missing.
- Figure 4.4c: Manually modified OpenDRIVE file, based on the real lanes and paths in this area.
- Figure 4.4d: Screenshot of the birds eye view from Google Maps.
- Figure 4.4e: 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 4.4f: Virtually constructed environment based on the 3D building models of Google Maps

4.2.2 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 [24][3]. At this point no concrete coordinates or maneuver paths are known, only relative positions as Figure 4.5 shows.

The situation generation also uses VIATRA, with graph patterns as constraints. Some of these can be seen in Example 44 and Example 45, along with their equivalent using the new syntax.

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 [36], but this is beyond our work.

> **Example 44.** There are multple AbstractDistanceRelation objects between the same pair of actors.

```
pattern ax3(d1 : DynamicComponent, d2 : DynamicComponent){
    AbstractDistanceRelation(ar1);
    AbstractDistanceRelation(ar2);
    find h3(d1, ar1, d2);
    find h3(d1, ar2, d2);
    ar1 != ar2;
}

pred ax3(DynamicComponent d1, DynamicComponent d2) <->
    AbstractDistanceRelation(ar1),
    AbstractDistanceRelation(ar2),
    h3(d1, ar1, d2),
    h3(d1, ar2, d2),
    !equals(ar1,ar2).
```

> **Example 45.** Ego cannot be connected to another object via an AbstractPositionRelation because of default visibility.

```
pattern c4(e : EgoActor, ar : AbstractPositionRelation){
    AbstractRelation.src(ar, e);
} or {
    AbstractRelation.tgt(ar, e);
}

pred c4(EgoActor e, AbstractPositionRelation ar) <->
    src(ar, e)
  ; tgt(ar, e).
```

> **Example 46.** Figure 4.6a and Figure 4.6b show two different scenes in Carla. However, both of them can be described by the same abstract situation depicted in Figure 4.6c, 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] Current implementation available at https://github.com/ArenBabikian/CarlaScenarioGen

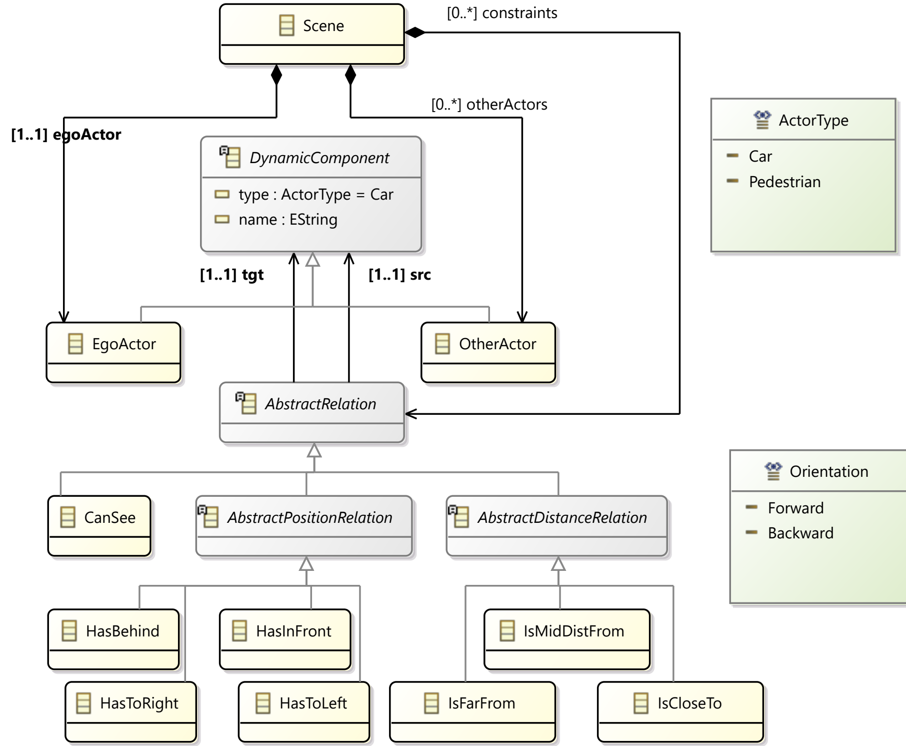


Figure 4.5: Metamodel of the abstract situation

4.2.3 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, [22, 6]).

Static objects and their perturbation is also generated here, based on the map file. Figure 4.7 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 47.** Figure 4.8a shows the layout of the dynamic objects, Figure 4.8b shows an example layout including static objects.

> **Example 48.** Figure 4.9 shows the generated static objects, in the background. Pictures in the same row were taken from the same location, with different object layout.

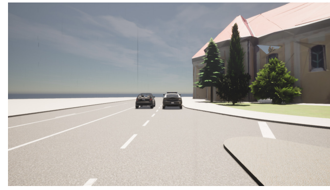
> **Example 49.** We constructed multiple weather blueprint, with different fog, rain, cloud and sun parameters, Figure 4.10 shows the same scenario with different weather.

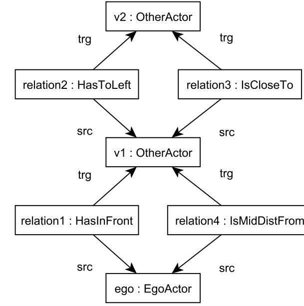
Combining the concrete static and dynamic objects on the map gives a scenario. The result of the layout generation looks like 4.8a. (excluding static objects, due to the visibility.)

