Karlsruhe Institute of Technology

Ontology-based corner case scenario simulation for autonomous driving

Bachelor thesis

Stefani Guneshka

Department of Informatics
Institute for Program Structures and Data Organization
and
FZI Research Center for Information Technology

Reviewer:	Prof. Dr. R. Reussner
Second reviewer:	Prof. Dr.–Ing. J. M. Zöllner
Advisor:	Daniel Bogdoll, M.Sc.

Research Period: 01. October 2021 – 01. March 2022

Ontology-based corner case scenario simulation for autonomous driving

by
Stefani Guneshka

Bachelor thesis
March 2022

FZI

Bachelor thesis, FZI
Department of Informatics, 2022
Reviewers: Prof. Dr. R. Reussner, Prof. Dr.–Ing. J. M. Zöllner

Department Technical Cognitive Systems
FZI Research Center for Information Technology

Affirmation

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Karlsruhe,
March 2022

Stefani Guneshka

Abstract

Safety assessment of autonomous driving functions is an emerging topic in the automotive industry. In order to launch an autonomous vehicle on the road, the system running it needs to be able to react adequately in most of the cases, just like a human driver, if not even better. Furthermore, the amount of existing data that contains corner cases is nowhere near the amount of data needed to sufficiently train autonomous driving systems. The identification, classification and generation of corner cases for autonomous driving is a crucial part of scenario-based validation. In today's world, a method that describes, generates and classifies those at the same time, is not available.

In this thesis, a method for the description and generation of corner cases for autonomous driving is proposed. The method uses a template ontology, as a base for the scenario generation of corner cases with established definitions. The proposed approach also allows combining already described scenarios into new ones without any further actions. For an easy usage of the template ontology, an OntologyGenerator library for the simplified generation of corner cases is provided. The library completely eliminates the need for the user to work directly with the ontology and provides a method for creating scenarios even for users with little or no experience with ontologies. Moreover, the proposed approach is clean and understandable, making it easily scalable and extensible. The technique, follows the structure of OpenSCENARIO, thus making it easy to use, if one already understands OpenSCENARIO. In addition, the proposed ontology provides the base for the classification of the described scenarios to a specific corner case category. With the proposed technique, I was able to describe and generate seven scenarios, one for each of the earlier mentioned corner case categories. The resulting ontologies(created using the template ontology), that describe different corner cases, deliver correctly defined OpenSCENARIO files, after going through the Onto2OpenSCENARIOConverter. The resulted files deliver a correct simulation within CARLA, corresponding to the scenario description, which shows that the proposed method functions as expected.

Contents

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Objectives . . . 2
- 1.3 Structure outline . . . 2

2 Foundations 3
- 2.1 Corner Case . . . 3
- 2.2 Ontology . . . 4
  - 2.2.1 Terminological Boxes . . . 5
  - 2.2.2 Assertional Boxes . . . 6
  - 2.2.3 Reasoner . . . 6
    - 2.2.3.1 Open World Assumption and Closed World Assumption . . . 7
  - 2.2.4 Protégé . . . 7
  - 2.2.5 Semantic Web Rule Language Rules . . . 7
    - 2.2.5.1 SWRL Rules Syntax . . . 8
- 2.3 Scenario . . . 9
  - 2.3.1 Simulation . . . 9
    - 2.3.1.1 Simulation Environment . . . 9
- 2.4 ASAM OpenSCENARIO Format . . . 10
  - 2.4.1 OpenSCENARIO Scenario Elements . . . 10
    - 2.4.1.1 Triggers . . . 12
      - 2.4.1.1.1 Conditions . . . 12
    - 2.4.1.2 Environment . . . 13
      - 2.4.1.2.1 Weather . . . 13
    - 2.4.1.3 EnvironmentAction . . . 13
    - 2.4.1.4 BoundingBox . . . 13

3 Related Work 14
- 3.1 Pure scenario description ontologies . . . 14
- 3.2 Ontologies in the field of decision-making and scenario description . . . 15
- 3.3 Conclusion . . . 18

4 Approach 19
- 4.1 Exploration Phase . . . 19
  - 4.1.1 Categories and examples . . . 20

4.1.1.1 Pixel Level 20
4.1.1.2 Domain Level 20
4.1.1.3 Object Level 20
4.1.1.4 Scene Level 20
4.1.1.5 Scenario Level 21
4.2 Template Ontology 21
4.2.1 Classes 22
4.2.2 Object Properties 22
4.2.3 Data Properties 23
4.2.4 Constants 24
4.2.5 Default Individuals 24
4.3 Corner Case ontology 25
4.3.1 SWRL Rules 26
4.3.2 Requirements for a correctly defined scenario 26
4.4 OntologyGenerator 27
4.4.1 Approach 27
4.4.2 Implementation 29
4.4.2.1 Functions 29
4.4.2.1.1 TeleportAction 29
4.4.2.1.2 New Entity 29
4.5 Onto2OpenSCENARIOConverter 30
4.5.1 Implementation 30
4.5.1.1 Init Initialization 31
4.5.1.2 Story initialization 32
4.5.1.3 Combination of scenarios 34
4.5.1.4 Discussion 34

5 Qualitative Evaluation **36**
5.1 Dead Pixel 36
5.2 Domain Shift scenario 36
5.3 Single-Point Anomaly scenario 37
5.4 Collective Anomaly 37
5.5 Contextual Anomaly 38
5.6 Risky Scenario 38
5.7 Novel Scenario 38
5.8 Anomalous Scenario 39
5.9 Combinatorial Scenarios 39

6 Conclusion **40**
6.1 Challenges 40
6.2 Future Work 41

Appendices **42**

A Functions **44**

A.0.1 getNameFromIri 44

A.1 Predicates 44

B OpenSCENARIO further attributes **45**

B.1 Priority 45

C Corner case categories and simulation results with pictures **46**

D Figures **49**

A List of Figures **59**

B List of Tables **60**

C Bibliography **61**

Acronyms

CARLA Car Learning to Act

SAE Society of Automotive Engineers

XML Extensible Markup Language

ODD Operational Design Domain

RONNY ROad Network oNtologY

lidar Light Detection and Ranging

SWRL Semantic Web Rule Language

OWA Open World Assumption

CWA Closed World Assumption

OWL Web Ontology Language

W3C World Wide Web Consortium

1 Introduction

1.1 Motivation

Autonomous driving is a very popular topic in today's world [59, 15**?** , 54]. More and more companies and car producers work around the clock in an attempt to launch the first level 4 on-road vehicle, as classified by the Society of Automotive Engineers (SAE) [12]. The SAE taxonomy has six levels, starting at level 0 which is a fully manual vehicle and ending with level 5 – a fully autonomous vehicle [12]. For the purpose of this thesis it is important to note that SAE level 4 means "No human interaction required" [33], in other words almost everything is autonomous, but the driver should be in condition to react, if something unexpected happens. The need of act, changes with level 5. Level 5 means a fully autonomous vehicle. Those vehicles are without limitations to the Operational Design Domain (ODD) and the need for any human assistance for any of its features [12]. One of the essential parts of the autonomous driving system is its ability to avoid accidents. In today's world, various validation methods exist [30, 65], but it's still not 100% clear which is the best way to do validate autonomous driving [30]. When thinking of validating and testing vehicles, the first thing that comes to mind is to perform a test drive. This leads us to the first type of validation – distance-based validation [16]. According to Shalev-Schwartz et al. [62], the probability of a fatal road accident per human driving hour is 10^{-6}. This may seem like a low probability of an accident, but people demand maximum safety when their lives are at stake. They also suggest that the probability value should decrease to 10^{-9}.
The quantity of data required to ensure a probability of 10^{-9} mortality is 10^9 hours. This is equivalent to about 50 billion km. Following their argumentation, one can see fast, that this amount of data is too much to be acquired using only real-world data, as also shown by Amersbach and Winner [16] this validation method is just not feasible.

When it comes to the validation of autonomous vehicles, it is important to mention the PEGASUS project [61]. They contribute significantly to this field by proposing various validation methods. One of those methods is the scenario-based validation. This method is also discussed by Amersbach and Winner [16]. Scenario-based validation, however, also has its weaknesses – due to the enormous amount of parameters in a single scenario, the possible combinations are also infeasible[16]. With this said, it's furthermore possible to have tons of data, which is not at all relevant for the autonomous driving system, or instead of generalizing, the system might overfit [29, 56]. This leads to the conclusion that the amount of existing data containing corner cases is nowhere near the amount needed to sufficiently train autonomous driving systems. For that reason, this thesis contributes to the description and generation of corner cases for autonomous driving systems. There are different ways to generate data and describe scenarios. For example, in his publication, Ishida [46] presents the “Q Language”, which is a Lisp based language, and is specially created to describe scenarios. However, the proposed by them language, is more of a language for the description of all day activities and less for autonomous driving. Furthermore, it doesn’t provide a reasoning machine, like the following method does. Ontologies provide knowledge bases that are understandable by humans and computers [38]. With the help of an ontology, a machine-readable format can be created as the basis for an automated scenario generation. For re-usability purposes, the scenario generation is achieved with the help of the ASAM OpenSCENARIO format (See Section 2.4). Since OpenSCENARIO is tool-independent, the simulator used for the scenarios is the Car Learning to Act (CARLA) Simulator (See Section 2.3.1.1).

1.2 Objectives

The first goal of this thesis is the detailed design and implementation of an ontology. The ontology should be able to describe various scenarios that contain corner cases. To achieve this, the different components of a scenario were explored and a deep dive into the OpenSCENARIO format was made.

The second goal of this work is to enable an autonomous agent to drive within the described scenarios and obtain its results. It is important to note, the behavior of the agent is out of the scope of this thesis, and an ego vehicle with autopilot was used.

1.3 Structure outline

In Chapter2 the theoretical foundations, which were used during the work on this thesis, are explained. In detail, what is an ontology and all its components. Furthermore, terms are defined. In Chapter 3 I go over the related work and show the existing research gap. Chapter 4 explains the main approach used to solve the research gap, and the template ontology is presented. Chapter 4 focus further on the detailed explanation of the main approach and its components. Furthermore, it presents the **OntologyGenerator**, which helps by the generation of an ontology, that contains a description of a corner case. At last, Chapter 4 explains the algorithm hidden behind the **Converter**, which generates an OpenSCENARIO file from the earlier generated ontology. In Chapter 5 I show and discuss the resulted ontologies and the simulations described by them. The last Chapter 6 gives final words and future work outlook.

2 Foundations

This chapter provides definitions of the various terms used in this thesis. It also explains the basics needed to understand the approach presented in this thesis. In the first part of the chapter, the term corner case and the chosen corner case categories are explained. Afterwards, the term ontology with all its components are presented. At last, the term simulation, together with the used simulation environment and the used for the simulations format(OpenSCENARIO [20]) are explained.

2.1 Corner Case

Since this thesis aims to improve the corner case description and data generation, it is essential to define the term "corner case". According to Termöhlen et al. [66], a corner case is "a relevant object (class) in a relevant location that a modern autonomous driving system cannot predict". Another proposal for the term "corner case" is given by Breitenstein et al. in [14], where a corner case is seen as an "unusual and potentioally dangerous circumstances".

Heidecker et al. [41] propose another definition based on the publication of Breitenstein et al. [26, 14], where a corner case is "rare or a never considered case or scene" [41]. All definitions are similar, but based on the chosen corner cases categories in this thesis, I adopted the definition provided by Heideldecker et al. [41]. Having the term "corner case" defined, in the following paragraph, I will first present the corner case categories introduced by Breitenstein et al. [14, 26]. The corner cases are divided in five main levels, beginning from the bottom level - **Pixel Level, Domain Level, Object Level, Scene Level** and at last the **Scenario Level**. Furthermore, with every level, the detection complexity for each corner case increases [26]. In the following, I will explain each level and the categories within it.

The first level is the **Pixel Level**, which is defined as "*(Perceived) errors in data*" [26]. The **Pixel Level** has two categories - **Global Outlier**, which is defined as "*All or many pixels fall outside of the expected range of measurement*" [26] and **Local Outlier**, which is defined as "*One or few pixels fall outside of the expected range of measurement*" [26].

The second level is the **Domain Level**, defined as "*World model fails to explain observations*" [26] and it consists only of one category - the **Domain Shift**, introduced as "*A large, constant shift in appearance, but not in semantics*" [26].

The third level is the **Object Level**, defined as "*Instances that have not been seen before* " [26]. This level also contain only one subcategory - the **Single-Point Anomaly**, which is defined as "*An unknown object*" [26]. This would just mean that an unknown object appears into the scene.

The fourth level is the **Scene Level**, and it's described as "*Non-conformity with expected patterns in a single image*" [26]. In this level, there are two subcategories - the **Collective Anomaly**, de-

fined as "*Multiple known objects, but in an unseen quantity*" [26] and the **Contextual Anomaly**, described as "*A known object, but in an unusual location*" [26].
The last level is the **Scenario Level**, which is defined as "*Patterns are observed over the course of an image sequence*" [26]. The **Scenario Level** consists of three subcategories - **Risky Scenario, Novel Scenario** and **Anomalous Scenario**. The **Risky Scenario** is defined as "*Pattern that was observed during the training process, but still contains potential for collision*" [26].
Furthermore, the **Novel Scenario** is seen as "*Pattern that was not observed during the training process, but does not increase the potential for collision*" [26] and at last the **Anomalous Scenario** is defined as "*Pattern that was not observed during the training process and has high potential for collision*". The same categories and definitions are also later on discussed by Breitenstein et al. [14]. In addition, based on those publications, later on Heidecker et al. [41] present a slightly different categorization, where the authors introduce three new layers and put the previous mentioned categories within those layers. Namely, the **Pixel Level** goes into the so-called **Sensor Layer**, the **Domain Level, Object Level** and **Scene Level** are presented under the **Content Layer** and at last, the **Scenario Level** is part of the newly introduced **Temporal Layer** [41]. For a graphical overview of the scenarios, please refer to Figure C.1 in the Appendix.

2.2 Ontology

Ontology research has gained much popularity in computer science in recent years. The term "Ontology" originates from philosophy, but computer scientists also defined the term on their own. According to Guarino et al. [39] an ontology is "a formal, explicit specification of a shared conceptualization". Noy and McGuinness give their own definition of the term - "an ontology is a formal explicit description of concepts in a domain of discourse (classes (sometimes called concepts)), properties of each concept describing various features and attributes of the concept (slots (sometimes called roles or properties)), and restrictions on slots (facets (sometimes called role restrictions))" [51]. Another definition is given by Zúñiga [71] which states that an ontology is a "formal language designed to represent a particular domain of knowledge". In this thesis, I adopt the definition presented by Zúñiga [71], because it is precise and clear. As stated by Zúñiga [71], an ontology is a way to represent knowledge. Furthermore, it consists of entities, object properties, data properties, and individuals. For more information, refer to Section 2.2.1 and 2.2.2. An ontology defines a set of concepts within a domain and the relationships between the distinct entities within this domain. The reason for the emerging interest in ontologies is the many applications of ontology in today's world [51, 40, 31]. An ontology provides not only a human-readable and a machine-readable format, but also the so-called reasoner exists for it, which works on the ontology and checks it for inconsistencies, so the user is able to define the ontology as correct as possible. For further information on the reasoner, please refer to Section 2.2.3.

An ontology consists of two main parts – the terminological box and the assertional box. Basically, the terminological box "defines the terminology of the application domain" [22] and the assertional box "states facts about a 'specific world'" [22]. Both of them, together, build the

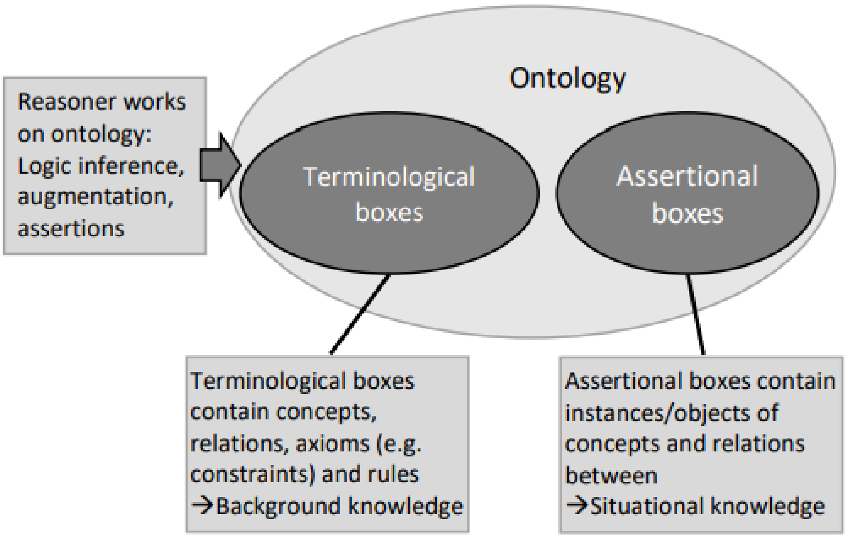

Ontology
Reasoner works on ontology: Logic inference, augmentation, assertions
Terminological boxes
Assertional boxes
Terminological boxes contain concepts, relations, axioms (e.g. constraints) and rules →Background knowledge
Assertional boxes contain instances/objects of concepts and relations between →Situational knowledge

Figure 2.1: Overview of the architecture of an ontology [23]

knowledge base of the ontology. In addition, a reasoner works with the ontology and provides logical conclusions, augmentations, and assertions(See Figure 2.1).

2.2.1 Terminological Boxes

The terminological boxes describe the fundamental a priori knowledge. It can be described with the help of classes (entities), object properties, data properties, and axioms[23]. Basically, with the help of an ontology, almost every sentence in the natural language can be described. A basic sentence is build from a substantives and verbs, for example, “The Woman wears a coat”. This sentence has two substantives (woman and coat) and one verb(wear). The same can be also represented in an ontology, with the classes *Woman* and *Coat* and the object property *wears*. Whereas the data properties would be then used in sentences, that have some kind of data type in them. In order to make the previous mentioned sentence complex, a data property can be added to the woman, i.e. “The woman is 20 years old”. The resulted knowledge then is “The 20 year old woman wears a coat”. In summary, the classes in an ontology represent substantives and the data/object properties represent the verbs. It is important also to note, that in every ontology with a hierarchy, which means that SubClass relationships exist within the ontology, there are 3 different levels of classes - top, middle and bottom [51]. The top level classes are the direct subclasses of **owl:Thing** and often the subclasses of those classes, if the ontology has a long and complex hierarchy. A bottom level class is every class which doesn’t have a subclass, and a middle level class are the classes in between. [51]. The axioms are the rules that are always true in the defined ontology. For example, this can be a simple rule like "Car is a Subclass of Vehicle", but they can also be much more complex, such as "A Teenager is equivalent to a person with a property has_age in range (13, 19)". Those rules are also called Web Ontology Language (OWL) Class Expressions [25]. The axioms are represented in the OWL, a semantic Web Language based on First-order Description

Logic [42]. It is important to note, that first-order description logic, is logic in which a statement or sentence is broken down into predicates and subjects.

2.2.2 Assertional Boxes

The Assertional Boxes are the instances of the classes, also called individuals, and the property assertions between the concrete individuals. The individuals are used for situational knowledge and representation of concrete world situations [23]. For example, we can create the individual “Jon” of class Person from the previously mentioned Terminological Box and give it the property assertion “has_age 14”. This is now an actual individual (instance). It is important to note, that if an individual has a property assertion, then simply the individual is connected(a relationship exists) to another individual or a data type via the previously mentioned property.

2.2.3 Reasoner

Nowadays, numerous reasoners exist and the decision which reasoner to use is crucial. However, to help with this decision, Dentler et al. [32] present a comparison of different reasoners.
A reasoner is a very powerful tool and in the following, I will explain how it works on an ontology. The reasoner is executed on both the Assertional and the Terminological box [23]. With its help, hidden knowledge can be found. For example, the class structure(the hierarchy) in the Terminological box can be changed. Different inconsistencies can be inferred, like classes that will never be instantiated or contradicting axioms.
Every rule(see Section 2.2.5) and axiom in the terminological box is applied to every individual in the assertional box, using the Tableau algorithm [22], which on its side test the rules and axioms and their satisfiability [68]. When the reasoner is finished, the individuals can have new assertions, for example, new class or new property assertions. With that said, if we were to run the reasoner on our theoretical Assertion and Terminology Boxes, the previously created individual Jon (See 2.2.2) would not only have the class “Person”, but the reasoner will also infer the class “Teenager”. Furthermore, the reasoner can also infer if two individuals are supposed to be the same. However, if it’s said in the assertional box that the individuals are different(by using the “different individuals” relationship), then it would lead to an inconsistency. An inconsistency means that some rules within the ontology contradict with each other, leading to an incorrectly defined ontology. Furthermore, as Wang and Yu state in [69], most of the reasoners are using Open World Assumption. The reasoner used in thesis (Pellet [63]) is also using the open world assumption. In the following section, I will briefly explain the differences between open and closed world assumption.

$$x = z + y$$

2.2.3.1 Open World Assumption and Closed World Assumption

The Open World Assumption (OWA) assumes that everything is “unknown”, unless stated otherwise [45]. If there is for example no connection between two entities and the reasoner makes the query to check if there is an existing connection between them, the answer wouldn’t be “no” as expected, but would be “unknown”. I.e., if something is not specifically said to be true, there is always the assumption that it might be true and thus not considered as false. With the help of OWA, it’s possible to infer hidden knowledge, because of its monotonic clause logic. A monotonic clause logic means that if there is knowledge to be inferred, it should be inferrable before a clause is added and after it was added. In other words, adding new information shouldn’t reduce the amount of inferred concepts, but only eventually increase it [55]. The Closed World Assumption (CWA) works with the so-called “Negation as failure”. That is, if something is not specifically said to be true, it’s assumed to be false. The information which is in a CWA is considered complete (because everything is defined one way or another). However, because of the non-monotonic logic in CWA, there is no possibility to infer **new** knowledge [55]. The previous statement also follows from the statement, that a CWA is considered complete. An example that illustrates the difference between CWA and OWA:
Statement: “Alex” “has” a “dog”.
Question: Does Johannes have a dog?
“Closed world Reasoner” answer: no.
“Open world Reasoner” answer: unknown.
The decision whether to work with CWA or OWA is crucial. From my research I concluded that OWA in our case is the better decision, because CWA is chosen when one has the complete information and in our case I want to infer information about the different corner cases, which would be impossible with a CWA, because of its completeness and non-monotonic logic [50].

2.2.4 Protégé

The used ontology editor in this thesis is Protégé – a free, open-source software [1]. With the help of this tool, an ontology in different formats, like *rdf* or *owl*, can be viewed, created and edited. The tool also has a variety of plugins that facilitate the editing of an ontology. In addition, there is an enormous selection of visualization tools within the Protégé tool such as OntoGraf, OWLViz, VOWL, etc. [36]
For visualization in this bachelor thesis, the tool OntoGraf is used, because it offers the greatest possibilities to visualize every part of an ontology.

2.2.5 Semantic Web Rule Language Rules

Semantic Web Rule Language (SWRL) is intended to be the rule language of the Semantic Web [24]. Those rules allow the implementation of logic in a given ontology. However, SWRL rules can’t change the hierarchy of an ontology itself, because the Reasoner only infers information about different individuals, using SWRL rules. I.e., for an individual, different properties can be inferred such that an adult is also a person, but this does not change the structure of the ontology as OWL Class expressions would.

2.2.5.1 SWRL Rules Syntax

In this section, I will explain the syntax of the SWRL rules and from where it originates and the idea behind its syntax. SWRL rules have a Horn-like syntax [11]. A definite Horn formula is a propositional formula in conjunctive normal form(CNF), for example :

$$(\neg A \vee \neg B \vee C) \wedge (\neg A \vee B \vee \neg C) \wedge (\neg A \vee \neg B \vee \neg C)$$

where “each disjunction contains at most one positive literal(a literal is every letter in a clause and it also always has a true or false value). Such a disjunction is called a definite Horn clause” [60]. To clarify, in the previously mentioned example, A, B and C are literals and $(\neg A \vee \neg B \vee C)$ is a clause. Furthermore, a conjunctive normal form, is a conjunction of disjunctions, like the expression shown above. Furthermore, a disjunction of literals with at most one positive would mean for example $X \vee \neg Y \vee \neg Z$. From the boolean logic it is known that

$$(\neg A \vee B) \Leftrightarrow (A \Rightarrow B)$$

and this is true for all kind of complex terms, which leads to

$$(X \vee \neg Y \vee \neg Z) \Leftrightarrow (\neg Y \vee \neg Z \vee X) \Leftrightarrow (\neg(\neg Y \vee \neg Z) \Rightarrow X) \Leftrightarrow (Y \wedge Z \Rightarrow X)$$

This is exactly the form of the SWRL Rules. The World Wide Web Consortium (W3C) refer to the left part of the implication as an Antecedent and the right part of the implication as Consequent [11]. The antecedent and consequent both consist of different predicates, which can be either data properties, object properties, classes, class expressions or the so called SWRL Built-Ins [11]. For a further information on predicates, please refer to Section A.1 in the appendix. It is important to note how does the reasoner work with the SWRL rules. Basically it checks for all individuals if they somehow satisfy xthe given predicates in the antecedent and if this is the case, the reasoner infers that everything, that is part of the consequent, should also be satisfied (true). In other words, the reasoner will assert the given conjunction of properties and classes in the consequent. Furthermore, it’s not allowed to use “not” in the SWRL rules syntax, because OWL supports “not” only in property or class expressions. If one tries to use it, then the error message “NOT may be used only in a class or property expression” appears. From this follows that in the Antecedent only “and” can be used as connection logic, and it’s not possible in a single rule to use a disjunction (logic “or”). There might not be direct disjunction, but the rules can be reformulated to match the requirements. Rules with conjunctions and disjunctions in the antecedent can be manually split into separate rules with the same consequent. The number of the resulting rules corresponds to the number of disjunctions in the original rule [52]. Furthermore, a disjunction in the consequent is not allowed.

2.3 Scenario

There are different definitions of the term scenario. As stated by Geyer et al. [35], the scenario term in the interaction design field was first introduced by Rosson and Carroll in their 2002 publication [58]. Rosson and Carroll state that a scenario is “simply a story about people carrying out an activity” [58]. According to [20], a scenario is “ a description of how the view of the world changes with time, usually from a specific perspective” [20]. Another definition is proposed by Go and Carroll: “Scenario is a description that contains (1) actors, (2) background information on the actors and assumptions about their environment, (3) actors goals or objectives, and (4) sequences of actions and events.” [37]. This definition is later on also discussed by Ulbrich et al. [67]. Ulbrich et al. however reject this scenario definition and suggest the following: “Scenario describes the temporal development between several scenes in a sequence of scenes. Every scenario starts with an initial scene. Actions and events as well as goals and values may be specified to characterize this temporal development in a scenario. Other than a scene, a scenario spans a certain amount of time”. Since in this thesis, the generation of simulations is achieved with the help of the OpenSCENARIO format, the chosen definition for scenario is the one, proposed by OpenSCENARIO [20].

2.3.1 Simulation

Since this thesis, does not only aim to describe corner cases for autonomous driving, but also to generate them, allowing the training of different autonomous functions, it is crucial to define the term scenario and furthermore, explain the used simulation environment.

2.3.1.1 Simulation Environment

Nowadays, the already released autonomous driving systems are equipped with all kinds of sensors, suggesting the need to generate various types of data [48]. For this reason, it is crucial, to choose the correct simulator. This would mean that the simulator support the retrieving of various types of data.

There are various simulators which are available, for example - AirSim [2], Virtual TestDrive [13], CARLA [3] and many more. The simulator AirSim is a simulator, that includes vehicles, drones and more [2]. The simulator enables the user to create different simulations, for vehicles and drones and to retrieve the data, within the simulation. This is available for both camera and lidar sensors. The Autonomous Driving Simulator CARLA [3] is a free open-source software specifically designed for autonomous driving simulations. CARLA also provides numerous sensors for retrieving data in its scenarios, especially both Light Detection and Ranging (lidar) and camera data.

Since this thesis is working with the OpenSCENARIO format (**xosc**), there is also the need of a converter from **xosc** to the chosen simulator.

ScenarioRunner [10] is an open-source execution engine that enables the user to execute a **xosc** file directly into the CARLA Simulator, given that the file is correctly defined.(See Section 4.3.2). Since this conversion is possible with the help of the open-source ScenarioRunner and CARLA provides more functions for autonomous driving vehicles(drones are not in the scope of this thesis), the chosen simulator in this thesis is CARLA and the used version is CARLA “0.9.13.”. Furthermore, the used ScenarioRunner version is “0.9.13.” as well.

2.4 ASAM OpenSCENARIO Format

ASAM OpenSCENARIO [20] is a file Extensible Markup Language (XML) format, which is used for “the description of the dynamic content of driving simulators” [18]. OpenSCENARIO is very popular nowadays and is used by many researchers in the field of autonomous driving [53, 57]. The format can be used in various simulation environments like CARLA ScenarioRunner [10], Esmini [6], Virtual Test Drive [13] etc.

2.4.1 OpenSCENARIO Scenario Elements

In the following, I will explain the main elements of an OpenSCENARIO scenario.

1. **RoadNetwork**
   The **RoadNetwork** is there to describe the whole scenario environment. This means all the roads, lanes, the surrounding environment(i.e. buildings, lakes etc) and traffic elements(i.e. traffic lights, traffic signs etc). Furthermore, each road and lane that is part of the **RoadNetwork**, must have a RoadId respectively LaneID.

2. **Entities**
   This group represents all the actors(i.e. vehicles, pedestrians, etc.) and static objects(i.e. vending machines, benches, barbecues etc.) that are added in the scenario by the user. This means that it’s not predefined in the selected road network.

3. **Actions**
   With the help of an **action**, the user can define how each entity behaves during the scenario. Furthermore, an **Action** can also be entity independent and used to change the simulation environment.

4. **Triggers**
   Triggers are used to trigger different parts of the scenario. Those parts can be either a **Story**, an **Event** or an **Act**, which are also explained later in this section.

Those however are not the only components of OpenSCENARIO. In this section, I explain the different components of a scenario. In OpenSCENARIO, almost the entire description of the scenario is a connected to the **Storyboard**. This means that the only connections that the **Scenario** has, other than the **Storyboard**, are the **Entities** and the **RoadNetwork**.
The **Storyboard** describes the questions “‘who’ is doing ‘what’, and ‘when’ in a scenario” [20].

It consists of two sections – one **Init** and one **Story** [20]. The following Figure 2.2 gives an overview of the components of the **Storyboard** and their components as well.

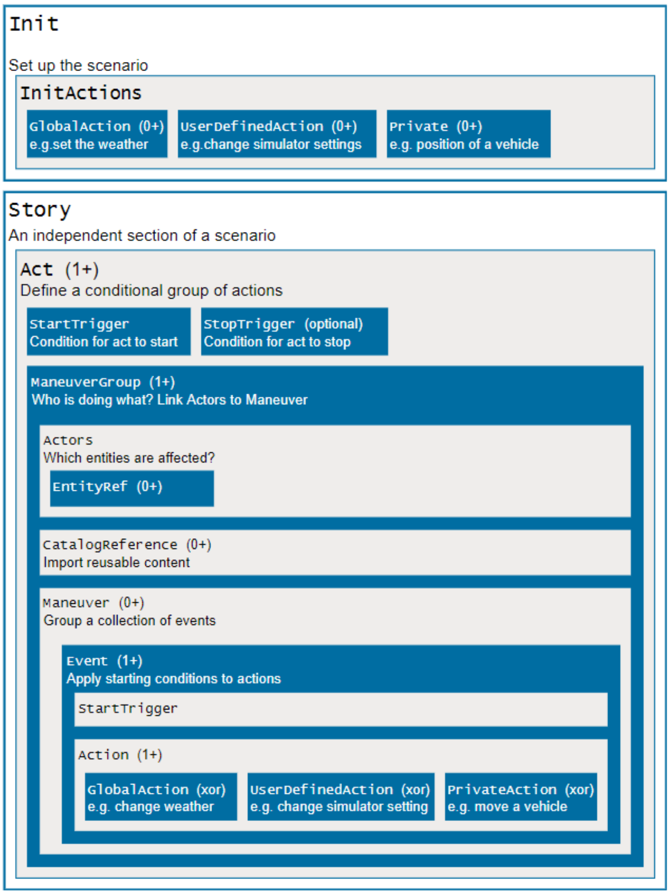


Figure 2.2: ASAM OpenSCENARIO Scenario Elements[20]

The first section is the **Init** of the **Storyboard**. The **Init** is used to define the initial conditions of the **Scenario**. As it can be seen in Figure 2.2, the **Init** can contain different **InitActions**. Those actions can be a **GlobalAction**, **UserDefinedAction** and a **PrivateAction**.
A **GlobalAction** is an “action that does not explicitly target an entity state” [21]. For example, it is used to set the time of the day or the weather. Whereas, a **PrivateAction** is an action, which is connected to an **Entity** and the action is to be executed by the entity. The **PrivateAction**s in the **Init** can be **SpeedAction** or **TeleportAction**. The second part of a **Storyboard** is the **Story**. There is usually only one **Story**, because one can just use more **Act**s to describe the additional **Stories**, but there is the possibility for one **Storyboard** to have more **Stories**. Just like in the narrative fiction, a **Story** is divided in different **Act**s. Each **Act** is supposed to answer the question “When is this happening?” [20] in a **Scenario**. In order to answer this question, each **Act** has a **StartTrigger** and a **StopTrigger**. An **Act** will then respectively start or stop, according to the conditions set in the triggers. For further information, refer to section “Triggers” below.
Another part of an **Act** is the **ManeuverGroup**. Just like **Act**s, multiple **ManeuverGroup**s can be part of one **Act**. Logically, a **ManeuverGroup** consists of different **Maneuver**s.
A **ManeuverGroup** however, also has an **Entity** reference. This **Entity** then performs all the

Maneuvers. Thus, the **ManeuverGroup** answers to the question “Who?”.
At last, a **Storyboard** should also answer to the question “What?”. This task is accomplished by the **Maneuver**s, which on their side are containers for the so-called **Event**s. Each **Event** holds different **Action**s which are executed depending on the **StartTrigger/**s that the **Event** has.

2.4.1.1 Triggers

The triggers in OpenSCENARIO can be either **StartTrigger** or **StopTrigger**. As stated earlier, the triggers are used to trigger different parts of the scenario. Respectively, if activated, the **StartTrigger** starts a specific element of the scenario and the **StopTrigger** terminates the process. Each trigger contains at least one **ConditionGroup**, which on its side contains different **Condition**s. If *all* **Condition**s in one **ConditionGroup** are evaluated to “true”, only then the **ConditionGroup** is also evaluated to “true”.
If at least one **ConditionGroup** of a trigger is evaluated to “true” then the trigger is considered active. One can also refer to the connection between the **ConditionGroup**s as a “logical or”.

2.4.1.1.1 Conditions

A **Condition** is a logical expression which is always evaluated to be either true or false. There are two types of **Condition**s - *ByValueCondition* and *ByEntityCondition*. Basically the *ByValueCondition* is dependent only on a specific value to be triggered and examples for this **Condition** are **SimulationTimeCondition** or **StoryboardElementStateCondition**. The *ByEntityCondition* on its side, is dependent on an **Entity**, for example **TraveledDistanceCondition** or **RelativeDistanceCondition**. The type of **Condition** defines the logical expression in the **Condition** and the further required parameters for the definition of it.
For example, if we choose the **SimulationTimeCondition**, then two further parameters must be given. The **SimulationTimeCondition** is activated by a *value* and a *rule*. The *rule* can have the six mathematics operators for comparison - *lessThan*, *lessEqual*, *equalTo*, *notEqualTo*, *greaterEqual* and *greaterThan*. According to the given *rule* and *value*, a logical expression is built, i.e., *SimulationTime* >= 15 : *rule* :- *greaterEqual*, *value* :- 15.
Furthermore, each **Condition**, regardless of the type, has three base parameters - a name, a delay and a conditionEdge.
The *delay* value is “the amount of time that needs to elapse between meeting the **Condition** and reporting it as met” [20]
The *conditionEdge* makes the **Condition** evaluation more complex and flexible. There are four possible values of *conditionEdge* - rising edge, falling edge, risingOrfalling edge and none. A **Condition** with *rising edge* returns true if its logical expression was evaluated previously to false and at the next moment as true. A *falling edge* is exactly the opposite – the expression is first evaluated to true and then to false. The *risingOrfalling edge* is a combination of both rising and falling edge, thus it returns true if either of the edges is detected. And last but not least, the *none edge* just always returns the value of the logical expression. Those are the base components of a

Condition. The other component, as mentioned earlier, is the **Condition** itself(*ByEnityCondition* or *ByValueCondition*). For further information on the available **Condition**s please refer to the OpenSCENARIO Documentation [5].

2.4.1.2 Environment

In this section, I will explain the elements that are needed in order to change the weather and/or time of the day The **Environment** requires **TimeOfDay, Weather** and **RoadCondition**. In order to define a **TimeOfDay** a *year, month, day, hour, minutes* and *seconds* is required. This attribute is important to be defined correctly, because depending on this, the scenario will also be either during the day or the night. The **RoadCondition** has only one property, and it represents the friction scale factor of the road.

2.4.1.2.1 Weather

Logically, the **Weather** in OpenSCENARIO is used to define the environment weather. The **Weather** has a **CloudState, Sun, Fog** and **Precipitation**. In OpenSCENARIO there are five different **CloudState**s - *cloudy, free, overcast, rainy* and *skyOff*. The next part of the **Weather** is the **Sun** and for an OpenSCENARIO **Sun**, *intensity, elevation* and *azimuth* values are needed. All of those values are float, where the *intensity* is in lux, the *azimuth* is in π, where 0 means North, $\pi \div 2$ East, π means South and $3 \div 2\pi$ means West.

The **Fog** also has additional attributes, which are the **BoundingBox**(see Section 2.4.1.4) and the visual range of the **Fog**. The visual range is an integer value, where anything above 100 means that there is no fog.