The 4th level of the 4 level model [49] is the environment conditions. Testing at different environment conditions is really vital, as proven in [44] small changes of the light or weather conditions can result in a bad (and dangerous) path prediction. At the perturbation we

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

(a) Situation 1 in Carla


(b) Situation 2 in Carla



(c) Abstract model

Figure 4.6: A model of an abstract situations and two examples of it in the simulator

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.

Figure 4.2.3 shows how we can represent four parameters of the weather in our model. It also has a possible constraint, which prevents the generation of weather conditions, where it is raining while there are no clouds in the sky.

> **Example 50.** The weather parameters are represented as integers. We consider it an error if it's raining while the cloud covers less than 30% of the sky.
>
> ```
> class Weather {
>     int cloud
>     int rain
>     int fog
>     int sun
> }
>
> pred sunshineAndRain(Weather w) <->
>     cloud(w,c),
>     rain(w,r),
>     c  < 30,
>     r >= 10.
> ```

4.2.4 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.

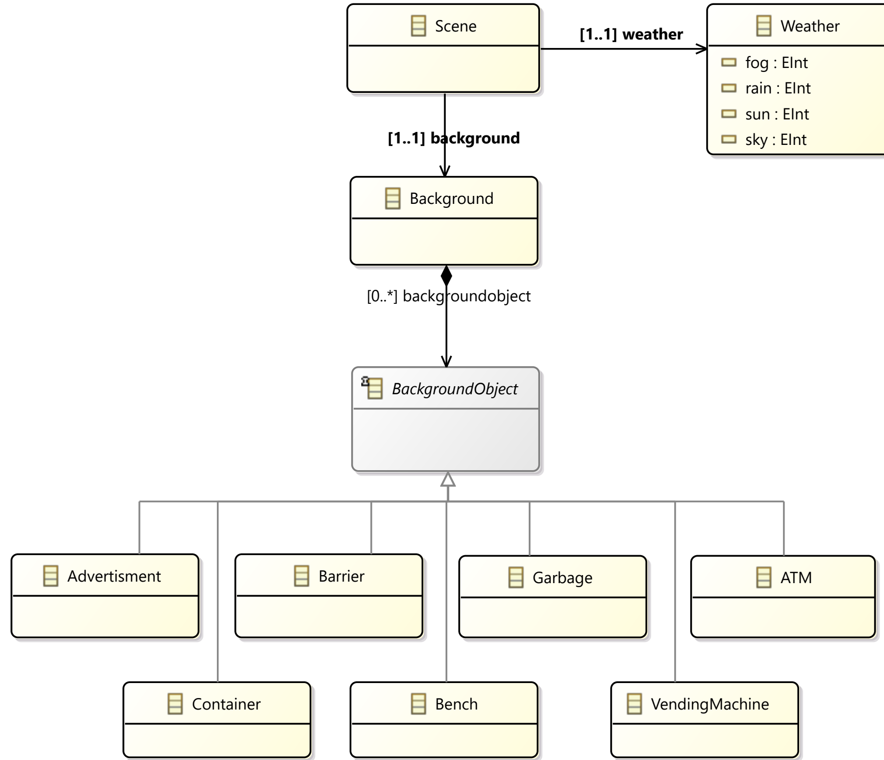


Figure 4.7: Metamodel of the perturbations

4.2.5 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 [51].

4.3 Testing approaches

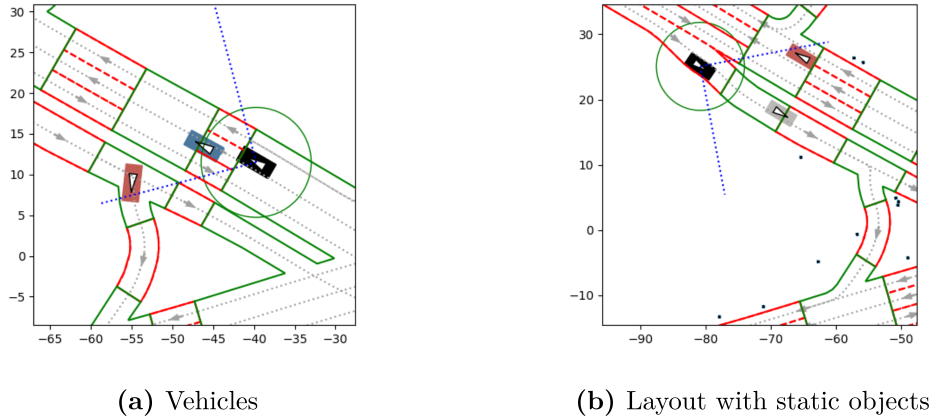
Proving the correct behavior of an ML/DL-based component or system is a challenging task: Verification is not possible on large scale, and testing requires a lot of resources (either data, computing capacity and/or time).