2.4.1.3 EnvironmentAction

In order to set the weather and the day /night seeting and in a OpenSCENARIO one must use an **EnvironmentAction**. The **EnvironmentAction** requires an **Environment**.

2.4.1.4 BoundingBox

The **BoundingBox** is important, because it is needed for the defintion of dynamic entities and for a fog. A **BoundingBox** has 6 parameters - *width, length, height, x_center, y_center* and *z_center*, where the centers refer to the geometrical center of the bounding box expressed in coordinates that refer to the coordinate system of the entity for which the bounding box is created.

3 Related Work

In previous years, safety assessment of autonomous driving has become a very discussed and researched topic [23, 54, 49, 34, 19, 44, 45, 17, 70, 43, 28, 47]. Furthermore, the research is also strongly focused on situation assessment and decision-making, rather than scenario creation. In today's world, there are a variety of ontologies that can be used to describe a scenario or at least part of a scenario, i.e., a single scene. In some of the papers, the safety assessment field is discussed in general, especially scenario-based validation. Furthermore, in most of the papers, an ontology is presented that describes a part of a scenario, a scene, and then in some papers a semantic reasoner is applied to the ontology, delivering new knowledge. Some of the ontologies are designed for highways or only intersections, while others have no specific ODD. In this section, I will discuss some of the mentioned work that is directly related to the contributions of this thesis.

3.1 Pure scenario description ontologies

In this section, I will present some ontologies, that are able to describe a scenario or a scene(part of a scenario). It is important to note that none of the mentioned ontologies is able to describe specific corner cases. Bagschik et al. suggest an ontology for a keyword based scenario description [23]. First they create the basics of the scene – lanes, roads or crash barriers. Afterwards, they add the positions of different elements. The elements of a scenario can be positioned according to their relative position to the other traffic participants (e.g., left, right, in_front_of). As the last step, the traffic participants are added with their corresponding maneuvers. The possible maneuvers include follow, approach, turn back, pass, drive up, lane change, turn and safe stop.
Based on this publication, later on, Menzel et al. [49] introduced a method for automatic generation of OpenSCENARIO and OpenDRIVE files. The approach takes a keyword-based scenario as an input, which is available in the previously mentioned Ontology, and then details the input into the so-called Parameter Space Representation. In this context, to detail the input, means to change the structure via a script and adjust it to the requirements of OpenSCENARIO/OpenDRIVE. After the detailing step finishes, they perform different format conversions to generate the corresponding OpenSCENARIO and OpenDRIVE files. The authors generated around 10,000 scenarios that are based on Germany Freeways [49]. The scenarios are available for execution in the simulation environment Virtual TestDrive and can only execute the maneuvers mentioned above. Also, since they used a mass scenario production via combination of maneuvers and 3 vehicles, there is no way to say whether a scenario with critical situation was created. Furthermore, they suggest in future work that also the OpenSCENARIO Detailing of their approach should be modeled explicitly, for example, with the help of an Ontology and not into their source code [49].

Fuchs et al. [34] introduce a scenario ontology for scene representation, with the purpose of describing the scene around the ego vehicle. They don't take the other participants actions into account. With the resulting ontology, one is also able to describe the lanes positions to each other and on which lane every participant is. In their ontology, Fuchs et al. describe the lanes of the traffic participants in the so-called "*trafficObjectIsValidForLane*" class, which, in my opinion, could also be modeled other ways with the help of object properties. Moreover, this ontology has been created with the intention to be developed later by its users and only reflects the main structure of a scenario. Thus, the proposed ontology is not able to describe a scenario detailed enough without any further changes. It is also important to note that the ontology doesn't have the purpose of creating a simulation.

The last ontology I would like to present in this section is the OpenXOntology of ASAM [19], since OpenSCENARIO is part of the OpenX family. The OpenXOntology has the sole purpose of translating the different ASAM projects into one unified language. Thus, it is more or less used as a taxonomy. In its current state, the ontology consists of 347 classes, which are there to describe the different OpenX parts of the project, which are OpenSCENARIO, OpenDRIVE, OpenLABEL, OpenCRG and OpenODD. Furthermore, every class is well described within the ontology, which achieves the previous mentioned goal of the OpenXOntology. However, the ontology seems to be missing the main components of OpenSCENARIO and thus is not able to describe a whole scenario.

3.2 Ontologies in the field of decision-making and scenario description

Hummel et al. [44] contributed to the scene understanding field. However, the resulting ontology cannot describe an entire driving scene to generate a scenario simulation. They introduce an ontology for road intersection **ROad Network oNtologY (RONNY)**. Their ontology is in the condition, with the help of a reasoner, to infer the answer to questions about different traffic rules, like in which direction is a specific lane. Hummel et al. [44] present a series of different SWRL rules integrated within the ontology. **RONNY** was tested in a sample set of 45 different intersections, including also very complex intersections, as stated by Hummel et al. [44]. The purpose of the test was to see whether any of the assertions that were made by the Reasoner of the ontology contradicts with the ground truth of these intersections. Nevertheless, the ontology is not in the condition to describe any corner cases, since it focuses on the description of road intersections and their identification.

In [45] the authors also present a method for the description of complex intersections using an ontology. They focused mainly on the description of the road network, and not so much on the interaction between the entities. This, automatically, leads to the conclusion that corner cases can not be described, because the interaction between the entities is a crucial part of the description.

In contrast to [45] publication, the authors of [17] present an ontology for the description of the road network, but with much bigger focus on the interaction between the entities. Armand et al. [17] furthermore, propose 14 different SWRL Rules to help with the scene assessment. Some rules include, whether the vehicle should stop on the intersection or, for example, whether a vehicle follows another vehicle. The authors even present a described small scene, but as I previously mentioned, the main focus is on the interaction between the entities and for this reason only actions are described. Logically, there is no way to generate a whole scenario or describe any sort of corner cases with the help of this ontology, but the reasoning within the ontology can be used as a good starting point for a decision-making system.

Another ontology for scene description is proposed by Zhao et al. [70]. The authors there present 3 different ontologies, which when taken together are supposed to be able to describe a scene(only a scene, because there is no way to represent a timeline within this ontology). The presented ontologies are the **Car, Map** and **Control** ontology. The **Car** ontology has 3 top classes (direct subclasses of **owl:Thing**) - *CarParts, Trajectory* and *Vehicle*. The *CarParts* consist of the different available sensors and also an engine. Furthermore, the subclasses of *Vehicle* allows the description of different vehicles and the *Trajectory* is used to define a path or different speed characteristics.

The **Map** ontology is used for the modeling of the road network and is specifically constructed for urban areas of Japan. There is the possibility to model different objects like buildings, junctions, traffic signs or lights and different road segments. At last, the authors present the **Control** ontology, which has the purpose to express different driving actions or the path of a vehicle. They introduce 9 driving actions and differentiate between an *IntersectionSegment* and a *LaneSegment* within the *Path* class of the ontology. The authors also constructed a partial map of Japan with the proposed map and control ontologies, which shows that the ontologies are capable of describing a scene. Their ontology, furthermore, contributes to the decision-making problem of autonomous vehicles. As stated by Zhao et al. [70], they developed a decision-making system which is able to make different driving decisions such as “ToLeft”, “Give Way” or “Stop”. However, just like the ontology presented by Fuchs et al. [34] this ontology does not have the purpose of creating a simulation and therefore is not able to do so, especially because there is no way to model entities other than vehicles.

A similar decision-making ontology was designed by Huang et al. [43]. The proposed ontology has 5 main classes on the top level (subclasses of **owl:Thing**): *EgoVehicle, Behavior, Obstacle, RoadNetwork* and *DrivingScenario* [43]. Logically, the *EgoVehicle* class represents the autonomous vehicle in the scenario and it doesn’t have any subclasses. With their *Behavior* class, the authors model the different actions of the ego vehicle, which in this case is divided into lateral (e.g., lane changing) and longitudinal (e.g., acceleration) behavior. The expression of the behavior in this ontology happens with the help of the two object properties *exeLatiDecision, exeLongDecision*, which are connected to the ego vehicle. Within the *Obstacle* class they differentiate between dynamic and static objects, where the dynamic objects can be animals, pedestrians or different vehicles, and the static objects are divided into natural objects and lane building, which contains construction items, i.e. traffic cone. The division of the static objects in this ontology is very incomplete, because a lot of static objects are missing, i.e. human-made static object, other than construction items. Furthermore, it is not clear, how exactly are those objects used. The next part of the ontology of Huang et al. [43] is the *RoadNetwork*. It consists of *RoadType* and *RoadPart*, with which different parts of a road network can be described. The goal is to be able to describe, both highways and urban roads. The last main part of their ontology is the *DrivingScenario*, which classifies the scenarios into 3 types of scenarios - the *InSpecialAreaScenario*, the *OnRoadScenario* and the *NearSpecialAreaScenario*. With the help of those classes, a scenario can be classified according to its position (e.g., a scenario in a tunnel or in a bridge). As mentioned earlier, the authors also present a decision-making system. The system uses Prolog [8](a programming language, based on predicate logic) to infer different logical conclusions, such as whether a lane change will change the outcome of the scenario. If it changes the outcome, then the lane change action is forbidden [43]. The proposed ontology [43], is somewhat in the condition, to describe different corner cases, because scenarios under bridges or in tunnels, can be considered as corner cases of in the Domain Level (See 4.1), but it is not in the condition to cover all the different categories.

Next, I will introduce the ontology introduced by Chen and Kloul [28]. In their ontology, Chen and Kloul present a way to model a scenario specifically for a highway. The authors present 3 ontologies – the **Highway Ontology**, the **Weather Ontology** and the **Vehicle Ontology**. Their **Highway ontology** was built based on French official documents and includes four main concepts - *RoadPart, Roadway, Zone* and *Equipment*. Those are also their top classes of the ontology (the direct subclasses of **owl:Thing** (See Section 2.2) [28]. With the *RoadPart* of the ontology, Chen and Kloul describe the long profile of a highway and with the *Roadway*, they describe the longitudinal profile of the highway [28]. With their **Weather ontology,** the authors are able to model different weather conditions at a particular place and time. As for the **Vehicle ontology,** they model a vehicle with 9 parameters, some of which are height, width, length, color, speed etc. The ontology, furthermore, consists of 8 different actions and also various lights, which can be used by the vehicle (fog light, parking light, etc) [28]. The authors further present the logic implemented within the ontology, with the help of first-order logic. Rules which, for example, can infer whether the road has a priority sign or not. The presented by Chen and Kloul ontology is, without doubt, a good example of a scene description. However, even if a scenario can be described, due to the limited actions and interactions between entities, there is no way to model a corner case scenario. Furthermore, as mentioned earlier, the presented ontology is created specifically for highways, which also drastically reduces the amount of scenarios, which can be described.

In her master thesis, Koch [47], compares both of the previous presented ontologies [70, 43] in aspects such as completeness, conciseness, minimality, clarity, acyclicity and prohibition of property's domain and range multiplicity [47]. As stated by Nina Koch, both of the ontologies have their weaknesses and for this reason she proposes a new consolidated ontology, which as also stated by Koch, improves the understanding of the concepts of a scenario within the research community. The integrated ontology, combines the **Control** and **Map** ontology introduced by Zhao et al. [70], parts of the *Obstacle* class of Huang et al. ontology [43] and introduces a new **Context** ontology, which takes selected parts of both of the ontologies. Furthermore, each ontology is further extended with classes and properties. According to her, the new ontology is much more complete and was able to fix some weaknesses, which both of the ontologies had, but it's still incomplete and limited. For example, the *DrivingScenario* is still limited only to bridges and tunnels. The extension of this class, of course, was also out of the scope of her master thesis, and thus the resulted ontology is still not in the condition to describe various corner case scenarios.

3.3 Conclusion

To the best of my knowledge, in the current state of the art, there aren't any ontologies, which can describe different corner cases for autonomous driving. Furthermore, this thesis contributes to the classification of corner case scenarios, to which also none of the existing ontologies contributes. In addition, even though some of the presented ontologies are in the condition to generate simulations (the files for it), none of them is in the condition to generate scenarios which contain all of the corner cases categorizations.

4 Approach

As I pointed out in Chapter 3, in the research world, a way to describe and generate corner cases is missing. For this reason, I propose the following approach(See Figure 4.1) to fill the identified research gap.

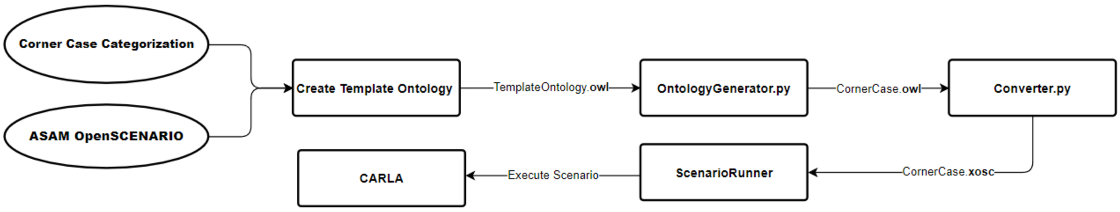


Figure 4.1: Flow diagram of the proposed ontology-based technique for description and generation of corner case scenarios.

As Figure 4.1 shows, the first step was to create the template ontology(**TemplateOntology.owl**). This is the ontology, which is used as base for every scenario, it is so to say the skeleton of the whole process. The template ontology is closely aligned to the OpenSCENARIO documentation [20], since the ontology is used as a backbone for the scenario simulations in CARLA. Furthermore, part of the ontology is based on the categories introduced in [14, 26, 41]. In the next step of Figure 4.1, the template ontology(**TemplateOntology.owl**), which is used as baseline for all scenarios, is read by the **OntologyGenerator** script. In this step, according to the chosen scenario description in the script, a new ontology (the **CornerCase.owl**), which contains the correct individuals and their property assertions needed for the description of the scenario within the ontology, is generated. Afterwards, the **CornerCase.owl** is passed to the **Onto2OpenSCENARIOConverter.py**. As it can be seen in Figure 4.1 the Onto2OpenSCENARIOConverter Script reads the given ontology(**CornerCase.owl**) and generates the final OpenSCENARIO(**.xosc** [20]) file, which in this case is called **CornerCase.xosc**. The **CornerCase.xosc** is then executed directly in the Scenario Runner [10], which executes the scenario into CARLA. In the next sections, the single elements of the flow are explained in detail, together with the chosen corner case examples.

4.1 Exploration Phase

In this section, the process of selecting the appropriate examples for each corner case category (introduced in Section 2.1) is presented, together with the corresponding examples. The selection of the corner cases was conducted in two steps. At first, different scenarios were selected from three sources. The first source is the presented examples in the paper presented by Breitenstein et

al. [14]. In addition, various video sources, such as dash cams and third-person videos of traffic situations. Furthermore, a brainstorming session was conducted, where personal experience and combination of the previous mentioned sources was included.
Afterwards, the examples that are most suitable and realizable for each corner case category were selected.

4.1.1 Categories and examples

In the following, I will present each example and the reason for the selection of exactly this case for the corresponding category. For the exact definition and hierarchy, please refer to Section 2.1. The resulted scenarios can be seen in Section 5.

4.1.1.1 Pixel Level

The examples which were chosen for this level, both for the **Local Outlier** and the **Global outlier**, are the so-called dead pixels. The decision for this example, was based on the proposed by Breitenstein et al. [14] example for **Dead-Pixels**. This category is independent of the scenario and is applicable to every of the following categories.

4.1.1.2 Domain Level

As mentioned in Section 2.1, **Domain shift** is “a large, constant shift in appearance, but not in semantics.”[14]. This means, in simpler words, that the environment changes. For this reason, the chosen in example for this category is the scenario, where “**The ego vehicle, suddenly drives into a dense fog**”.

4.1.1.3 Object Level

As mentioned earlier, a **Single-Point Anomaly** is characterized with an unknown object in the scene. Since “unknown” object is subjective, because it is based on the data that was trained on, I chose the type of object based on the popular Cityscapes dataset [4]. As an example in this category, a scenario with “**a suddenly appearing unknown object(i.e. a vending machine)**” will be implemented.

4.1.1.4 Scene Level

The Scene level is separated into two subcategories – Contextual Anomaly and Collective Anomaly.
A **Contextual anomaly** is “when a known object appears, but in an unusual location” [14]. Just like in the previous example, it is hard to guess what is a known object to the autonomous vehicle, and for this reason I have taken an object, which first, is part of the cityscape dataset, and second, it is logically part of the training — a street sign. The selected for this category example is a scenario, where “**the vehicle is driving and suddenly there are a lot of signs in the middle of the street, and they are falling**”. The scenario is inspired by a video taken during a heavy storm.
A **Collective anomaly** would be when in the scenario there are “multiple known objects, but in an

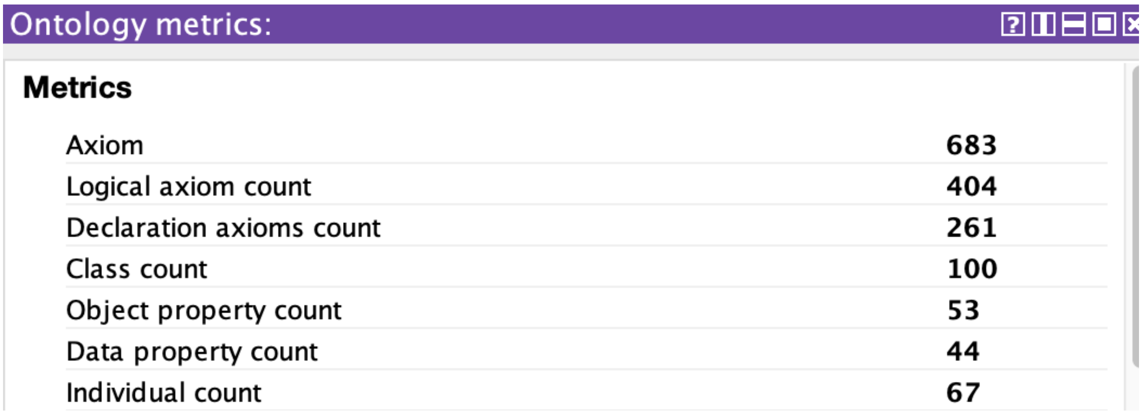
unseen quantity" [14]. The following scenario is a product of the brainstorming session I considered some data samples, which include "**A lot of pedestrians/cyclist running/cycling driving in front of the ego vehicle**".

4.1.1.5 Scenario Level

As mentioned earlier, the Scenario Level consists of three subcategories – Anomalous Scenario, Novel Scenario, Risky Scenario. Since all of the categories in this level, are closely connected to the training process, all of the chosen scenarios are based on assumptions. For the **Anomalous Scenario** category, the chosen scenario is "**A person is suddenly running into the ego car**". The scenario is inspired by insurance frauds [27].
For the **Novel Scenario** category, I chose the scenario, where "**A cyclist is driving strangely on the opposite lane of the ego car**". This scenario was chosen after the brainstorming session, where different cases were taken into account - first, the scenario shouldn't happen in the lane of the ego vehicle, and secondly it should be an unobserved behavior, thus the strange driving.
Lastly, for the **Risky Scenario** simply a scenario where, a "**a vehicle is cutting in front of the ego vehicle**" was chosen, because this is an event that occurs very often in a normal driving day(as also shown in a lot of dash cam sources), but is still very risky, because the other driven can often underestimate the remaining space in the lane of the ego vehicle.