With our testing workflow multiple testing approach is realizable:

- Metamorphic testing
- Coverage based

4.3.1 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,



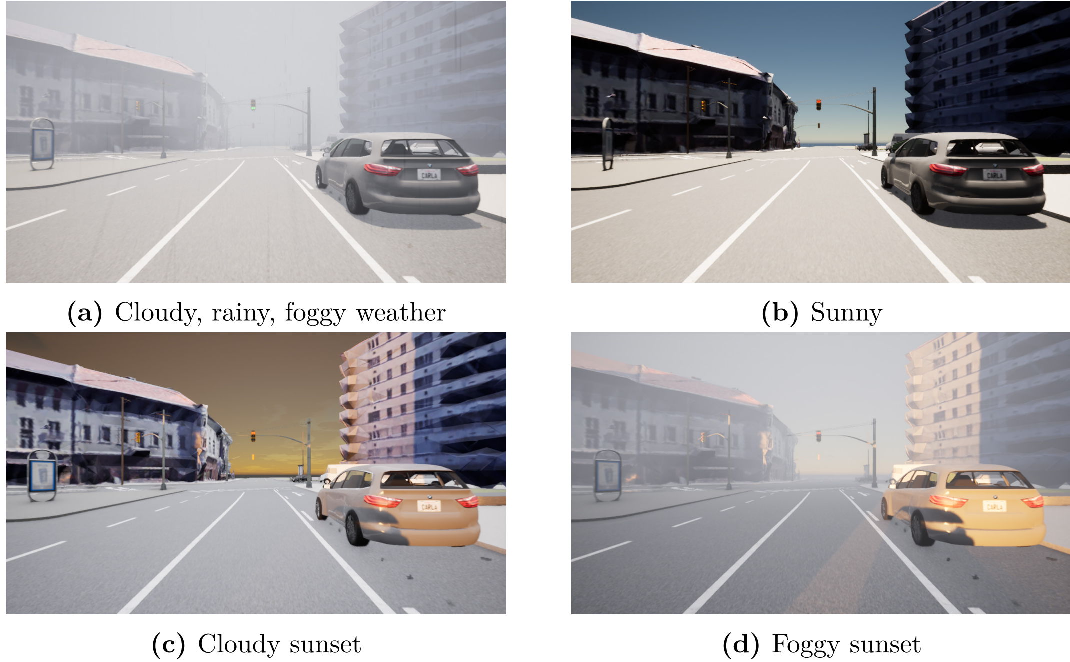
(a) Vehicles **(b)** Layout with static objects

Figure 4.8: Layout of an abstract situation

Figure 4.9: Static object comparison

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 [19, 47] 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)



(a) Cloudy, rainy, foggy weather

(b) Sunny

(c) Cloudy sunset

(d) Foggy sunset

Figure 4.10: Weather conditions in Carla

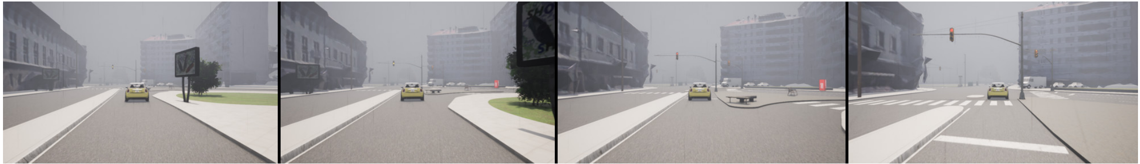


Figure 4.11: Execution

Different filters on an image can also make computer vision components fail, as presented in [44]. In [44] (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.

4.3.2 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.

4.4 Summary of scenario building

During the testing workflow in 4.1, we followed the 4 level scenario hierarchy, mentioned in [49]. This allowed us, to handle the different levels of a scenario in each abstraction layer differently, making the process more transparent and portable.

- *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*

  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 [8, 2, 7] 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 [8] or in [2]

Chapter 5

Evaluation

5.1 Research questions

I evaluated my solution by formulating various research questions and answering them by measuring execution times. These are the research questions I aim to answer:

RQ1 How does the pattern matching scale if we increase the model size?

RQ2 How does the pattern matching scale if we increase the number of changes?

5.2 Selected domain

For the measurement I used a simple simple metamodel shown in Figure 5.1. The implementation of the metamodel using the new specification language is shown in Listing 5.1.

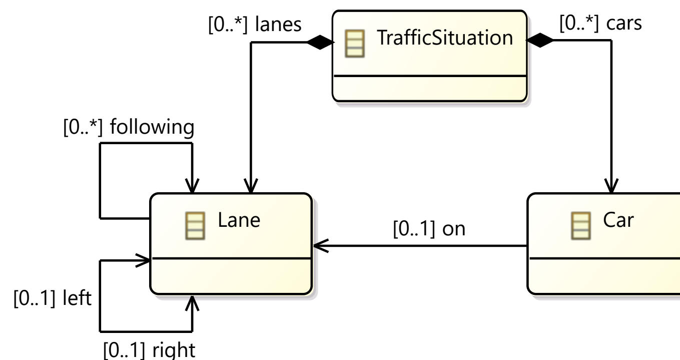


Figure 5.1: Metamodel used in the measurements

```
class Lane {
    Lane[0..*] following
    Lane[0..1] left opposite right
    Lane[0..1] right opposite left
}
class Car {
    Lane[0..1] on
}
```

Listing 5.1: Implementation of the metamodel using the new syntax

5.2.1 Patterns in VIATRA

For the measurements using VIATRA I implemented the required graph patterns in VQL, as shown in Listing 5.2.

```
pattern closeCars(car1: Car, car2: Car) {
    // same lane
    Car.on(car1,lane);
    Car.on(car2,lane);
    car1 != car2;
} or {
    // following lane
    Lane.following(lane1,lane2);
    Car.on(car1,lane1);
    Car.on(car2,lane2);
} or {
    // neighboring lanes
    Lane.left(lane1,lane2);
    Car.on(car1,lane1);
    Car.on(car2,lane2);
}

pattern followingLane(from: Lane, to: Lane) {
    Lane.following(from,to);
}
pattern carOnLane(car: Car, lane: Lane) {
    Car.on(car,lane);
}

pattern spawnCar(lane : Lane) {
    neg find followingLane(_,lane);
    neg find carOnLane(_,lane);
}
pattern despawnCar(lane : Lane, car: Car) {
    neg find followingLane(lane,_);
    find carOnLane(car,lane);
}

pattern moveCar(from: Lane, to: Lane, car: Car) {
    Lane.following(from,to);
    Car.on(car,from);
} or {
    Lane.left(from,to);
    Car.on(car,from);
} or {
    Lane.right(from,to);
    Car.on(car,from);
}
```

Listing 5.2: VQL implementation of the graph patterns

5.2.2 Patterns with the new syntax

The predicates implementing the graph patterns in the new specification language are shown in Listing 5.3. I chose to use direct predicates, so I could specify to match for **true** and **unknown** values. These predicates are semantically equivalent to the ones implemented in VQL.

```
direct pred closeCars(car1, car2) <->
    on(car1,lane)=true|unknown,
    on(car2,lane)=true|unknown,
    equals(car1,car2)=false
  ; following(lane1,lane2)=true|unknown,
    on(car1,lane1)=true|unknown,
    on(car2,lane2)=true|unknown
  ; left(lane1,lane2)=true|unknown,
    on(car1,lane1)=true|unknown,
    on(car2,lane2)=true|unknown.

direct pred hasFollowing(from) <->
    following(from,_to) = true|unknown.

direct pred hasPrevious(to) <->
    following(_from,to) = true|unknown.

direct pred carOnLane(car,lane) <->
    on(car,lane) = true|unknown.

direct pred spawnCar(lane) <->
    Lane(lane) = true|unknown,
    !hasPrevious(lane),
    !carOnLane(_,lane).

direct pred despawnCar(lane,car) <->
    !hasFollowing(lane),
    carOnLane(car,lane).

direct pred moveCar(from,to,car) <->
    following(from,to) = true|unknown,
    on(car,from) = true|unknown
  ; left(from,to) = true|unknown,
    on(car,from) = true|unknown
  ; right(from,to) = true|unknown,
    on(car,from) = true|unknown.
```

Listing 5.3: Implementation of the graph patterns using the new syntax

5.3 Measurement setup

The measurement workflow is shown in Figure 5.4. The measurements have three parameters:

- x: The number of lanes following each other.
- y: The number of parallel lanes.
- n: The number of times the changes are applied and pattern matching is executed.

First, I initialize the predicates, an empty model and the query engine in the Init phase. Next, in the Build step I build up the model, which consists of $x * y$ lanes in a grid, and y cars placed randomly on these lanes. Figure 5.2 shows an example of a four by four grid of lanes with four cars. The forward direction is to the right, and the arrows show which lane each car is able to move to.

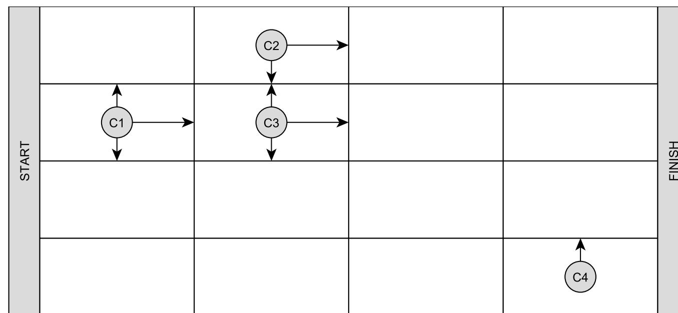


Figure 5.2: An example model of a four by four grid of lanes with four cars

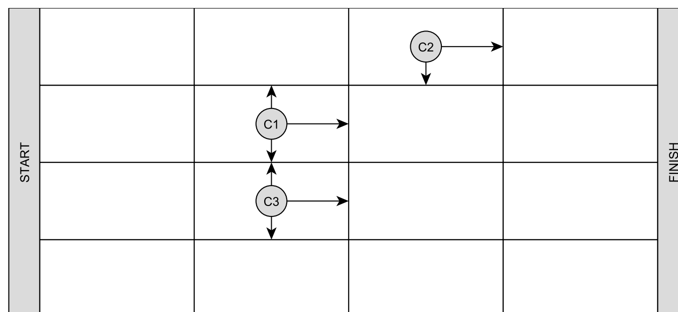


Figure 5.3: An example model of a four by four grid of lanes with three cars after despawning and moving

In the next step an iteration starts, where I first despawn all cars that are on a lane that has no following lane. In Figure 5.2 car C4 would be removed, since it can no longer go forward. Next I move all remaining cars from their current lane to its following lane or either of the lanes next to them. After this step the example might look like Figure 5.3. The third and final step in the iteration is spawning new cars to make sure there are y cars in the model. These new cars are placed randomly on the lanes. These three steps are repeated n times. After n iterations the measurement stops. The runtime of each step is measured separately.