4.2 Template Ontology

The idea of the template ontology, is to provide the classes, properties and individuals needed for the description of every scenario, and it is to be used to create new ontologies which only have further new individuals. The following Figure 4.2 shows the metrics of the template ontology.



Metrics

Axiom	683
Logical axiom count	404
Declaration axioms count	261
Class count	100
Object property count	53
Data property count	44
Individual count	67

Figure 4.2: Metrics of the Ontology

The resulted ontology consists of 100 Classes, 53 Object Properties, 44 Data Properties, 683 Axioms and 67 Individuals. A graph of the whole ontology can be found in the Appendix, Figure D.1.

4.2.1 Classes

At first, I would like to introduce the top classes of the ontology. The following Figure 4.3 shows an overview of those classes.

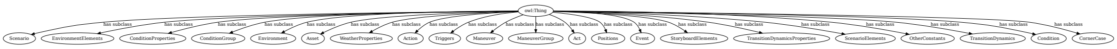


Figure 4.3: Top Level classes of the ontology, subclasses of **owl:Thing**

The classes of the ontology are derived from the OpenSCENARIO documentation. Those, for example, are **Scenario, Storyboard, Init, Act, Event, ManeuverGroup, Maneuver, Event, Action** etc. They are used to describe the main structure of the scenario. In addition, there is the class **Positions**, which holds as subclasses, the different availalbe positions - **Position, RelativeObjectPosition** and **Orientation**. Furthermore, there are some additional helper classes added in the ontology to improve readability - **OtherConstants, WeatherProperties, ConditionProperties, EnvironmentElements, ScenarioElements, StoryboardElements, TransitionDynamicsProperties**. The class names of this group correspond to the main classes in the ontology. For example, **WeatherProperties** corresponds to **Weather** class in the ontology. Furthermore, the subclasses of **WeatherProperties** are the ranges(See Section 2.2.1) of the different object properties that are needed for initialization of a Weather OpenSCENARIO object (see Section 2.4.1.2.1). So the subclasses of the **WeatherProperties** class are **Precipitation, Fog, CloudState, Sun**. All help classes are constructed to have more than one subclass.

The rest of the mentioned helper classes, with the exception of **OtherConstants**, are also constructed with this approach. The **OtherConstants** class contains all the properties of ontology elements, which were supposed to have only one subclass. For example, it has a subclass, the class **PrecipitationType**, which is the only property of **Precipitation**. Furthermore, properties which are properties of two different classes are also part of **OtherConstants**, i.e., **BoundingBox**, which is a property of **Fog** and the different **Entities**. The rest of the classes are part of the OpenSCENARIO Documentation and are thus needed for the description of the scenario.

4.2.2 Object Properties

With the current design, the resulted ontology consists of 50 **Object Properties** and 44 **Data Properties**. Seven of the existing object properties are help properties to improve readability. Similar help properties can be also found in the data properties of the ontology.

The following table (See Table 4.1) gives an overview of the available object properties in the ontology.

The Main OpenSCENARIO properties, are used for the description of the main objects in OpenSCENARIO, i.e. **Storyboard, Init, Story, Act, ManeuverGroup, Maneuver** and **Event**. For example, an individual of class **Scenario** would have the property assertion *has_storyboard*, connected to an individual of a class **Storyboard**. Furthermore, the **Storyboard** individual would have the property assertions *has_init* and *has_story* with the corresponding individuals of the cor-

Object Properties		
Main OpenSCENARIO Properties	**Help Properties**	**Multidomain properties**
has_storyboard	*condition_properties*	*has_start_trigger*
has_init	*event_properties*	*has_stop_trigger*
has_story	*relative_distance_condition*	*has_bounding_box*
has_act	*main_part_properties*	*has_entity_ref*
has_maneuver_group	*storyboard_element_state_condition*	*has_transition_dynamics*
has_maneuver	*transition_dynamics_properties*	*has_position*
has_event	*weather_properties*	*has_orientation*
	environment_properties	

Table 4.1: Object properties within the ontology

responding class. This then goes on for the rest of the properties from this group. The multidomain properties, are properties that have more than one domain, which for example in the case of *has_start_trigger* are the **Act** and the **Event**, since both of the mentioned elements, requeire a **StartTrigger**.
In the next table (see Table 4.2) I will present the subproperties of each element from the help properties in Table 4.1.

Help Properties		
condition_properties	**scenario_properties**	**relative_distance_condition**
has_condition	*has_storyboard*	*has_relative_distance_type*
has_condition_edge	*has_entity*	*has_target_reference*
has_condition_group	*has_town*	**storyboard_element_state_condition**
has_condition_rule	*has_road_network*	*has_state*
transition_dynamics_properties	**environment_properties**	*has_storyboard_element_ref*
has_dynamics_dimension	*has_road_condition*	**event_properties**
has_dynamics_shape	*has_time_of_day*	*has_priority*
has_transition_dynamics	*has_weather*	has_action

Table 4.2: Help object properties within the ontology

4.2.3 Data Properties

In this subsection, I will present the data properties within the ontology. As a reminder, data properties are relations which on the one end have an ontology entity and on the other end have a data type(i.e. integer, float, string, etc.). For further information, see Section 2.2.1).

As it can be seen in Table 4.3, just like in the object property tables (See table 4.1) the data properties are also grouped. The properties in **bold** are the helper data properties, which are used for better an overview. In the ontology also each Range and Domain of the properties can be seen.

Data Properties		
bounding_box_properties	**time_of_day_properties**	**sun_properties**
has_z_center	*has_animation*	*has_azimuth*
has_y_center	*has_day*	*has_elevation*
has_x_center	*has_hour*	*has_intensity*
has_width	*has_minute*	**other_properties**
has_length	*has_month*	*has_value*
has_heigth	*has_second*	*has_target_lane_id*
condition_properties	*has_year*	*has_maximumExecutionCount*
has_distance_value	**orientation_properties**	*has_target_speed*
has_traveled_distance_value	*has_roll*	*has_visual_range*
has_simulation_time_condition	*has_pitch*	*has_friction_scale_factor*
has_condition_speed_value	*has_heading*	
has_delay		

Table 4.3: Data properties within the ontology

4.2.4 Constants

The different constant values from the OpenSCENARIO Documentation, that are needed in the various scenarios, are represented in the ontology with the help of individuals. For example, the different condition rules, mentioned in Section 2.4.1.3, are individuals within the ontology. The following table 4.4 presents the constant individuals available in the ontology. Table 4.4 gives an overview of the available within the ontology constants and helps with the creation of a scenario. For further information on the constants, a description can be found in the ontology by the corresponding class.

Constants			
ConditionEdge	**PrecipitationType**	**CloudState**	**State**
falling	*dry*	*cloudy*	*completeState*
none	*rain*	*free*	*endTransition*
rising	*snow*	*overcast*	*runningState*
ConditionRule	**Priority**	*rainy*	*standbyState*
equalTo	*overwrite*	*skyOff*	*startTransition*
greaterOrEqual	*parallel*	**DynamicsShapes**	*stopTransition*
greaterThan	*skip*	*cubic*	***RelativeDistanceType***
lessEqual	***DynamicsDimension***	*linear*	*cartesianDistance*
lessThan	*distance*	*sinusoidal*	*euclideanDistance*
notEqualTo	*rate*	*step*	*lateral*
	time		*longitudinal*

Table 4.4: Constant Individuals

4.2.5 Default Individuals

The following Section gives an overview of the available default individuals within the ontology, which are created in order to make the scenario generation easier.

Default Individuals		
Towns	**Start/StopTrigger**	**Environment**
Town01	*def_start_trigger_st_0*	*def_env_action*
Town02	*def_stop_trigger_st_30*	*def_environment*
Town03	*def_st_condition_0*	*def_fog (no fog)*
Town04	*def_cond_group_0*	*def_fog_bounding_box*
Other	*def_st_condition_30*	*def_precipitation_dry*
def_ego_vehicle	*def_cond_group_30*	*def_sun*
def_ego_bounding_box	*def_empty_stop_trigger*	*def_time_of_day (Daylight)*
def_ego_speed_10		*def_weather_sunny*
def_transition_dynamics		*def_road_condition*

Table 4.5: Default Individuals

As shown in Table 4.5, there are different default individuals available, divided into groups. It is important to note, that CARLA specific asset or town individuals must have the name corresponding to the CARLA documentation (e.g. *vehicle.tesla.model3*). Furthermore, every default individual begins with "def_", in order to avoid duplicate names during the creation of new scenarios. First are the towns of CARLA. Those towns can be extended by just adding the name of the town into the ontology and giving it the class "Town". The next individual category is the Ego Vehicle. It is important to note, that logically, only one Ego vehicle can exist within one scenario. Furthermore, part of the Ego Vehicle category are the individuals needed for the creation of the *def_ego_vehicle* and a *SpeedAction* for it. Furthermore, there is an *EnvironmentAction* available, the *def_env_action*, which when added to the *InitAction*s sets the environment to a light night. This individual was created, because for some reason the scenario runner sets the time to afternoon and the scenario plays after sunset by default and the scene is very dark. Even though all of the scenes are during the night, everything is clearly visible. This can, of course, be changed by creating another environment action. The last group of default individuals are the *StartTrigger*s and the *StopTrigger*s. There is one *StartTrigger*, with the necessary assertions, which triggers, when the simulation time is greater or equal to 0 and two *StopTrigger*s. The *def_stop_trigger_st_30* triggers, when the simulation time is greater or equal to 30. The reason for the number 30 is because, most of the scenarios which I created are executed in less than 30 seconds. Furthermore, there is a *def_empty_stop_trigger* individual, which is just an empty trigger, that can be used for the *Story*. If the *Story* has an empty *StopTrigger*, then it stops when the last *Act* finishes.

4.3 Corner Case ontology

The Corner Case Ontology, which is part of the template ontology, was built on the previously mentioned corner case categories. There are classes for each Corner Case and its corresponding category and level, built accordingly to the categorization proposed by Heidecker et al. [41]. Furthermore, each of the bottom ontology level corner case classes (this means there aren't further subclasses) also has a constant individual, which the Reasoner can use for Corner Case Classification. This is necessary, since the classification, can only be possible with the help of SWRL rules, and as mentioned earlier(See Section 2.2.5), those rules only work with individuals. The following Figure 4.4 shows an overview of the description of the corner cases within the ontology. The reason that, the individuals of the different categories are needed is that, as mentioned earlier (See Section2.2.5), the SWRL Rules work only with individuals. Thus, in order to infer that a particular scenario belongs to a particular category, an individual from that category is required.

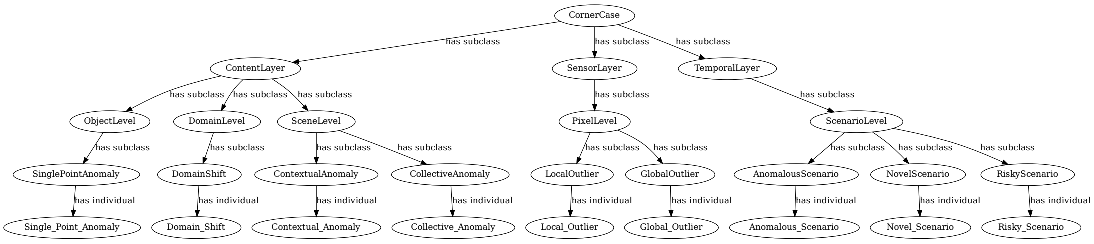


Figure 4.4: Corner case categories description within the ontology

4.3.1 SWRL Rules

With the help of SWRL Rules and the reasoner, we were able to differentiate for a scenario whether it has a high collision potential scenario or not. A high collision potential scenario can be characterized in many ways. The rule that I implemented is based on that whether an entity is initially spawned in the *EgoVehicle* lane or not. Furthermore, it's possible to identify some of the corner cases categories. For example, the **SinglePointAnomaly** can be inferred. The rule checks whether the scenario has an unknown object or not. If this is the case, then the scenario has the corner case category **SinglePointAnomaly**.

4.3.2 Requirements for a correctly defined scenario

The following presents the basic requirements of a scenario in the ontology for it to be correctly defined. In order to create a **Scenario** according to the OpenSCENARIO Documentation [20], a Storyboard as well as all participating **Entities** are needed.
The **Storyboard** needs an **Init**, a **Story** and a **StopTrigger** to be initialized.
The **Init** on itself requires all initialization **Action**s(**TeleportAction** is a **must**) of the already defined **Entities** and optional **GlobalAction**s.
The **Story** needs an **Act** and an optional **StopTrigger**.
Each **Act** requires minimum one ManeuverGroup, as well as **StartTrigger** and **StopTrigger**.
In order to create a **ManeuverGroup** one needs to define **Maneuver**s with defined **Event/**s inside them.
The **Event**s can be only initialized with an already existing **Action** and a **StartTrigger**.
Every **StopTrigger** or **StartTrigger**, if not empty, requires a **ConditionGroup** which should have at least one **Condition** inside it.

4.4 OntologyGenerator

Since the resulted template ontology is complex, there are a lot of new individuals to be created for each scenario. The creation of individuals, however, requires a lot of manual work inside the Protégé software. Furthermore, the manual and exhausting initialization of the different individuals can lead to incorrectly asserted or forgotten object/data properties and classes.
Therefore, I designed an OntologyGenerator, which automatically creates individuals in the ontology with predefined class and property assertions. The OntologyGenerator is a python library that reads the main template ontology (introduced in Section 4.2) and then uses it as a template to create a new ontology, filled with individuals. The only new parts of the ontology, when the script is finished, are new individuals. No new classes or properties are created during the execution.

4.4.1 Approach

Since the creation of most elements of a scenario requires that other elements have already been created, the entire process of creating a scenario is bottom-up. This means, that the **Scenario** individual is created last and not first, as one would expect. With the help of the following flow chart 4.5, I will explain the process of creating new individuals(See Section 2.2.2) in the ontology with the correct property assertions.
In order to make the explanation of the process easier, the following Table 4.6 was created. The table contains the available methods within the OntologyGenerator (See Table 4.6).

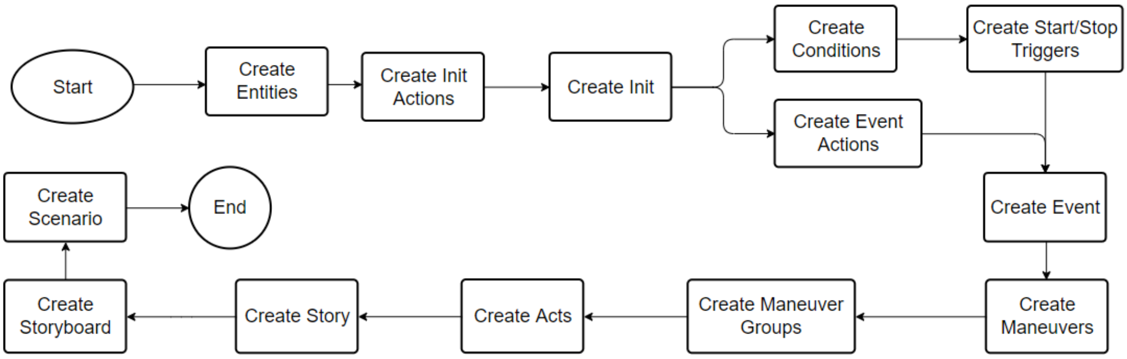


Figure 4.5: Flow Chart of a scenario generation within the ontology

As Figure 4.5 shows, the first thing that needs to be created are the *Entities*. This is necessary, because in order to create the *InitAction*s, an entity reference is required. The creation of the entities is possible with the help of the " *newPedestrian(), newVehicle(), newBicycle(), newEgoVehicle()*" methods and for static entities - "*newMisc()*" (See also Table 4.6). Since one of the goals of this thesis is to enable the usage of an agent within the scenarios, a default *EgoVehicle* individual is already available within the ontology (See Section 4.2.5) Each entity further requires a *BoundingBox* to be created, for which a default *BoundingBox* individual is available, or a new box can be created with the help of the "newBoundingBox()" method. Furthermore, each entity should have a *TeleportAction*, which specifies the initial spawn location, so that the entity can be

OntologyGenerator methods		
Actions	**Assets**	**Main Elements**
newEnvironmentAction	*getBicycleAssets*	*newScenario*
newSpeedAction	*getCarAssets*	*newStoryboard*
newTeleportActionWithPosition	*getPedestrianAssets*	*newInit*
newTeleportActionWithRPAO	*getMiscAssets*	*newStory*
newRelativeLaneChangeAction	*newAsset*	*newAct*
Environment and Weather		*newManeuverGroup*
newEnvironment	*newFog*	*newManeuver*
newTimeOfDay	*newPrecipitation*	*newEvent*
newWeather	*newSun*	*newAction*
Entities	**Other**	**Conditions**
newEgoVehicle	*changeWeather*	*newSimulationCondition*
newCar	*newStopTrigger*	*newRelativeDistanceCondition*
newPedestrian	*newStartTrigger*	*newStoryboardESCondition*
newMisc	*newRoadCondition*	*newTraveledDistanceCondition*
	newTransitionDynamics	

Table 4.6: OntologyGenerator methods, where RPAO stands for RelativePositionAndOrientation, and ES stays for ElementState

spawned later, otherwise the simulator wouldn't know where to spawn it.

There are different possibilities available for the creation of a *TeleportAction* 4.4.2.1.1. Furthermore, various actions, e.g., *SpeedAction, ControllerAction* or different *GlobalAction*s (See Section 2.4.1.3), can be added at this point. Next, the *Init* can be created. With the newInit() method (See Table 4.6), one is able to create an *Init* individual and add every created action until now to it. As mentioned before (See 4.3.2), *StartTrigger, StopTrigger* and an *Act* are required for a *Story* creation. For the triggers, at least one *Condition* is required, so every *Condition* and its corresponding *ConditionGroup* can be initialized at this point. For further information on *Trigger*s refer to Section 2.4.1.1. Also see Section 4.2.5 for the available default *Trigger*s. Parallel to this, the *Action*s for the different *Story Event*s can be created. As shown in Figure 4.5 the next step is to create the *Event*s of the *scenario*. With the defined *StartTrigger*s and *Action*s, an *Event* can be initialized for each *Action*. This happens with the *newEvent()* method. Afterwards, the *Maneuver*s and the *ManeuverGroup*s are created. It is important to remind the reader that every *ManeuverGroup* has an entity reference, and this entity will execute every *Maneuver* within the *ManeuverGroup*(See Section 2.4.1).

With this said, the *Maneuver*s can be created and grouped accordingly. This can be done with the *newManeuverGroup() and newManeuver()* methods. The next step, as Figure 4.5 shows, is to create the *Act*s. When all desired *ManeuverGroup*s are available, together with the corresponding *Start/StopTrigger*s an *Act* can be created with the help of the *newAct()* method. Once all Acts are defined, they are used for the initialization of the Story. Together with the *Act*s, a **StopTrigger** (See Section 2.4.1.1) must be passed to the *newStory()* method and a *Story* is created. Here, one can also use the available default *StopTrigger*s (See Section 4.2.5).

Having those components created, one can now create the *Storyboard* using the *Init* and *Story*.