I ran the same measurement using both the original VIATRA and the new extension. I ran the measurement on five different model sizes, ran the model modification for 5000 iterations, while saving the runtime after every 1000 iterations. Before the measurement

of both tools, I ran a similar, but smaller setup to account for the JVM warm up. Each measurement was repeated 25 times, and I used the median value of the results, to filter out the noise.

Model sizes used for measurement:

- **50:** 50x50 lanes, 50 cars (10100 nodes+edges)
- **100:** 100x100 lanes, 100 cars (40200 nodes+edges)
- **150:** 150x150 lanes, 150 cars (90300 nodes+edges)
- **200:** 200x200 lanes, 200 cars (160400 nodes+edges)
- **250:** 250x250 lanes, 250 cars (250500 nodes+edges)
- **500:** 500x500 lanes, 500 cars (1001000 nodes+edges)
- **750:** 750x750 lanes, 750 cars (2251500 nodes+edges)
- **1000:** 1000x1000 lanes, 1000 cars (4002000 nodes+edges)
- **1250:** 1250x1250 lanes, 1250 cars (6252500 nodes+edges)

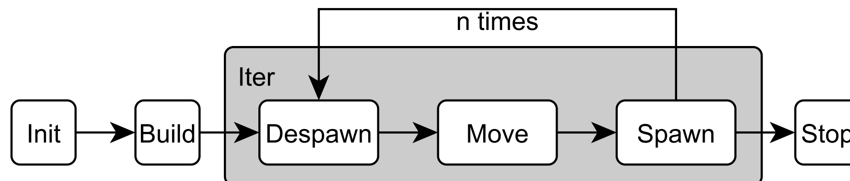


Figure 5.4: Measurement setup

For the measurements I used the following hardware, software versions, and settings:

- Java version: 17
- Maximum Java heap size: 8GB
- VIATRA version: 2.6.0
- Windows 10
- CPU: Intel Core i7-9750H

5.4 Measurement results

Figure 5.5a shows how the runtime changes if we increase the number of nodes and edges in the model used in the setup. The horizontal axis is the size of the model on which I ran the pattern matching, as detailed above. The vertical axis shows the time it took to complete 5000 iteration of modification on the model, in milliseconds.

Figure 5.5a shows the runtime the runtime of one iteration, measured between iterations 4000 and 5000. The horizontal axis is the nodes and edges in the model on which I ran the

pattern matching, as detailed above. The vertical axis shows the time it took to complete one iteration of modification on the model, in milliseconds.

For Figure 5.5c, I used the measurement result from the 1250x1250 model size. The horizontal axis shows the number of iterations completed, while the vertical axis is the execution time in milliseconds.

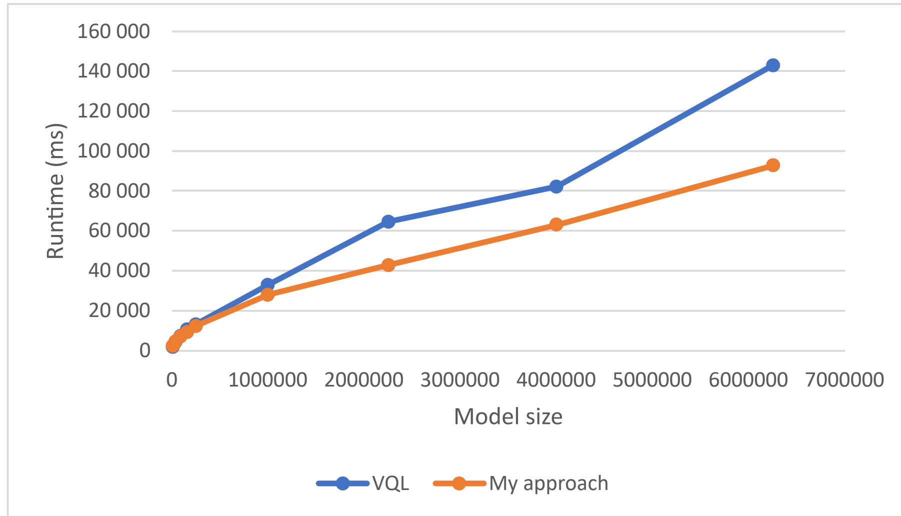
5.5 Discussion of the results

As we can see in Figure 5.5, all three types of measurement show that my solution not only kept the performance of VIATRA, on large models it outperformed it, even on custom datastructures. From Figure 5.5a we can see that on smaller model sizes there is no significant difference between my approach and VIATRA, however on larger models my solution is about 35% faster ().

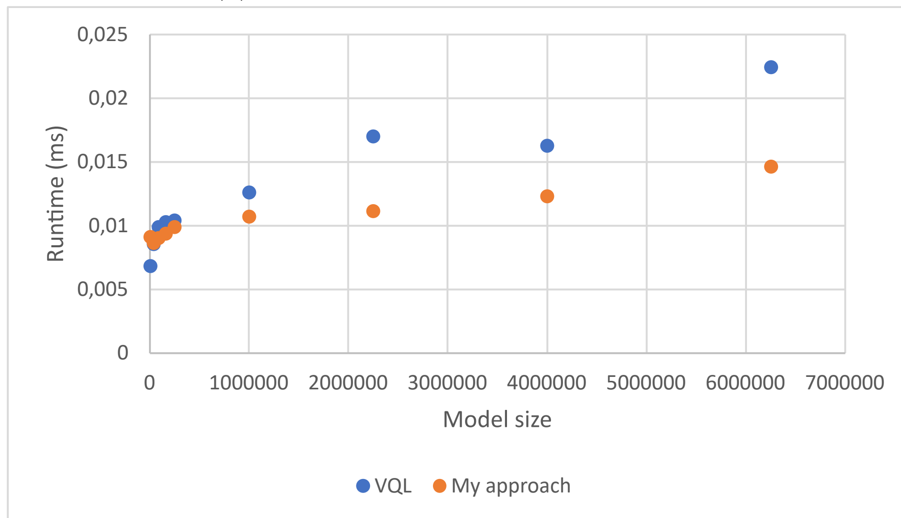
On Figure 5.5b we can see that the size of the model has only negligible effect on the execution time of performing a query and applying a change to the model, as it is nearly constant. Figure 5.5c shows the same result, the total executing time linearly increases as we increase the model size.

RQ1 With respect to the model size, my solution scales better than the original VIATRA. The runtime increases close to linearly with the size of the model.

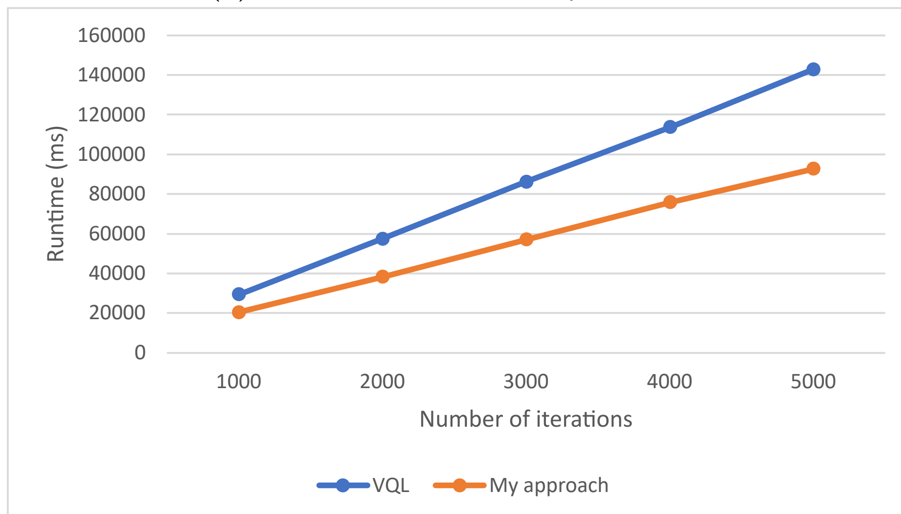
RQ2 The runtime of the pattern matching linearly increases with the number of changes applied. A modification on the model takes the same amount of time at the beginning, as after many iterations.



(a) Change in total runtime by model size



(b) Runtime of one iteration by model size



(c) Change in total runtime by number of iterations

Figure 5.5: Runtime measurements

Chapter 6

Conclusions and future work

Graph generator algorithms are used in a multitude of scientific disciplines, as well as the development and testing of software, and critical systems. Since these use cases usually involve generating a substantial amount of models, often with immense sizes, it is crucial to improve the performance of the graph generator algorithm as much as possible. It is also important to make the solution portable, since requiring a difficult setup may deter some from using our solution, even if it best suits their needs. The key elements of my work are the following:

- I designed and implemented a solution to execute pattern matching on custom datastructures using VIATRA.
- I created a way to transform predicates written in disjunctive normal form (DNF) into a form that is expected by VIATRA.
- I demonstrated the use of graph generation, on a complex case study. This shows a modular testing approach for testing autonomous systems, which utilizes graph generation in several of its steps.
- I compared the performance of my solution to the original VIATRA by applying small changes to different sizes of models and measuring the runtime over thousands of iterations.

Future work I intend to further evaluate the performance of the new predicate evaluator and custom datastructures on an established benchmark [43], and compare it to the original VIATRA, as well as other existing query technologies.

Acknowledgements

I would like to thank the co-author of my Students' Scientific Conference report, Balázs Pintér, and his advisor, dr. András Vörös. I also want to thank Aren A. Babikian for his contribution in modeling and generating situations, which we used in the Students' Scientific Conference report, and was also part of my case study. Finally, I would like to thank my advisor, dr. Oszkár Semeráth, for all the help and guidance in the past years.

List of Figures

2.1 Functional overview of a generic ADAS component. 6
2.2 Metamodel of a simple traffic situation 8

3.1 Logic values 16
3.2 Truth tables of information merge and logical connectives 16
3.3 Class diagram of DNF classes 17
3.4 Partial model with unknown relation values 19
3.5 Class diagram of some of the classes created for pattern matching on custom datastructures. 22
3.6 Processing changes applied to the model 23

4.1 Functional overview 26
4.2 Metamodel of the road network 27
4.3 Metamodel of the road network used in [22] 28
4.4 Map generation methods 29
4.5 Metamodel of the abstract situation 31
4.6 A model of an abstract situations and two examples of it in the simulator . 32
4.7 Metamodel of the perturbations 33
4.8 Layout of an abstract situation 34
4.9 Static object comparison 34
4.10 Weather conditions in Carla 35
4.11 Execution 35

5.1 Metamodel used in the measurements 37
5.2 An example model of a four by four grid of lanes with four cars 40
5.3 An example model of a four by four grid of lanes with three cars after despawning and moving 40
5.4 Measurement setup 41
5.5 Runtime measurements 43

A.1.1 Testing workflow 57

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] Road vehicles – functional safety, 2018.