The last step is then to create the *Scenario*, which requires the *Storyboard* and the *Entities* that were created at the beginning, together with a *Town* (See Section 4.2.5). In the source code, multiple examples are available for better understanding. Furthermore, an example with the different connections between the entities can be seen in Chapter 5.

4.4.2 Implementation

The resulted script consists of multiple methods, as can be seen in Table 4.6. Each of them is designed for a specific class of the ontology. Every function has as input parameters the so-called n1 and n2 parameters, which have the purpose to give each new ontology individual a unique name. This could have been also achieved with only one parameter, but with the second there, it allows more name combinations for the user.
If a method is called twice with the same n1 and n2 parameters, then only one new individual with doubled property assertions would be created. This can lead to unexpected behavior.
As an example, for a SpeedAction individual with n1 = "Ego" and n2 = "02", the resulting name will be "*Ego02SpeedAction*".

4.4.2.1 Functions

As one can see in Table 4.6, the OntologyGenerator has functions, to allow the easier initialization of some elements, those are the so-called "new" functions and every one of them creates a new object in the ontology.

4.4.2.1.1 TeleportAction

There are several ways to create a TeleportAction. As stated earlier, each action needs an entity reference, so that one knows to which entity corresponds the action. The first way is to use the newTeleportAction() function, which requires a predefined *Position* individual in the Ontology and gets it passed as an argument. The *Position* can be created with help of the *newPosition()* function. Another option would be the newTeleportActionWithPosition() function, which requires a road ID, lane ID, s and an offset, where s is the exact position on the lane. The function then creates both Position and TeleportAction Individuals in the ontology with correct property assertions.

As last option there is the newTeleportActionWithRPAO() which requires a relative object reference, x, y, z, pitch, roll and heading, where x, y and z are the differences between the main object and the relative object. The function then creates a new *TeleportAction*, new *Position* and a new *Orientation*. The new *Position* is then associated with the *TeleportAction*, and the *Orientation* gets connected with the Position via an object property (*has_orientation*, see also Table 4.1).

4.4.2.1.2 New Entity

The new entity functions in the current implementation are newEgoVehicle, newVehicle, newPedestrian, newBicycle and newMisc (See also Table 4.6). Those functions take as a parameter the so-called asset_name. This property is supposed to contain the real CARLA Asset name. However, because of the dot symbols("**.**") in CARLA Asset names, the python compiler gets confused, and it can lead to an unexpected behavior. If the name of the Asset is directly written during the generation(e.g. vehicle.tesla.model3) and not inside quotation marks, the compiler interprets the dot as a dot notation. For this reason, the asset_name parameter is intended to be a string which later on gets created in the ontology or passed further on as variable, where the problems with the dot notation can not appear. Furthermore, with this implementation, assets, which are not defined within the ontology can be used.

4.5 Onto2OpenSCENARIOConverter

Since this thesis contributes not only to the description of corner cases, but also their simulation, a tool to convert the description of the scenario in the ontology to a simulation was needed. For this reason, I created the Onto2OpenSCENARIOConverter . The Onto2OpenSCENARIOConverter is a python script, that takes as an input an ontology and returns as an output an OpenSCENARIO XML file (**xosc**). The given ontology, must consists of individuals, based on the OpenSCENARIO scheme and must be either created by the OntologyGenerator(see Chapter 4.4) or manually by following the instructions mentioned in Section 4.3.2. The Onto2OpenSCENARIOConverter then reads the ontology and, according to the class and property assertions of the individuals within the ontology, generates an OpenSCENARIO XML File.

The current implementation was possible because of the PYOSCX scenariogeneration package [9]. With the help of the library, one can generate a XOSC file format. Furthermore, the PYOSCX library has more or less the structure of OpenSCENARIO (like the created in this thesis template ontology), which leads to the repetition of names in the following Section 4.5.1. To summarize, the basics steps of the Onto2OpenSCENARIOConverter are:

1. Read the ontology
2. Create **PYOSCX** objects accordingly
3. Export the **PYOSCX** objects to an **XML** file

4.5.1 Implementation

Onto2OpenSCENARIOConverter functions		
Check functions		**Other functions**
checkDynamicsDimension	*checkEnvironment*	*getNameFromIri*
checkDynamicsShapes	*checkEnvironmentAction*	*addEntitiesToScenario*
checkTransitionDynamics	*checkPrecipitation*	*addActionsToInit*
checkBoundingBox	*checkSun*	*addTrigger*
checkCloudState	*checkFog*	
checkConditionRule	*checkTimeOfTheDay*	

Table 4.7: Functions implemented within the Onto2OpenSCENARIOConverter

Before I explain how does the main process of the Onto2OpenSCENARIOConverter, it is important to note that there are two types of functions in the Onto2OpenSCENARIOConverter source code (See Table 4.7). As shown in Table 4.7, there are the check functions, which are basically used to check the individual's class/name and its properties, and then create the corresponding PYOSCX Object. The checked element is then corresponding to the name, that comes after "check" in the function's name. For example, "checkBoundingBox()" gets as input an ontology *BoundingBox* individual and returns an **PYOSCX BoundingBox** object with the corresponding in the ontology values. The main process of converting an ontology into a scenario XOSC file format is divided into the Init initialization and the Story initialization. In parallel, the process has the initialization of the scenario and entities, which are also explained in the initialization of Init and Story. It is also important to note, that the **only** objects that get created during this process are the **PYOSCX** objects, with one exception. The exception comes, when a *Scenario* description consists of two combined Scenario descriptions (See Section. 4.5.1.3). Everything related to the ontology should have been created in the previous step of the main approach (see Figure 4.1) and in this step the ontology is **read-only**. Furthermore, from now on, everything in **bold** is a **PYOSCX** object and everything in *italic* is an ontology element.

4.5.1.1 Init Initialization

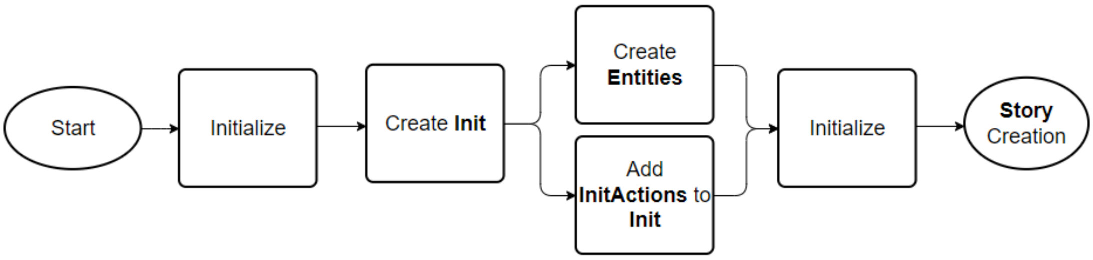


Figure 4.6: Flow Chart of the Init construction.

In the Initialize step shown in the flow chart diagram 4.6, the main variables of the **Scenario** are initialized. First, the **Scenario** name is assigned, by calling the instances of the *Scenario* class in the main ontology. With it available, the *Storyboard*, the *Story* and the *Act* list can be also read and put into variables.

The second step is to create the **Init**. With the initialization of the **Init**, we can also add the **InitAction**s to it. At this point every *InitAction* in the array "init_actions", returned by the "*has_init_action*" object property, is added to the **PYOSXC Init** object. This happens with the help of the "addActionstoInit()" function(See Table 4.7). The method "addActionsToInit()" reads every *Action*, which is passed as a parameter of the type array(via the *has_init_action* object property), and checks its class. Depending on the class of the *Action* (e.g *SpeedAction, TeleportAction* etc), various other property assertions are checked if they are true or false. Afterwards, the **Action**s are created and added to the **Init** with the help of "add_init_action()", a **PYOSCX** function.

At last, the function returns the **Init**, and the next step of the flow chart can begin (See Figure 4.6) - create the **Entities** of the **Scenario**.

In this step, the entities are added to an **Entities PYOSCX** object. To do this, each "*has_entity*" object property assertion that the *Scenario* individual has, is checked. According to the class of the individual, each gets a special property assigned - **ego_vehicle** or **simulation**. If the individual has the class *EgoVehicle* then the property assigned is **ego_vehicle**, otherwise it is **simulation**. This property is necessary, because otherwise the ScenarioRunner [10] throws an error. Also, a **BoundingBox** is needed for the initialization of each entity. The **BoundingBox** can be either default or newly created in the ontology.

If the *Entity* is not some kind of *Vehicle*, then the whole step can be finished for the given entity and a **PYOSCX Entity** object is created. If, however, the *Entity* is of class *Vehicle*, then Axles are required, which can also be either default or asserted in the Ontology. Afterwards, the step comes to its end for the given *Entity* and an **PYOSCX** instance is created. The newly created **PYOSCX Entity** object is added to the **PYOSCX Entities**.

As shown in Figure 4.6, the next step is to create the Storyboard. The **PYOSCX Storyboard** requires an **Init** and a **StopTrigger** for its initialization. For this reason, in the "Create Storyboard" step, also a **StopTrigger** is created, given that the *Storyboard* ontology individual has the "*has_stop_trigger*" assertion. If this is not the case, a default **StopTrigger** is created and added to the **Storyboard**. At last, a **Story** is created.

With the creation of the **Story**, we come to the second and final part of the XOSC file generation - the **Story** initialization.

4.5.1.2 Story initialization

Before explaining the whole flow diagram, it is important to note, that the diagram is divided into 6 different loops, which are numbered, so it's easier to follow the diagram. In addition, all of the mentioned loops are nested, and each loop has all of the following loops(loops with greater number) nested within it. Furthermore, the red path of the diagram, is the path that the algorithm is supposed to follow during the first iteration. The black path then represents the path of the program when there are no more elements in the corresponding arrays and the description goes to the next *Story/Act/ManeuverGroup/Maneuver* or *Event*.

As shown in Figure 4.7 the first loop is the *has_story empty?*. It is important to note, that at the end of this loop, the created **PYOSCX Story** will be added to the **PYOSCX Storyboard**. So in this step, at first the objects in the return array of the *has_story* object property(See Table 4.2.2) are checked, and if there are still elements in the array, the algorithm comes to the initialization step and a new **PYOSCX Story** is created, ready to be filled with **PYOSCX Act**s.

In the next step, the algorithm comes to the second loop, which creates a **PYOSCX Act**, fills it and then at the end, adds the **PYOSCX Act** to the **PYOSCX Story**. As shown in Figure 4.7, first, it is checked whether there are *Act* objects in the return array of the "*has_act*" object property(See Table 4.2.2). If it's not empty, then the creation of **PYOSCX StartTrigger**s and **StopTrigger**s begins. This happens with the help of the "addTrigger()" function (See Table 4.7). Once this is available, the initialization step of the **Act** can begin and a **PYOSCX Act** object is created. Furthermore, the current ontology individual *Act* is temporarily saved.

After this step, the algorithm comes to the third loop, which creates and fills **PYOSCX ManeuverGroup**s. As next, the "*has_maneuver_group*" object property of the *Act* is checked. If it's also not empty, then a **PYOSCX ManeuverGroup** object, with the name of the ontology *ManeuverGroup* individual, is initialized. Like the ontology *Act*, the ontology *ManeuverGroup* gets temporarily saved as well.
Afterwards, in the fourth loop, which is responsible for the creation and filling of **PYOSCX Maneuver**s, the algorithm checks whether the "*has_maneuver*" object property of the *ManeuverGroup* has more elements. If this is the case, then a **PYOSCX Maneuver** with name corresponding to the *Maneuver* ontology individual is initialized. The ontology *Maneuver* is also temporarily saved in a variable.
Afterwards, the next step is to create the **PYOSCX Event** and fill it with **Action**s. This happens the following way. First, the "*has_event*" array of the *Maneuver* is checked. If the array is not empty, the ontology *Event* is temporarily saved and the program enters the initialization step. In this step, the name of the element in the "*has_priority*" object property of the ontology event is also saved in a variable. Since the *Priority* is a constant already implemented within the ontology(See Section 4.2.4 and for the explanation - Section B.1), only the name of the individual is needed. The name can be accessed with the help of the "*getNameFromIri()*" function. Once the name is available, it's then compared to the different types of priorities[20] and an **PYOSCX Event** is created accordingly.
As a next step, the algorithm comes to the final sixth loop, which is responsible for the creation of the **PYOSCX Action**s. Here, the "*has_action*" object property of the ontology *Event* individual is checked. If it's not empty, then the ontology *Action* is temporarily saved and its class is then compared to the different available in the ontology action classes. Depending on the class various checks are performed and at the end the correct **PYOSCX Action** is created. Once the **Action** is created, afterwards it's added to the **PYOSCX Event** that was created earlier. At last, the ontology *Action* is removed from the "*has_action*" array. Then the process is repeated until there are no more *Actions* left in the "has_action" array of the current ontology *Event* variable.
Once there are no more *Action*s available, the flow chart can continue to the next step and add the **PYOSCX Event**, which is filled with **Action**s to the previously initialized **PYOSCX Maneuver**. Parallel with this, the ontology *Event* variable is removed from the "*has_event*" array.
Once again, then the "*has_event*" array is checked for emptiness and the process repeats. If however the array is empty, then the**PYOSCX Maneuver**, which at this point is filled with events, is added to the **PYOSCX ManeuverGroup** and the ontology *Maneuver* is removed from the "*has_maneuver*" array. The process is again repeated for all the *Maneuvers* in the array. Once this array is empty, the *ManeuverGroup* ontology variable gets the actor for which this *ManeuverGroup* is supposed to be via the "*has_entity_reference*" and adds it to the **PYOSCX ManeuverGroup**. Afterwards, the **PYOSCX ManeuverGroup** is added to the **PYOSCX Act** and the ontology *ManeuverGroup* is removed from the "*has_maneuver_group*" array. The whole process is once again repeated if the array is not empty.
If the array is empty, then the **PYOSCX Act** is added to the **Story** and the ontology *Act* is removed from the "*has_act*" array. While there are items in the resulted array, the entire flow, described

until now, is repeated. If, however, the array is empty, the algorithm comes to the next step, which is to add the **PYOSCX Story** to the previously created **PYOSCX Storyboard**. Afterwards, the *has_story* array is once more checked, and while its not empty, the whole process is repeated, for each *Story* of the description. When the array is empty, the algorithm comes to its final step. The final step of the flow chart (See Figure 4.7 is to create the **PYOSCX Scenario**. For the initialization of the **PYOSCX Scenario** several things are needed. First, A CARLA Town ID is required, which is taken from the *Scenario* ontology individual via the "*has_town*" object property (The towns are also a constant as showed in Section 4.2.4). Furthermore, a **RoadNetwork** and a **Catalog** are needed. The **RoadNetwork** is then created, depending on the *Town* individual and the **Catalog** is the default **PYOSCX Catalog**, since another one is not needed for the scenario_runner implementation.

When everything of the above-mentioned, is available and using the previously initialized **PYOSCX Entities** and **Storyboard**, a **PYOSCX Scenario** is created. With this said, the flow chart also comes to its end.

4.5.1.3 Combination of scenarios

In the current implementation, it is possible for the Onto2OpenSCENARIOConverter to generate a correctly defined OpenSCENARIO file, even if there are more than one described *Scenario*s in the ontology. This means that it is possible to combine different corner case scenarios and create a new scenario that contains within itself **all** of the scenarios. There are, however, some limitations of the scenario combination method, for further information, please refer to Section 5.9.

As seen in the previous section, the Onto2OpenSCENARIOConverter works with only one *Scenario*, which leads to the need for adjustment for an ontology with more than one *Scenario*. In the following paragraph, I will explain how it works.

In simple words, the resulted ontology is adjusted and a new *Scenario* with the combined *Storie*s and *Init*s from the other *Scenario*s is created. The reconstruction works as follows. At first, a "main" *Scenario* individual is created, which is then connected with every *has_entity* connection that the *Scenario*s have. Afterwards, the *Storyboard*s of the *Scenario*s are read and their *Init*, which comes from the property assertion *has_init*, is saved into a buffer variable. The *has_init_action* property assertions of the buffer *Init* are asserted to the new *main_init* individual. This is repeated for each *Scenario*. Afterwards, the *has_story* property assertions of the buffer *Storyboard*s are transferred to the "main" *Storyboard*. When this is done, the combined *Scenario* is ready and is then read as it was earlier explained in this chapter. The resulted **.xosc** file can be directly executed in CARLA ScenarioRunner[10].

4.5.1.4 Discussion

After running the **Onto2OpenSCENARIOConverter** script on the resulted ontologies, as mentioned earlier, a **.xosc** file is generated (See Section 4.1). The resulted **.xosc** file is correctly defined and thus can also be directly, without any further actions, executed within the scenario runner. This is also true for the combined scenarios.

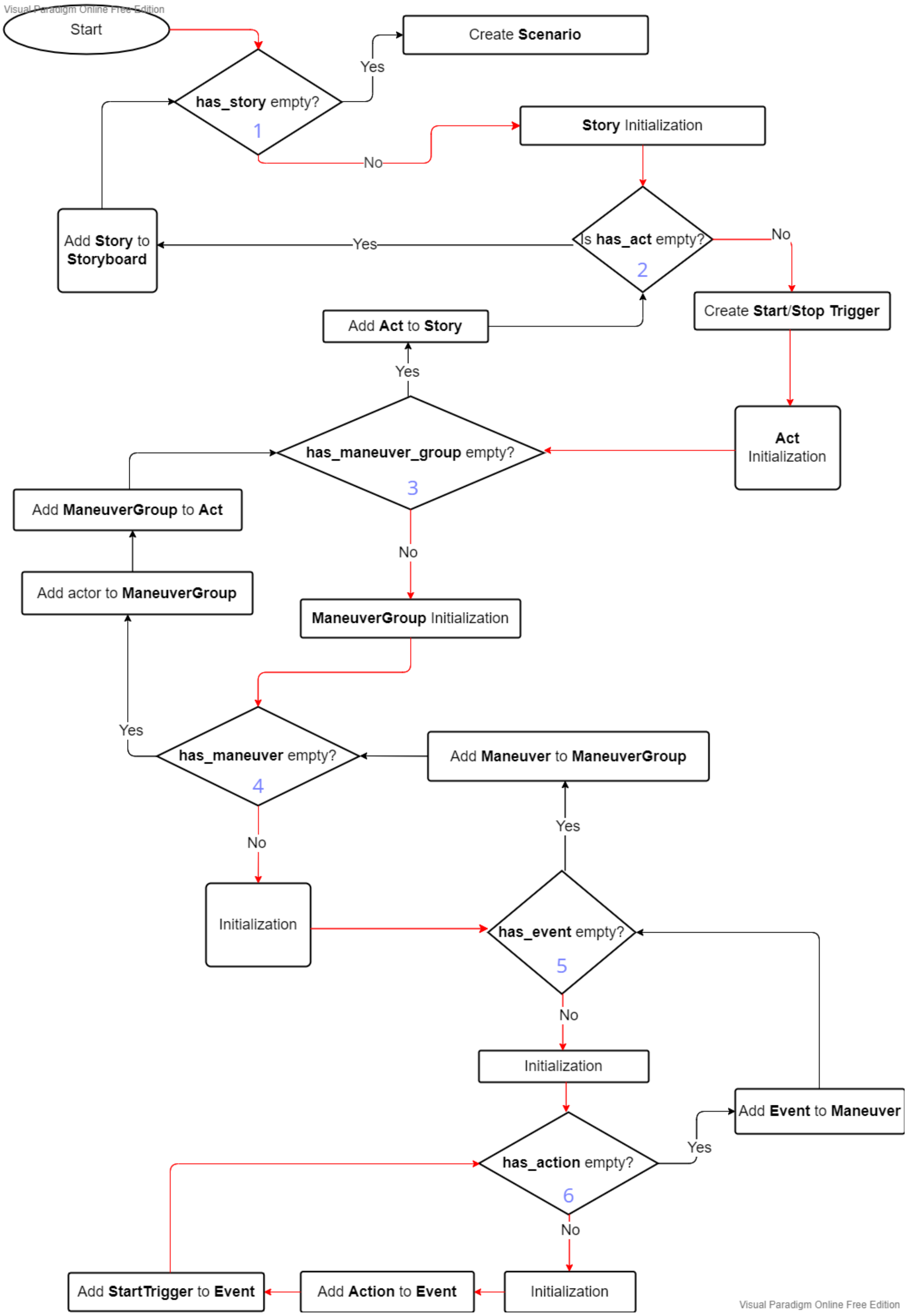


Figure 4.7: Flow diagram of the story initialization

5 Qualitative Evaluation

After the OntologyGenerator is executed, an ontology is generated depending on the selected OntologyGenaration method. The following section shows the resulting ontologies and scenarios after executing the corresponding method. To better understand the resulting ontology graphs, it is important to know that all new individuals have a name starting with, "*indiv_*" and anything starting with a capital letter is an ontology class. Furthermore, as mentioned in Section 4.2.4, all the constants in the ontology, have only small letters. The arrows in the figure represent the object properties which were used to describe the scenario. Furthermore, it is important to note that the template ontology consists of 67 individuals (constants and default individuals, see Section 4.2.4 and 4.2.5). The videos of the resulting scenarios are available in the open-source GitHub of the thesis [64].