[5] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Rev. Mod. Phys.*, 74:47–97, Jan 2002. DOI: 10.1103/RevModPhys.74.47. URL https://link.aps.org/doi/10.1103/RevModPhys.74.47.

[6] 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.

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

[8] 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.

[9] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999. DOI: 10.1126/science.286.5439.509. URL https://www.science.org/doi/abs/10.1126/science.286.5439.509.

[10] Nuel D. Belnap. *A Useful Four-Valued Logic*, pages 5–37. Springer Netherlands, Dordrecht, 1977. ISBN 978-94-010-1161-7. DOI: 10.1007/978-94-010-1161-7_2. URL https://doi.org/10.1007/978-94-010-1161-7_2.

[11] Gábor Bergmann, András Ökrös, István Ráth, Dániel Varró, and Gergely Varró. Incremental pattern matching in the viatra model transformation system. In *Proceedings of the Third International Workshop on Graph and Model Transformations*, GRaMoT '08, page 25–32, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605580333. DOI: 10.1145/1402947.1402953. URL https://doi.org/10.1145/1402947.1402953.

[12] Gábor Bergmann, Zoltán Ujhelyi, István Ráth, and Dániel Varró. A graph query language for emf models. In Jordi Cabot and Eelco Visser, editors, *Theory and Practice of Model Transformations*, pages 167–182, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-21732-6.

[13] 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.

[14] Marsha Chechik, Shiva Nejati, and Mehrdad Sabetzadeh. A relationship-based approach to model integration. *Innovations in Systems and Software Engineering*, 8 (1):3–18, Mar 2012. ISSN 1614-5054. DOI: 10.1007/s11334-011-0155-2. URL https://doi.org/10.1007/s11334-011-0155-2.

[15] On-Road Automated Driving (ORAD) committee. *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*, apr 2021. URL https://doi.org/10.4271/J3016_202104.

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

[17] George Dimitrakopoulos. *The Future: Towards Autonomous Driving*, pages 113–121. Springer International Publishing, Cham, 2017. ISBN 978-3-319-47244-7. DOI: 10.1007/978-3-319-47244-7_6. URL https://doi.org/10.1007/978-3-319-47244-7_6.

[18] 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.

[19] 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.

[20] P. Erdős and A Rényi. On the evolution of random graphs. In *PUBLICATION OF THE MATHEMATICAL INSTITUTE OF THE HUNGARIAN ACADEMY OF SCIENCES*, pages 17–61, 1960.

[21] Attila Ficsor and Balázs Pintér. Simulation-based robustness testing of adas systems. In *Student Research Societies Report*, 2021.

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

[23] Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982. ISSN 0004-3702. DOI: https://doi.org/10.1016/0004-3702(82)90020-0. URL https://www.sciencedirect.com/science/article/pii/0004370282900200.

[24] 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.

[25] Rafael Gómez-Bombarelli, Jennifer N. Wei, David Duvenaud, José Miguel Hernández-Lobato, Benjamín Sánchez-Lengeling, Dennis Sheberla, Jorge Aguilera-Iparraguirre, Timothy D. Hirzel, Ryan P. Adams, and Alán Aspuru-Guzik. Automatic chemical design using a data-driven continuous representation of molecules. *ACS Central Science*, 4(2):268–276, 2018. DOI: 10.1021/acscentsci.7b00572. URL https://doi.org/10.1021/acscentsci.7b00572. PMID: 29532027.

[26] Paul W. Holland, Kathryn Blackmond Laskey, and Samuel Leinhardt. Stochastic blockmodels: First steps. *Social Networks*, 5(2):109–137, 1983. ISSN 0378-8733. DOI: https://doi.org/10.1016/0378-8733(83)90021-7. URL https://www.sciencedirect.com/science/article/pii/0378873383900217.

[27] 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.

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

[29] Norihiro Kamide and Hitoshi Omori. An extended first-order belnap-dunn logic with classical negation. In Alexandru Baltag, Jeremy Seligman, and Tomoyuki Yamada, editors, *Logic, Rationality, and Interaction*, pages 79–93, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg. ISBN 978-3-662-55665-8.

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

[31] 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.

[32] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker graphs: An approach to modeling networks. *J. Mach. Learn. Res.*, 11:985–1042, mar 2010. ISSN 1532-4435.

[33] Yibo Li, Liangren Zhang, and Zhenming Liu. Multi-objective de novo drug design with conditional graph generative model, 2018.

[34] Yujia Li, Oriol Vinyals, Chris Dyer, Razvan Pascanu, and Peter Battaglia. Learning deep generative models of graphs, 2018.

[35] Renjie Liao, Yujia Li, Yang Song, Shenlong Wang, Charlie Nash, William L. Hamilton, David Duvenaud, Raquel Urtasun, and Richard S. Zemel. Efficient graph generation with graph recurrent attention networks, 2020.