5.1 Dead Pixel

For the dead pixel scenario, there is a separate script created. The script takes images as input and then puts a random amount of dead pixels in it. This means, that the results of this corner case, can be applied on every other corner case category scenario. The results can be seen in Figure C.2(a).

5.2 Domain Shift scenario

The scenario simulates a car that suddenly enters a dense fog after driving about 70 meters.
Figure D.2 shows the connections between the different individuals and also the classes used to describe the scenario. The resulted *IntoFogOntology* has 94 individuals, which means that for this scenario, 27 Individuals were created by the OntologyGenerator Script. The *Scenario* was modeled with only one *Entity* inside – the ego vehicle.
There are two *Events* – the main one, which includes the *EnvironmentAction* and one which is only there to keep the simulation going a little longer after the vehicle drives into the fog. As mentioned in section 2.4.1.3, the *EnvironmentAction* is used to change the environment within the simulation, creating a dense fog in our case. Follow Figure D.2 to see the further relationships. There are also 3 *StartTrigger*s, 2 of which are used in the *Event*s and one is used to stop the *Story*.
The *StopTrigger* of the *Story* has a *SimulationTimeCondition* and a corresponding *ConditionGroup*. In this scenario, two *SimulationTimeCondition*s were needed - one for the *StartTrigger* of the *Act* and one for the *StopTrigger*. Also, as mentioned earlier, the fog suddenly appears after the vehicle has traveled at least 70 meters. This is done using the individual **indiv_DistanceStartTrigger**, which is a *StartTrigger* with a *TraveledDistanceCondition* in its *ConditionGroup*. As it can also be seen in Figure D.2, the **indiv_70TraveledDistanceCondition** also

has a reference to the *ego_vehicle*. In addition, the condition also has a data property "*has_travaled_distance_value*", which in this case is 70 (See Table 4.3).
The "_KeepLongerEvent" individual is an *Event*, that has as a *StartTrigger*, which triggers when the previous *Event* ends and this is possible with the help of the *StoryboardElementStateCondition*. This condition has then also the *has_storyboard_element_ref* property assertion, connected to the previous *Event*, which in this case is the *__ChangeWeatherEvent*. When this *Event* reaches "completeState", then the "_KeepLongerEvent" starts. The said *Event* then just changes the speed of the ego vehicle, but gives enough time to see the results of the previous *Event*s before the episode finishes.

5.3 Single-Point Anomaly scenario

The scenario simulates a car driving and suddenly a coca-cola vending machine appears on the road and falls near the vehicle.
The resulted ontology consists of 93 individuals, which means that for this scenario, 26 new individuals were created. The relations and individuals can be seen in Figure D.5 and the corresponding picture in Figure C.3(c) Just like in the into fog scenario, this scenario has the same *StartTrigger* triggered at 0 simulation time and the *StopTrigger* triggered at 30 simulation time. Furthermore, it has a *StartTrigger*, which triggers after the ego vehicle is closer than 15 meters to the vending machine. This scenario also uses the default ego vehicle and has as *InitAction* the *def_env_action*. In addition, there is a *TeleportAction* with *RelativeObjectPosition*, that is used to push the vending machine to the ground when the ego vehicle is close enough. The *Action* is triggered, with the help of the previously mentioned *StartTrigger*. It is important to note, that in order to change smoothly the position of an entity, one should always use *RealtiveObjectPosition* with entity reference, the same entity.

5.4 Collective Anomaly

In this scenario, N amount of people and the ego vehicle are spawned on the same street. The pedestrians are moving slowly on the road and after some time they start running. The extraordinary is the amount of the pedestrians. The relations and individuals can be seen in Figure D.4 and the corresponding picture in Figure C.3(a).
The resulted ontology consists of 164 individuals, which means that for this scenario, 97 new individuals were created. In the real resulting scenario, the pedestrians are 10, but in order to have a readable graph, there are only 3 pedestrians within it. Everything that those 3 pedestrians have as assertions, exists as assertion by the rest of the individuals. The scenario, has for every pedestrian a *TeleportAction* to spawn it at the correct place. The lane is the same, but the "s" increases for every pedestrian with 4, to avoid collisions. Furthermore, each pedestrian has 2 *SpeedAction*s. One for the *Init* and one for the *Story*, which forces the pedestrian to run, instead of walk. Furthermore, there is an event which, just like in the other simulations, is with the purpose to keep the simulation going a bit longer. The resulted ontology consists of 155 individuals, which means that for this

scenario, 88 new individuals were created.

5.5 Contextual Anomaly

The scenario simulates a car driving and suddenly 3 traffic signs appear on the road and fall in front of the vehicle. The relations and individuals can be seen in Figure D.3 and the corresponding picture in Figure C.3(b). The resulted ontology consists of 111 individuals, which means that for this scenario, 44 new individuals were created. The scenario has the *ego_vehicle* default individual and 3 new *Entities* - the traffic signs. Every sign has a *TeleportAction* with its corresponding *Position* for the *Init* and then a *TeleportAction* with a corresponding *RelativeObjectPosition*, for the *Story*. The second *TeleportAction* is used for the falling *Event* of the street signs. Furthermore, every *Event* has a corresponding *Maneuver*, *ManeuverGroup* and *StartTrigger*. The *StartTrigger* triggers when the car is has traveled at least 70 meters.

5.6 Risky Scenario

The scenario simulates two vehicles. Both vehicles start on the same lane. A vehicle follows the ego vehicle and when reaching a certain distance between the ego vehicle and itself, the vehicle changes to the left lane. When the vehicle passes the ego vehicle, it makes again a lane change, back to the right lane. The second maneuver is very risky, because, the lane change happens too close to the ego vehicle. The corresponding scheme to this scenario can be seen in Figure D.7, the corresponding Picture in Figure C.5. The resulted ontology consists of 104 individuals, which means that for this scenario, 37 new individuals were created. In this scenario, there are different *StartTrigger*s. There are the normal *StartTrigger* and *StopTrigger*, which trigger on 0 and 30 *SimulationTime*. Furthermore, there is a *StartTrigger*(**indiv_DistanceTrigger2StartTrigger**), that triggers, when the second vehicle is less than 15 meters to the ego vehicle. This trigger is used for a *RelativeLaneChangeAction* to the left. Afterwards, another *StartTrigger*(**indiv_DistanceTriggerStartTrigger**) is used to trigger the *RelativeLaneChange* to the right. This happens after the vehicle has driven 70 meters. The rest of the individuals are used to keep the simulation going longer, change the environment in the beginning and spawn the entities at the correct place. It is important to note here, that this scenario was especially hard to model. At first, there is no available map with the IDs of the different roads in CARLA towns. This lead to the creation of random *TeleportAction*s, only to find the ID of the desired street. In addition, as mentioned earlier (See Section 4.4.2.1.1), not only the “road ID” is required but also a “lane ID”and a “s”. The cut-in scenario required random testing of different values in order to force the second vehicle to make a really close cut-in.

5.7 Novel Scenario

The scenario simulates a bicycle rider doing strange maneuvers in the opposite lane of the ego vehicle. The corresponding scheme to this scenario can be seen in Figure D.6, the corresponding Picture in Figure C.5. The resulted ontology consists of 94 individuals, which means that for this scenario, 27 new individuals were created. In this scenario, there are only a cyclist and the ego vehicle, as participants. Both are spawned on the same road, but in opposite lanes. The main *Event* here is, when the cyclist starts driving strange. This is achieved with the help of an *TeleportAction* with *RelativeObjectPosition*. The *Event* is triggered by a *StartTrigger* with *TraveledDistanceCondition* with the rule *greaterThan* 20. This means, after the cyclist has traveled 20 meters, then it triggers. There is one more *Event* in this scenario, which is responsible to keep the scenario longer.

5.8 Anomalous Scenario

The scenario simulates a pedestrian, that waits for the vehicle to get closer and suddenly runs very close, in front of the vehicle. The corresponding scheme to this scenario can be seen in Figure D.8, the corresponding Picture in Figure C.5. The resulted ontology consists of 95 individuals, which means that for this scenario, 25 new individuals were created. The scenario has a *StartTrigger*, which triggers a *TeleportAction* and a *SpeedAction* for the pedestrian, forcing him to face the street and start running towards the ego vehicle. The mentioned *StartTrigger* is with a *RelativeDistanceCondtion* with value 15, which means that when the ego vehicle is closer than 15 meters, the trigger starts.

5.9 Combinatorial Scenarios

Furthermore, the scenarios were tested, whether they can be combined. The tests show that if both scenarios are made for the same map and street, then most probably the combination will work without problems. There is however a further restriction to this method, if the scenarios spawn entities on at the same place, then this would lead to a conflict and one of the entities probably will not be spawned. However, if for example one of the scenarios is a domain shift, that includes only environment change, then this scenario can be combined with any other scenario, since the environment can be changed everywhere. It is also important to note, that the default individual *ego_vehicle* must be used. Figure D.9 shows the resulted ontology, after merging two scenarios - the collective anomaly scenario and the novel scenario. Figure C.2 shows a picture from the resulting Scenario.

6 Conclusion

This bachelor thesis focuses on ontology-based corner case scenario simulations. As the thesis highlights, the amount of existing data containing corner cases is nowhere near the amount needed to sufficiently train autonomous driving systems. The state-of-the-art shows, that in today's world there is no way to describe and generate all of the corner case categories introduced by Breitenstein et al. [14, 26]. There are approaches which are able to describe and generate scenarios, but not specifically the previously mentioned categories. This thesis proposes a technique to describe and automatically generate and classify corner cases. The approach is based on an template ontology, with which one can describe different corner case scenarios and classify some of them with the help of a semantic reasoner and SWRL Rules. With the proposed technique, I described, generated, and simulated seven corner cases in the autonomous driving simulator CARLA. There is one corner case for each corner case category presented by Breitenstein et al. [14]. The examples have been carefully selected from a variety of sources so that they cover all categories. In order to achieve the second goal of this thesis, I developed a python library named OntologyGenerator, which is used to replace the exhausting individual creation within the ontology. In addition, the OntologyGenerator prevents its user from making description mistakes, which can quickly happen by manual work with an ontology. To simulate the corner cases described in the ontology, the so-called Onto2OpenSCENARIOConverter was developed. The Onto2OpenSCENARIOConverter takes an ontology created by the OntologyGenerator as an input and returns an OpenSCENARIO file format as an output, which can be directly executed in the simulation environment CARLA. Furthermore, the resulted approach allows the user to describe and generate a single scenario and combine the single scenarios into one whole scenario. This results in a powerful tool for combining corner case scenarios and their easy scalability.

Each scenario is represented as an ontology, from which a correctly generated **.xosc** file was exported and resulted in a simulation, looking as expected. Furthermore, a combination of the described scenarios is available and can also be successfully executed in CARLA. The resulting simulation data can be retrieved and used for scenario-based validation.

6.1 Challenges

The complex OpenSCENARIO syntax and its many variables was one of the challenges that had to be overcome. Furthermore, I wanted to implement slightly different scenarios, i.e. instead of vending machine, a tree in the middle of the street, but the integration of new assets within CARLA was harder than expected and because of time limits, impossible to finish on time. However, once new assets are integrated into CARLA, they can be used directly in the ontology without any problems.

6.2 Future Work

Since the ontology is still incomplete, in the future, the whole OpenSCENARIO can be implemented within it and also within both scripts. For example, some *Condition*s are still missing to provide the *StartTrigger*s with even more possibilities. Furthermore, more *Action*s can be added, to also provide more possibilities for scenarios. Everything, said so far, can be achieved by following the implementation of the already finished OpenSCENARIO elements. Further SWRL rules can be written to classify the described corner cases. This can also be upgraded with help of the rules presented by Huang et al. [43]. Since the Prolog language [8] is very similar to the SWRL Rules syntax (See 2.2.5.1), the proposed rules can be used as extension to the SWRL rules of this thesis. In addition, the ontology can be further developed, by adding more default individuals. Also, a graphical user interface can be built for the OntologyGenerator for easier scenario generation. Another possibility to make the development of the scenarios easier is by creating maps of CARLA with already marked road IDs.

Furthermore, the proposed technique would be much more useful, if one can simulate a lot of corner cases fast. To this end, a similar mass way of Scenario Generation, that was conducted by Menzel et al. [49] can be applied to the proposed in this thesis ontology. Another possibility would be to create base scenario parts (i.e. *Event*s) and then use variations to combine them and create different scenarios. In addition, lidar data can be extracted in the future within the scenarios, by the sensors provided in CARLA. To conclude, this thesis significantly contributes to the description and generation of corner cases for autonomous driving. Even though the resulting ontology is incomplete, it provides the base and a good starting point for scenario-based corner case validation. Most importantly, the proposed technique is scalable and can be easily extended.

Appendices

A Functions

A.0.1 getNameFromIri

This is a very important function within the OntologyGenerator, because it is used by almost every comparison of the ontology.
When using the owlready2 library [7] and reading an ontology, almost every part of it gets returned as an *iri*. An *iri* looks, for example, the following way:
"*http://www.semanticweb.org/stefi/ontologies/2021/10/21/untitledontology56#CornerCase*"
This however is not the information one normally wants to retrieve. Actually, only the last part after the # is needed. This would be in this example the name of the class - *CornerCase*. For this reason, the *getNameFromIri* function was created. The function always returns whatever comes after the first # and since this is always the only # in an *iri*, it always works.

A.1 Predicates

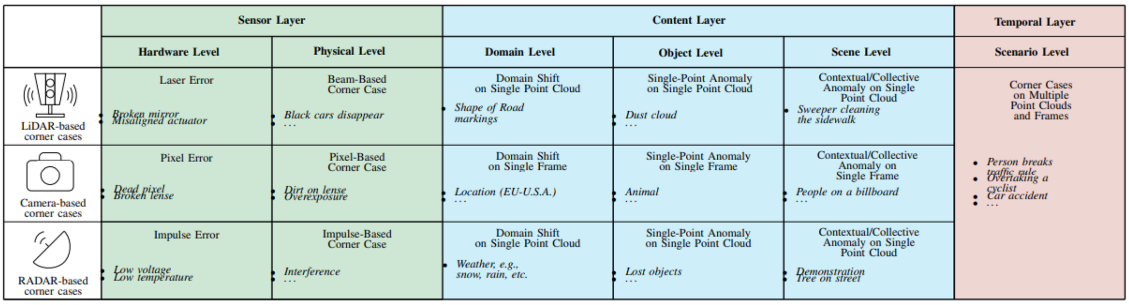
The classes and the properties can be seen as a predicate. A predicate is a boolean function, which in the case of data and object properties takes two parameters (domain and range of the property) and in the case of class takes only one parameter. The Domain is the left part of a property and the Range is the right part of the property. For example, if we take the random object property *has_age*, it would have the domain *Person* and the range *integer*.

B OpenSCENARIO further attributes

B.1 Priority

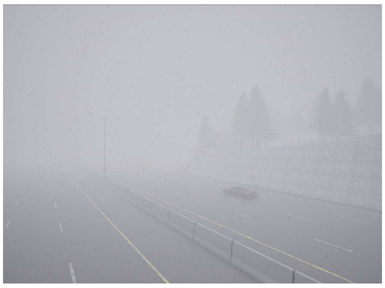
The **Priority** in OpenSCENARIO is an attribute of the **Event**. There are 3 types of priority - *overwrite, parallel* and *skip*. The *overwrite* is used when one wants to stop every other **Event**s executing in the corresponding **Maneuver** and run only this **Event**. If the *skip* **Priority** is set, then the corresponding **Event** will be only executed, if there aren't any other currently running **Event**s. And at last it the *parallel* **Priority**, which is executed parallel to the other **Event**s in the **Maneuver**.

C Corner case categories and simulation results with pictures

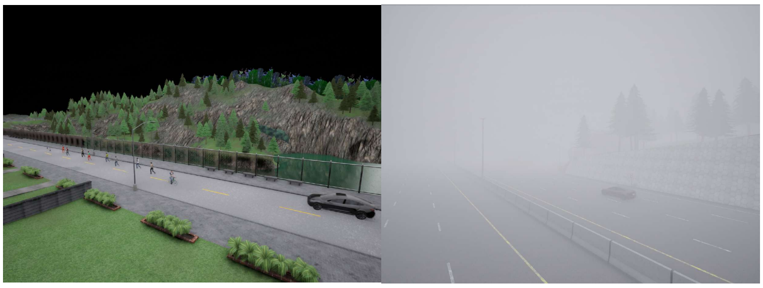


	Sensor Layer		Content Layer			Temporal Layer
	Hardware Level	Physical Level	Domain Level	Object Level	Scene Level	Scenario Level
LiDAR-based corner cases	Laser Error • *Broken mirror* • *Misaligned actuator*	Beam-Based Corner Case • *Black cars disappear* • ...	Domain Shift on Single Point Cloud • *Shape of Road markings*	Single-Point Anomaly on Single Point Cloud • *Dust cloud* • ...	Contextual/Collective Anomaly on Single Point Cloud • *Sweeper cleaning the sidewalk*	Corner Cases on Multiple Point Clouds and Frames • *Person breaks traffic rule* • *Overtaking a cyclist* • *Car accident* • ...
Camera-based corner cases	Pixel Error • *Dead pixel* • *Broken lense*	Pixel-Based Corner Case • *Dirt on lense* • *Overexposure*	Domain Shift on Single Frame • *Location (EU-U.S.A.)* • ...	Single-Point Anomaly on Single Frame • *Animal* • ...	Contextual/Collective Anomaly on Single Frame • *People on a billboard* • ...	
RADAR-based corner cases	Impulse Error • *Low voltage* • *Low temperature*	Impulse-Based Corner Case • *Interference* • ...	Domain Shift on Single Point Cloud • *Weather, e.g., snow, rain, etc.*	Single-Point Anomaly on Single Point Cloud • *Lost objects* • ...	Contextual/Collective Anomaly on Single Point Cloud • *Demonstration* • *Tree on street*	

Figure C.1: Corner case Categorization from Heidecker et al. [41]

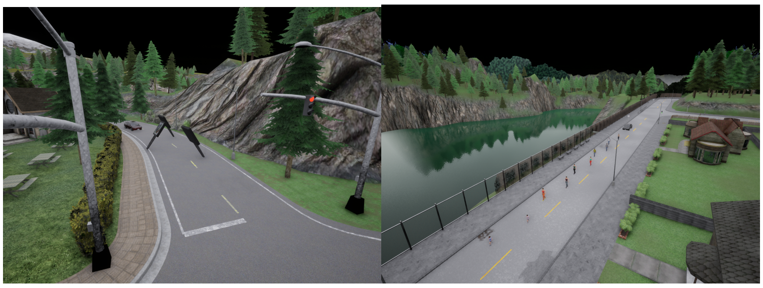


(a) Dead Pixel



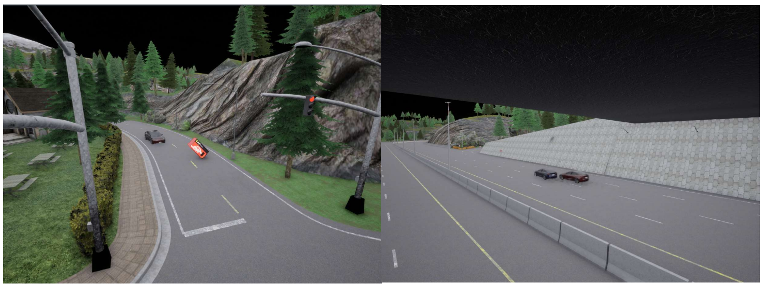
(b) Merged scenario (c) Domain Shift

Figure C.2: Pixel Level, Merged Scenario and Domain Level Scenario results



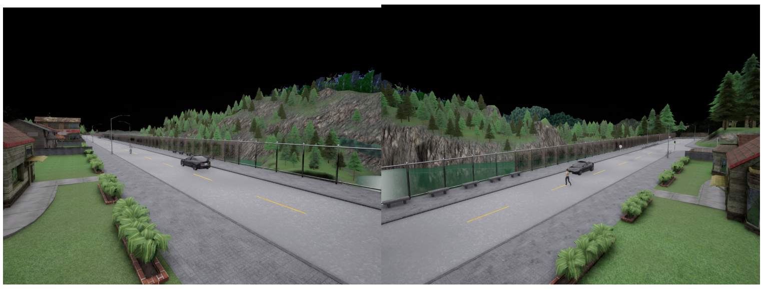
(a) Contextual Anomaly (b) Collective Anomaly