[36] 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.

[37] Kristóf Marussy, Oszkár Semeráth, Aren A. Babikian, and Dániel Varró. A specification language for consistent model generation based on partial models. *J. Object Technol.*, 19:3:1–22, 2020.

[38] 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.

[39] OMG. OMG Object Constraint Language (OCL), Version 2.4, February 2014. URL `http://www.omg.org/spec/OCL/2.4/`.

[40] Marwin H. S. Segler, Thierry Kogej, Christian Tyrchan, and Mark P. Waller. Generating focused molecule libraries for drug discovery with recurrent neural networks. *ACS Central Science*, 4(1):120–131, 2018. DOI: `10.1021/acscentsci.7b00512`. URL `https://doi.org/10.1021/acscentsci.7b00512`. PMID: 29392184.

[41] 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.

[42] Oszkár Semeráth, Aren A. Babikian, Sebastian Pilarski, and Dániel Varró. Viatra solver: A framework for the automated generation of consistent domain-specific models. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 43–46, 2019. DOI: `10.1109/ICSE-Companion.2019.00034`.

[43] Gábor Szárnyas, Benedek Izsó, István Ráth, and Dániel Varró. The train benchmark: cross-technology performance evaluation of continuous model queries. *Software & Systems Modeling*, 17(4):1365–1393, Oct 2018. ISSN 1619-1374. DOI: `10.1007/s10270-016-0571-8`. URL `https://doi.org/10.1007/s10270-016-0571-8`.

[44] 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.

[45] 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.

[46] 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`.

[47] 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.

[48] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, Jun 1998. ISSN 1476-4687. DOI: `10.1038/30918`. URL `https://doi.org/10.1038/30918`.

[49] 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.

[50] Saining Xie, Alexander Kirillov, Ross Girshick, and Kaiming He. Exploring randomly wired neural networks for image recognition, 2019.

[51] 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.

[52] Jiaxuan You, Bowen Liu, Rex Ying, Vijay Pande, and Jure Leskovec. Graph convolutional policy network for goal-directed molecular graph generation, 2019.

[53] G. Udny Yule. A mathematical theory of evolution, based on the conclusions of dr. j. c. willis, f.r.s. *Philosophical Transactions of the Royal Society of London. Series B, Containing Papers of a Biological Character*, 213:21–87, 1925. ISSN 02643960. URL http://www.jstor.org/stable/92117.

Appendix

A.1 Workflow in operation

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

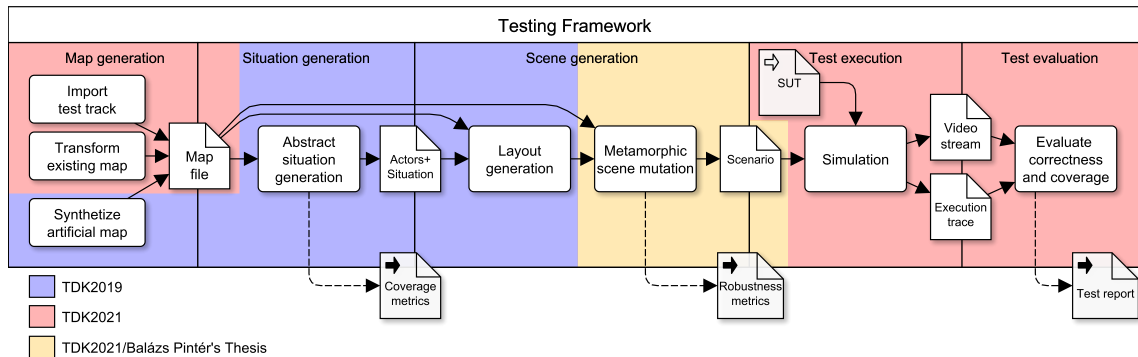


Figure A.1.1: Testing workflow

A.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.

A.1.2 Abstract situation generation

The generation of abstract situation is based on the metamodel in Figure 4.5. 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.

A.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 51.** 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
>     # right of: relative to orientation of
>     # OrentedPoint in the edge of roads
>     obj = ATM right of spawnPoint by Range(0.5,1.5)
>     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 52.** 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 @ -57.03033065795898),
>     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 4.2.2. 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 53.** 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 54.** 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
> ```

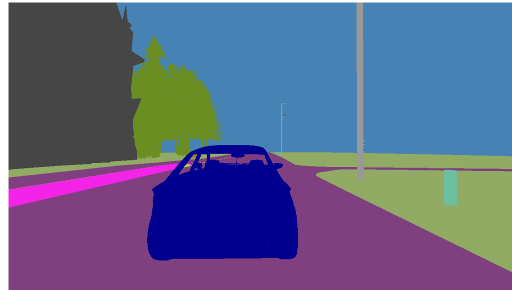
Let's see what all the arguments mean in Example 54:

- `.\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

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.

(a) RGB image from Carla



(b) Semantic segmentation image from Carla

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 A.1.2b (e.g., building in a different color than vehicles). This gives us the ground truth needed for the evaluation of an object detection AI.

A.2 Scenic

Scenic [3, 24] 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.

A.2.1 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.

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.

A.3 CARLA

CARLA [1, 18] 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.

A.3.1 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.

A.3.2 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.

A.3.3 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.

A.3.4 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.

A.3.5 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