Figure C.3: Object and Scene Level Scenario results



(a) Single-Point Anomaly (b) Risky Scenario

Figure C.4: Object Level and Scenario Level results



(a) Novel Scenario (b) Anomalous Scenario

Figure C.5: Rest of Scenario Level Scenario results

D Figures

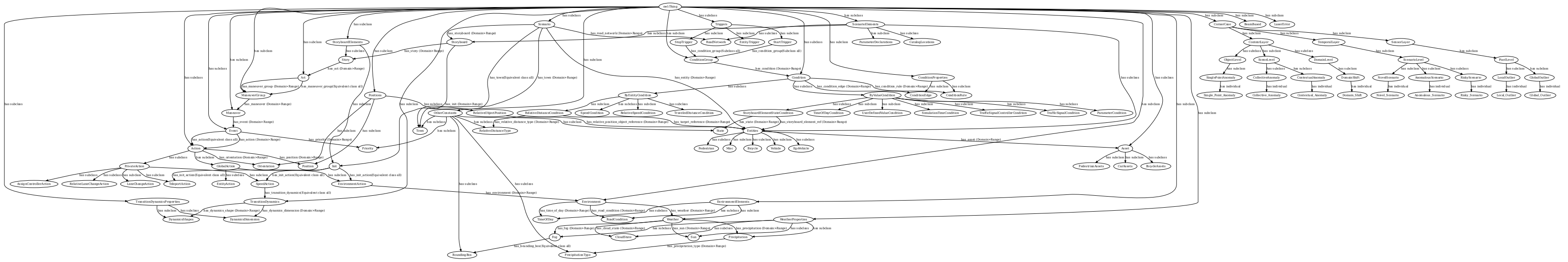


Figure D.1: Diagram of the whole ontology structure

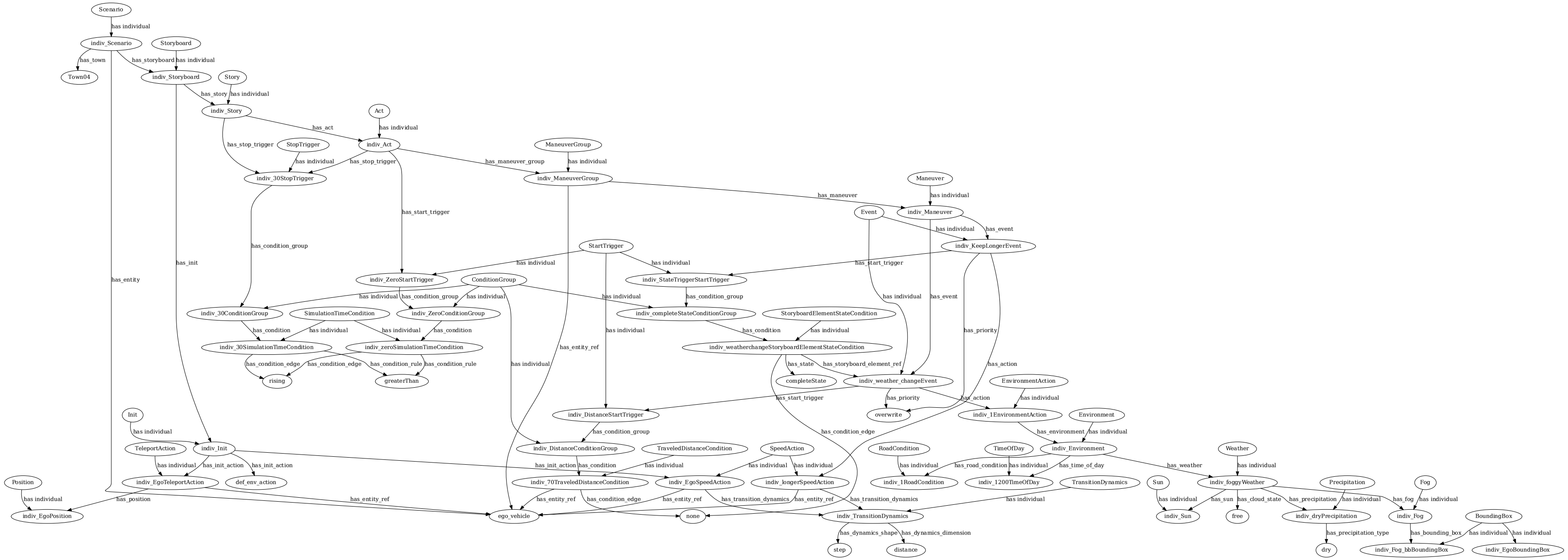


Figure D.2: Graph of the IntoFog Scenario description

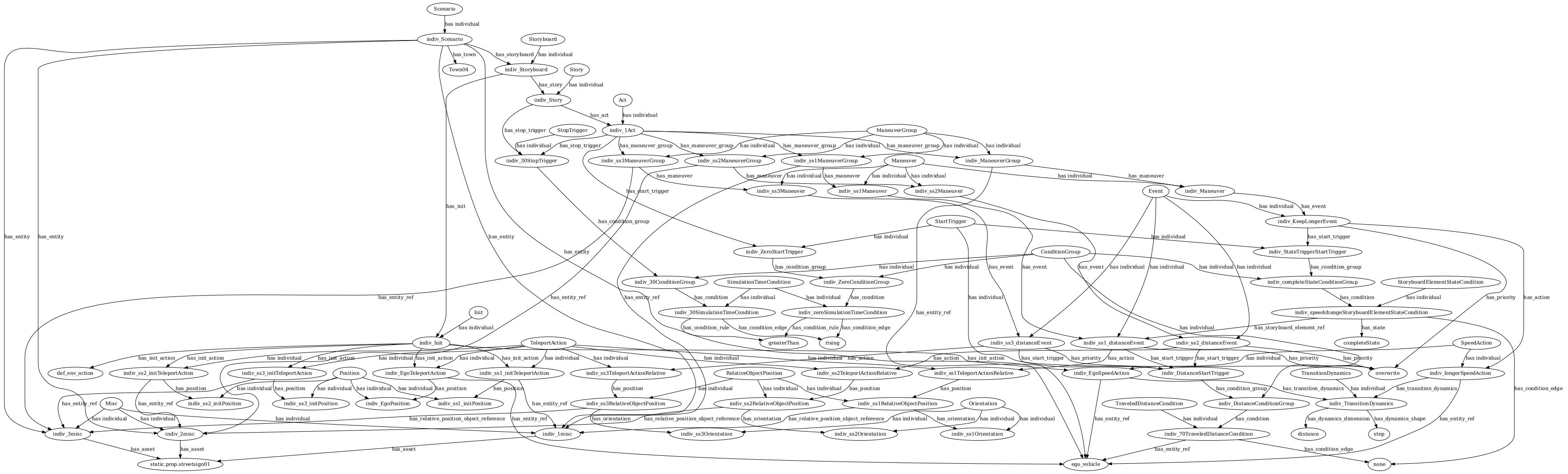


Figure D.3: Relations between the individuals in a contextual anomaly scenario

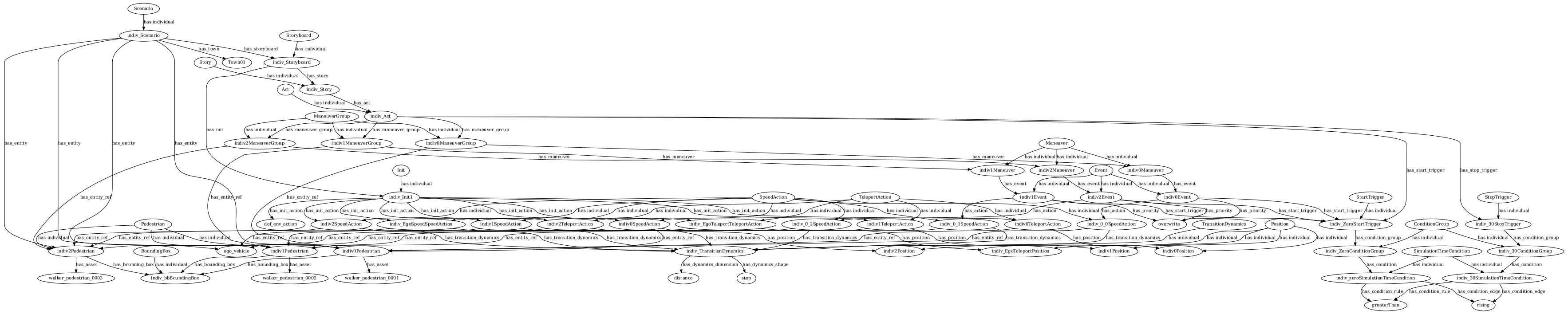


Figure D.4: Relations between the individuals in a collective anomaly scenario

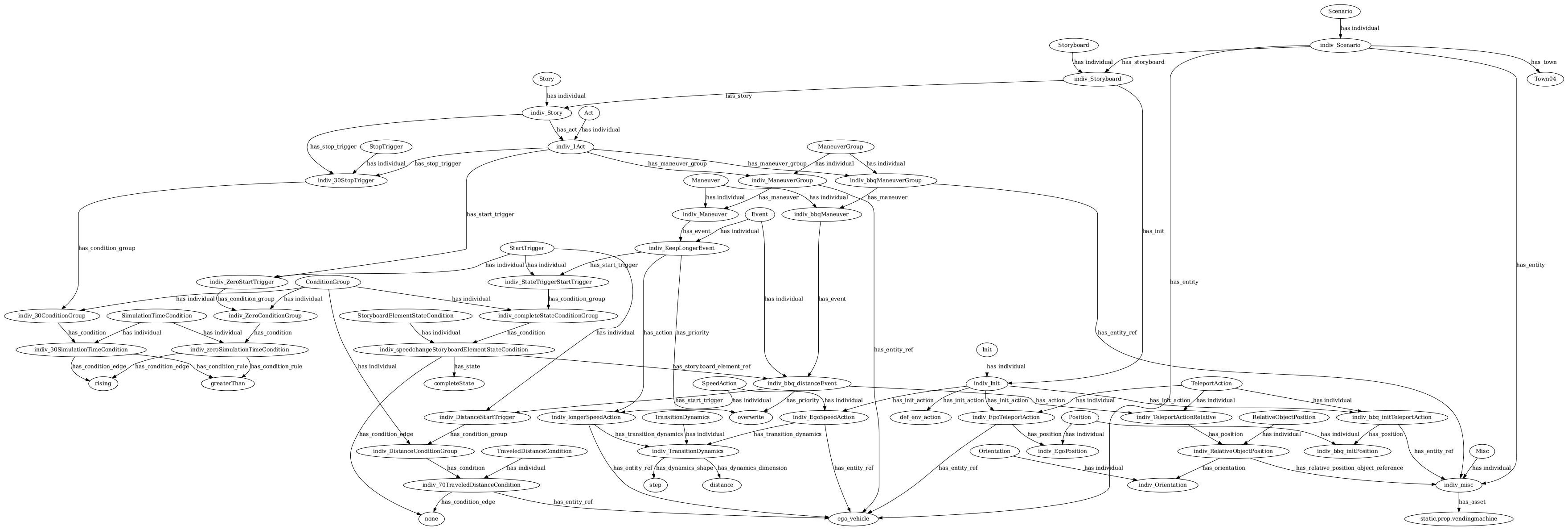


Figure D.5: Relations between the individuals in a Single-Point anomaly scenario

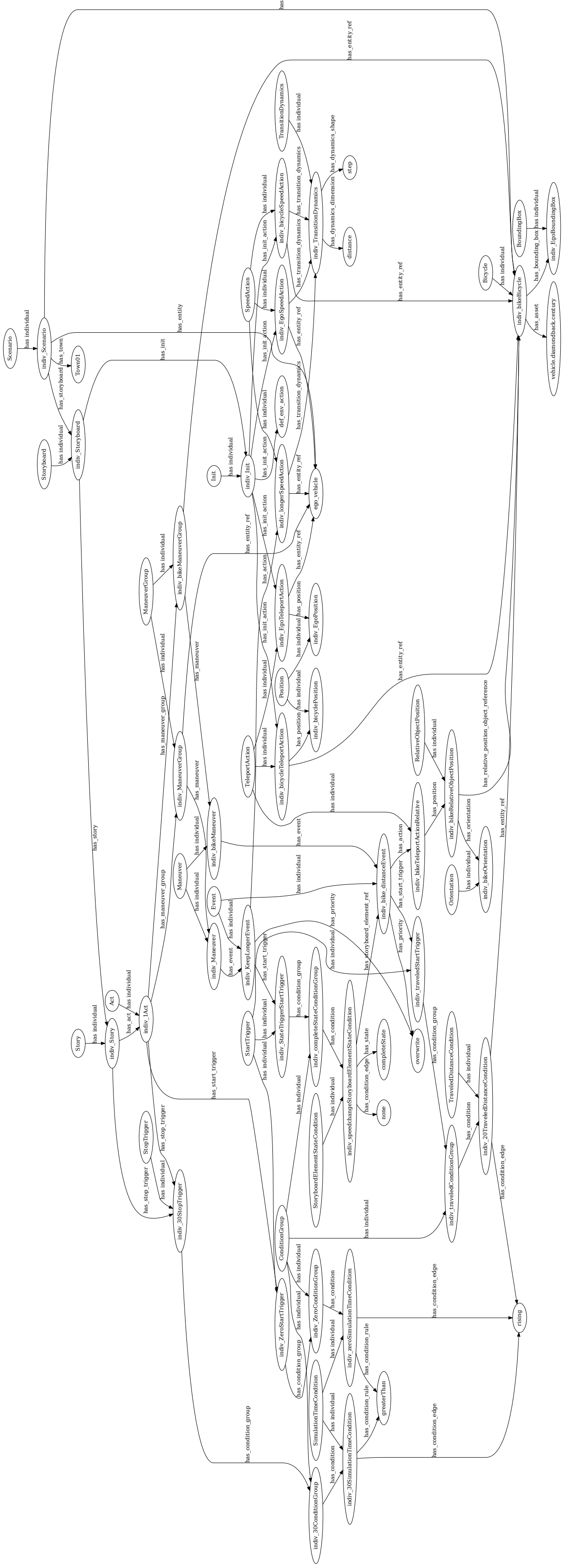


Figure D.6: Relations between the individuals in a novel scenario

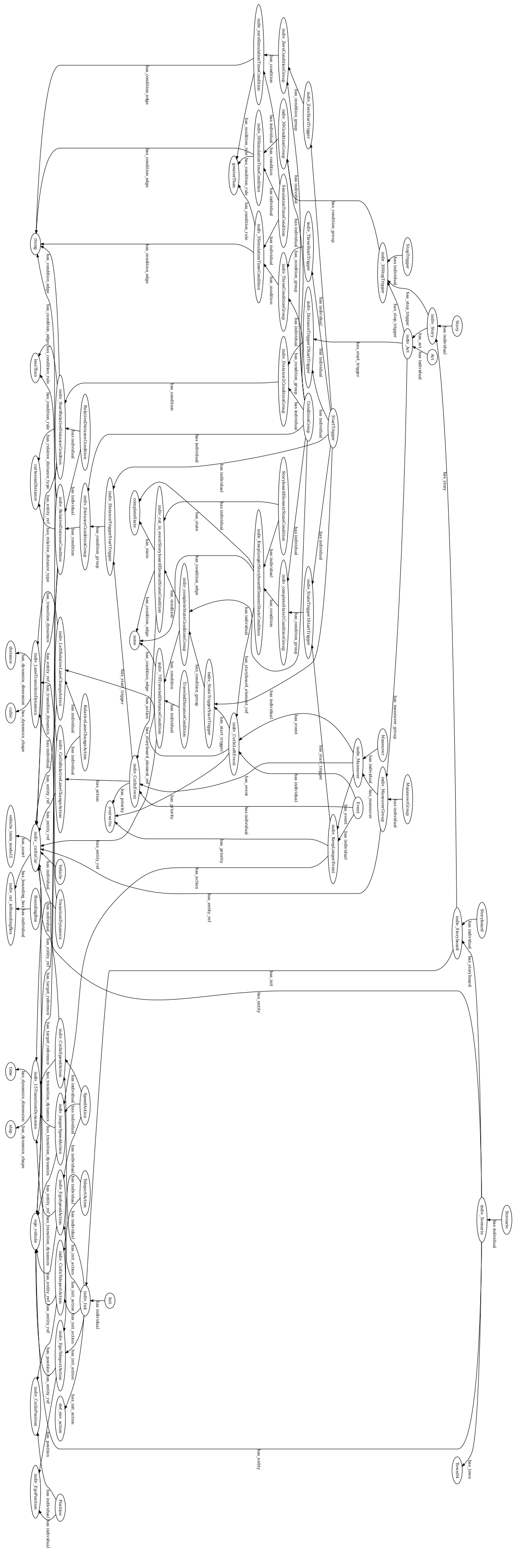


Figure D.7: Relations between the individuals in a risky scenario

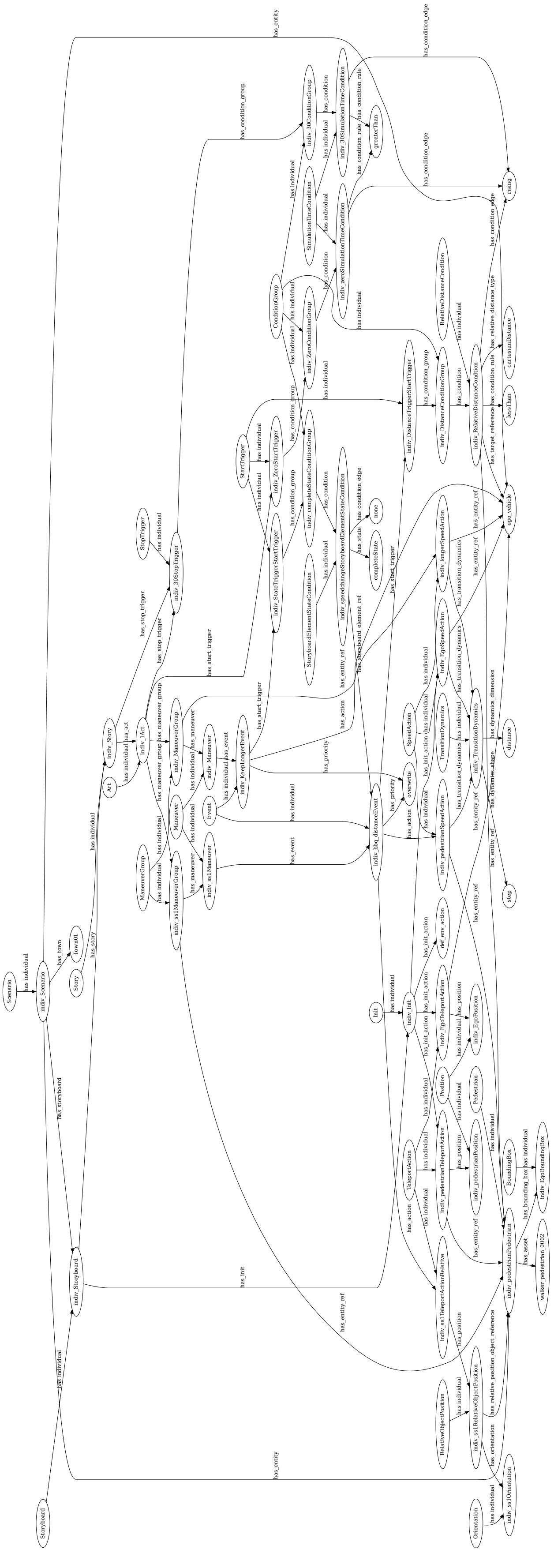


Figure D.8: Relations between the individuals in an Anomalous scenario

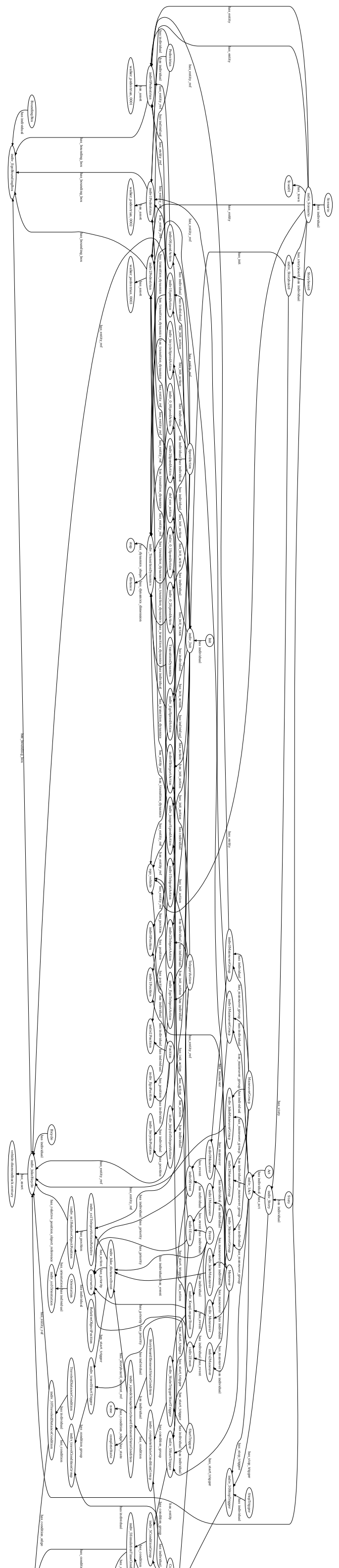


Figure D.9: Relations between the individuals in the merged scenario(Collective Anomaly and Novel Scenario)

A List of Figures

2.1 Overview of the architecture of an ontology [23] 5
2.2 ASAM OpenSCENARIO Scenario Elements[20] 11

4.1 Flow diagram of the proposed ontology-based technique for description and generation of corner case scenarios. 19
4.2 Metrics of the Ontology 21
4.3 Top Level classes of the ontology, subclasses of **owl:Thing** 22
4.4 Corner case categories description within the ontology 26
4.5 Flow Chart of a scenario generation within the ontology 27
4.6 Flow Chart of the Init construction. 31
4.7 Flow diagram of the story initialization 35

C.1 Corner case Categorization from Heidecker et al. [41] 46
C.2 Pixel Level, Merged Scenario and Domain Level Scenario results 47
C.3 Object and Scene Level Scenario results 47
C.4 Object Level and Scenario Level results 48
C.5 Rest of Scenario Level Scenario results 48

D.1 Diagram of the whole ontology structure 50
D.2 Graph of the IntoFog Scenario description 51
D.3 Relations between the individuals in a contextual anomaly scenario 52
D.4 Relations between the individuals in a collective anomaly scenario 53
D.5 Relations between the individuals in a Single-Point anomaly scenario 54
D.6 Relations between the individuals in a novel scenario 55
D.7 Relations between the individuals in a risky scenario 56
D.8 Relations between the individuals in an Anomalous scenario 57
D.9 Relations between the individuals in the merged scenario(Collective Anomaly and Novel Scenario) 58

B List of Tables

4.1 Object properties within the ontology 23
4.2 Help object properties within the ontology 23
4.3 Data properties within the ontology 24
4.4 Constant Individuals 24
4.5 Default Individuals 25
4.6 OntologyGenerator methods, where RPAO stands for RelativePositionAndOrientation, and ES stays for ElementState 28
4.7 Functions implemented within the Onto2OpenSCENARIOConverter 30

C Bibliography

[1] A free, open-source ontology editor and framework for building intelligent systems. `https://protege.stanford.edu/`. Accessed: 2022-02-28.

[2] Airsim. `https://microsoft.github.io/AirSim/`. Accessed: 2022-02-28.

[3] CARLA Simulator. `https://carla.org/`. Accessed: 2022-02-28.

[4] Cityscape Dataset. `https://www.cityscapes-dataset.com/dataset-overview/`. Accessed: 2022-02-28.

[5] Class Condition. `https://releases.asam.net/OpenSCENARIO/1.0.0/Model-Documentation/content/Condition.html`. Accessed: 2022-02-28.

[6] Esmini github. `https://github.com/esmini/esmini`. Accessed: 2022-02-28.

[7] OWLReady Documentation. `https://owlready2.readthedocs.io/en/v0.36/`. Accessed: 2022-02-28.

[8] Prolog. `https://en.wikipedia.org/wiki/Prolog`. Accessed: 2022-02-28.

[9] Scenario generation(pyoscx library) github. `https://github.com/pyoscx/scenariogeneration`.

[10] Scenario Runner Github. `https://github.com/carla-simulator/scenario_runner`. Accessed: 2022-02-28.

[11] SWRL Documentation. `https://www.w3.org/Submission/SWRL/`. Accessed: 2022-02-28.

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

[13] Virtual test drive. `https://www.mscsoftware.com/product/virtual-test-drive`. Accessed: 2022-02-28.

[14] Corner cases for visual perception in automated driving: Some guidance on detection approaches. 2021.

[15] M. Althoff, O. Stursberg, and M. Buss. Safety assessment of autonomous cars using verification techniques. In *2007 American Control Conference*, 2007.

[16] C. Amersbach and H. Winner. Defining required and feasible test coverage for scenario-based validation of highly automated vehicles*. In *IEEE Intelligent Transportation Systems Conference (ITSC)*, 2019.

[17] A. Armand, D. Filliat, and J. Ibañez-Guzman. Ontology-based context awareness for driving assistance systems. In *2014 IEEE Intelligent Vehicles Symposium Proceedings*, 2014.

[18] ASAM. ASAM OpenSCENARIO. `https://www.asam.net/standards/detail/openscenario`. Accessed: 2022-02-28.

[19] ASAM. ASAM OpenXOntology. `https://www.asam.net/project-detail/asam-openxontology/`. Accessed: 2022-02-28.

[20] ASAM. OpenSCENARIO Documentation. `https://releases.asam.net/OpenSCENARIO/1.0.0/ASAM_OpenSCENARIO_BS-1-2_User-Guide_V1-0-0.html`. Accessed: 2022-01-28.

[21] ASAM. OpenSCENARIO Documentation Global Action. `https://releases.asam.net/OpenSCENARIO/1.0.0/Model-Documentation/content/GlobalAction.html`. Accessed: 2022-01-28.

[22] F. Baader. Tableau Algorithms for Description Logics. `https://lat.inf.tu-dresden.de/~baader/Talks/Tableaux2000.pdf`. Accessed: 2022-02-28.

[23] G. Bagschik, T. Menzel, and M. Maurer. Ontology based scene creation for the development of automated vehicles. In *IEEE Intelligent Vehicles Symposium (IV)*, 2018.

[24] T. Berners-Lee. Semantic Web. `https://www.w3.org/standards/semanticweb/`. Accessed: 2022-02-28.

[25] B. P. Boris Motik, Peter F. Patel-Schneider. Owl 2 web ontology language structural specification and functional-style syntax. `https://www.w3.org/TR/owl2-syntax/`. Accessed: 2022-02-28.

[26] J. Breitenstein, J.-A. Termöhlen, D. Lipinski, and T. Fingscheidt. Systematization of corner cases for visual perception in automated driving. In *IEEE Intelligent Vehicles Symposium (IV)*, 2020.

[27] V. N. Chart. Car Insurance Scam Attempt Fail Compilation. `https://www.youtube.com/watch?v=pt7-pJPw5t4&ab_channel=ViralNewsChart`. Accessed: 2022-02-28.

[28] W. Chen and L. Kloul. An ontology-based approach to generate the advanced driver assistance use cases of highway traffic. 2018.

[29] F. Codevilla, E. Santana, A. M. Lopez, and A. Gaidon. Exploring the limitations of behavior cloning for autonomous driving. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019.

[30] F. Concas, J. Nurminen, T. Mikkonen, and S. Tarkoma. Validation frameworks for self-driving vehicles: A survey. 2020.

[31] R. Couto, A. Ribeiro, and J. Campos. Application of ontologies in identifying requirements patterns in use cases. *Electronic Proceedings in Theoretical Computer Science*, 147, 2014.

[32] K. Dentler, R. Cornet, A. Teije, and N. de Keizer. Comparison of reasoners for large ontologies in the owl 2 el profile. *Semantic Web*, 2, 2011.

[33] J. Dyble. Understanding SAE automated driving – levels 0 to 5 explained. `https://technologymagazine.com/ai-and-machine-learning/understanding-sae-automated-driving-levels-0-5-explained`. Accessed: 2022-02-28.

[34] S. Fuchs, S. Rass, B. Lamprecht, and K. Kyamakya. A model for ontology-based scene description for context-aware driver assistance systems. 2008.

[35] S. Geyer, M. Baltzer, B. Franz, S. Hakuli, M. Kauer, M. Kienle, S. Kwee-Meier, T. Weigerber, K. Bengler, R. Bruder, F. Flemisch, and H. Winner. Concept and development of a unified ontology for generating test and use-case catalogues for assisted and automated vehicle guidance. *Intelligent Transport Systems, IET*, 8, 2014.

[36] F. Ghorbel, N. Ellouze, E. Métais, F. Hamdi, F. Gargouri, and N. Herradi. Memo graph: An ontology visualization tool for everyone. *Procedia Computer Science*, 96, 2016.

[37] K. Go and J. Carroll. The blind men and the elephant: Views of scenario-based system design. *Interactions*, 11, 01 2004.

[38] A. Gomez-Perez and O. Corcho. Ontology languages for the semantic web. *IEEE Intelligent Systems*, 17(1), 2002.

[39] N. Guarino, D. Oberle, and S. Staab. *What Is an Ontology?* Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[40] H.-J. Happel and S. Seedorf. Applications of ontologies in software engineering. 2006.

[41] F. Heidecker, J. Breitenstein, K. Rösch, J. Löhdefink, M. Bieshaar, C. Stiller, T. Fingscheidt, and B. Sick. An application-driven conceptualization of corner cases for perception in highly automated driving. 2021.

[42] I. Horrocks. Owl: A description logic based ontology language. In P. van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005*, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[43] L. Huang, H. Liang, B. Yu, B. Li, and H. Zhu. Ontology-based driving scene modeling, situation assessment and decision making for autonomous vehicles. In *4th Asia-Pacific Conference on Intelligent Robot Systems (ACIRS)*, 2019.

[44] B. Hummel. Description logic for scene understanding at the example of urban road intersections. 2010.

[45] M. Hülsen, J. M. Zöllner, and C. Weiss. Traffic intersection situation description ontology for advanced driver assistance. In *IEEE Intelligent Vehicles Symposium (IV)*, 2011.

[46] T. Ishida. Q: a scenario description language for interactive agents. *Computer*, 35(11), 2002.

[47] N. Koch. Scenario-Based Validation for Autonomous Driving: Matching and Integration of Ontology-Based Scenario Modelling Approaches, 2021.

[48] J. Kocić, N. Jovičić, and V. Drndarević. Sensors and sensor fusion in autonomous vehicles. In *26th Telecommunications Forum (TELFOR)*, 2018.

[49] T. Menzel, G. Bagschik, L. Isensee, A. Schomburg, and M. Maurer. From functional to logical scenarios: Detailing a keyword-based scenario description for execution in a simulation environment. 2019.

[50] G. Ng. Open vs closed world, rules vs queries: Use cases from industry. 2005.

[51] N. F. Noy and D. L. McGuinness. Ontology Development 101: A Guide to Creating Your First Ontology. `https://protege.stanford.edu/publications/ontology_development/ontology101-noy-mcguinness.html`. Accessed: 2022-02-28.

[52] D. H. Paulheim. Vorlesung Semantic Web. `https://www.ke.tu-darmstadt.de/lehre/archiv/ws-12-13/semantic-web/slides/13_Regeln.pdf`. Accessed: 2022-02-28.

[53] C. Pilz, G. Steinbauer, M. Schratter, and D. Watzenig. Development of a scenario simulation platform to support autonomous driving verification. In *IEEE International Conference on Connected Vehicles and Expo (ICCVE)*, 2019.

[54] J. Ploeg, E. de Gelder, M. Slavík, E. Querner, T. Webster, and N. de Boer. Scenario-based safety assessment framework for automated vehicles, 2021. Accessed: 2022-02-28.

[55] D. L. Poole and A. K. Mackworth. *Artificial Intelligence: Foundations of Computational Agents*. Cambridge University Press, USA, 2nd edition, 2017.

[56] A. Prakash, A. Behl, E. Ohn-Bar, K. Chitta, and A. Geiger. Exploring data aggregation in policy learning for vision-based urban autonomous driving. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[57] R. Queiroz, T. Berger, and K. Czarnecki. Geoscenario: An open dsl for autonomous driving scenario representation. In *IEEE Intelligent Vehicles Symposium (IV)*, 2019.

[58] M. B. Rosson and J. Carroll. Usability engineering: Scenario-based development of human-computer interaction. *Inf. Res.*, 8, 2003.

[59] C. Rösener, F. Fahrenkrog, A. Uhlig, and L. Eckstein. A scenario-based assessment approach for automated driving by using time series classification of human-driving behaviour. 2016.

[60] P. Schmitt. Formale Systeme Script. `https://formal.kastel.kit.edu/teaching/FormSysWS1415/skriptum.pdf`. Accessed: 2022-02-28.

[61] H.-P. Schoener and J. Mazzega. Introduction to pegasus. 2018.

[62] S. Shalev-Shwartz, S. Shammah, and A. Shashua. On a formal model of safe and scalable self-driving cars. 2017.

[63] E. Sirin, B. Parsia, B. Grau, A. Kalyanpur, and Y. Katz. Pellet: a practical owl-dl reasoner. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5, 2007.

[64] D. B. Stefani Guneshka. Ontology-based corner case scenario simulation for autonomous driving. `https://github.com/daniel-bogdoll/corner_case_ontology`. Accessed: 2022-02-28.

[65] J. Stellet, M. Woehrle, T. Brade, A. Poddey, and W. Branz. Validation of automated driving - a structured analysis and survey of approaches. 2020.

[66] J.-A. Termöhlen, A. Bär, D. Lipinski, and T. Fingscheidt. Towards corner case detection for autonomous driving. 2019.

[67] S. Ulbrich, T. Menzel, A. Reschka, F. Schuldt, and M. Maurer. Defining and substantiating the terms scene, situation, and scenario for automated driving. In *IEEE 18th International Conference on Intelligent Transportation Systems*, 2015.

[68] S. University. Satisfiability. `http://intrologic.stanford.edu/extras/satisfiability.html`. Accessed: 2022-02-28.

[69] Q. Wang and X. Yu. Reasoning over owl/swrl ontologies under cwa and una for industrial applications. 2011.

[70] L. Zhao, R. ICHISE, Z. Liu, S. MITA, and Y. Sasaki. Ontology-based driving decision making: A feasibility study at uncontrolled intersections. *IEICE Transactions on Information and Systems*, E100.D, 2017.

[71] G. L. Zúñiga. Ontology: Its transformation from philosophy to information systems. In *Proceedings of the International Conference on Formal Ontology in Information Systems - Volume 2001*, FOIS '01, New York, NY, USA, 2001. Association for Computing Machinery.