Passau University

Master Thesis

Declarative Test Case Generation for Autonomous Cars

Author:
Johannes Müller
Matrikelnummer 67232

Supervisor:
Prof. Dr.-Ing. Gordon Fraser

Second Supervisor:
Prof. Dr. Marco Joachim Kuhrmann

Advisor:
Alessio Gambi, Ph.D.

Monday 17th February, 2020

Contents

Abstract

Autonomous cars are no vision of the future any more, with car manufacturers planning on releasing them in the near future. With software controlling vehicles in traffic, human live is at risk, when this software malfunctions. There is an infinite amount of traffic scenarios this software needs to be able to handle without failure, hence it is necessary to cover as much different scenarios as possible. Computer simulations provide a fast method to execute test cases, but generating test configuration for virtual environments still remains a costly and time consuming process. In this work I propose a method that tackles this problem by automatically generating test cases based on a received specification. For this task a model for describing test cases in form of finite state machines is introduced, that allows splitting the generation process into small steps. In each step values for properties of the test case model are picked constrained by the received specification and already present values until every property has a value assigned. The test case generated this way is then used as a starting point for local search algorithms to find test cases that reach goals in terms of trajectory, velocity and timing. The focus of the generation process lies on the generation of a test case that's values are as close as possible to the received specification. For the evaluation of the proposed method for automated test case generation a prototype system was implemented, that can generate three different accident types. It is shown, that the system can generate test cases fast and that the scenarios are conform to the received specification in most cases. This method allows testers to focus on **what** they want to test by providing a test case specification rather than on how to set up trajectories feasibly and enable the correct timing between vehicles.

1 Introduction and Motivation

Autonomous cars or self driving cars are no vision of the future any more. In the german city "Bad Birnbach" a self driving electric bus owned by the "Deutsche Bahn" company is already used to transport passengers from the city centre to thermal springs and the railway station[6]. The American company "Waymo" is using autonomous cars as taxis in a suburb of Phoenix in the US state Arizona [2]. A similar program from "Daimler" is in place in San Jose, where autonomous cars drive people from the western city district to downtown [2]. As shown autonomous cars already find application to a small degree in everyday traffic and car manufacturers are investing in research and development and plan to release some sort of self driving cars in the near future [20].

With self driving cars software is expected to be able to handle road traffic without failures, but new software always comes with bugs and malfunctions. The impact of these malfunctions ranges from being annoying and time consuming, for example when a program crashes and needs to be restarted, to being very costly, like the Ariane 5 accident in which the rocket exploded because old software was reused which was not customized for the faster engines of the new Ariane model. This explosion costed 500 million US dollar and was caused because a too large number was converted into a 16 bit integer which lead to incorrect data [8]. While the Ariane 5 accident was very costly, in road traffic human live is at stake. Therefore, it is absolutely necessary, that autonomous cars can handle every traffic situation and do not malfunction, in order to preserve human health.

While it is clear that it is a necessity that autonomous cars can handle traffic scenarios, especially critical ones, software bugs are present with the introduction of software to road traffic. Self driving cars can be tested on public roads in California, but require companies to report every "disengagement", meaning they have to report, when it was required for the human safety driver to take away control from the software. "Between September 2014 and November 2015, Google's autonomous vehicles in California experienced 272 failures and would have crashed at least 13 times if their human test drivers had not intervened" [10]. A group of researchers from the Max-Planck Institute for Intelligent Systems presented on the ICCV in Seoul a colour scheme that causes the image processing of self driving cars to fail [7] and therefore, prevent the car from operating correctly. This can happen even if the colour blot only takes less than one percent of the image [7]. A fatal crash in which a pedestrian was killed, involving a self driving car, was reported in Arizona [21]. The person was crossing the street with her bicycle in front of the car. She was visible, but the car did not seem to recognize her [22]. These examples of malfunctioning self driving cars show clearly that extensive software testing on autonomous cars is necessary and that they are not reliable, yet.

Testing autonomous cars is not an easy task. There are different approaches to this. A common practice is running tests in the real world which means either in real traffic or on test tracks specifically built for testing purposes [36]. A problem with this approach is that running such a test case is very costly. Therefore, only a small fraction of all possible test cases can be run in the real world. In addition to that, running tests in real traffic brings high risks with it. Hence test cases that do not lead to critical situations are executed. While real world tests should not be replaced, a meaningful addition to them are virtual tests. In virtual tests the test scenario and the inputs for the various sensors of the autonomous car are simulated and therefore, it is possible to cover a big variety of test scenarios in a short amount of time. This not only tackles

the monetization problem of real world tests, but also enables the execution of test cases that lead to critical events. In 2016 in Germany 3214 people were killed in car accidents [18]. Because about 90% of traffic accidents are results of human failure an increase in road safety is expected by handing over control to an AI [19]. In order to make this increase in safety possible the software in use has to be reliable and need to handle traffic situations humans could not handle. Alessio Gambi et al. [33] propose a system (AC3R) for creating simulations of car crashes from police reports. This approach picks up on the idea of testing self driving cars in critical scenarios, where humans failed. This thesis presents a different approach for the automatic generation of critical test cases. AC3R relies on police reports for generating test cases. The system presented in this thesis does not rely on any input, but generates test cases completing received partially specified input. The only information needed is the kind of critical event such as "Front to Rear" crash or "Lane Departure". Using this information the system generates test cases that are conform to the provided event. That means that the accident happens as declared, if no intervention happens. I present a model for critical test cases that contains every property necessary to execute the test case. The user has the option to define values for these test case properties beforehand as an input for the system. The system generates the test case regarding these inputs with the priority on not changing them. This ensures that the tester can declare parts of the finished test case beforehand. The finished test case provides a trajectory for every vehicle and trajectories of roads on which the cars drive. The system also ensures that cars that are supposed to crash at a certain location do so. A set of preconditions is generated that ensures that cars have reached locations and velocities that are mandatory for the test case to be valid. When the preconditions are met the AI can take over control of the Ego Car (EC) and the actual test execution can start. Hence the system under test controls a vehicle that is about to be involved into an accident if it does not intervene.

1.1 Problem Statement

Testing autonomous cars is an important task for guaranteeing safety in road traffic. A difficulty is to find critical test cases. There is an infinite number of scenarios an autonomous car can be put in, but not all of them stress the car adequately and expose a failure. So obviously a very large amount of test scenarios and test case configurations needs to be considered when testing autonomous cars. It is impossible to test all these scenarios in real world tests due to time and monetary limitations. In addition to that the manual creation of test cases is a time consuming process.

This thesis tackles the problem of time consuming test case generation by presenting an automated approach of generating test cases that can be executed in a simulation environment by using given information about the desired scenario. When generating test cases time consuming tasks are finding trajectories that cars can follow. A car can not necessarily follow any trajectory with a certain velocity. Often times curves are too sharp and the car drifts off its trajectory. In addition to that the timing of cars needs to be adjusted so that they do actually crash if their trajectories are intersecting. This is a non trivial task because slight deviations in timing can lead to very different outcomes. In the domain of car accidents this can be a deciding factor whether cars crash or pass each other. The stochastic nature of physical simulations makes these adjustments on timing within a test case specification difficult. The system i have generated for this thesis tackles both of these challenges and provides a fast automated method for solving them. In order to find trajectories a car can follow and adjust timing to guarantee the crash,

local search algorithms are used. Local search algorithms search for solutions close to the current one which achieve a better grade regarding the optimization goals.

Hence the approach presented in this thesis allows the user to not worry about feasibility, trajectory planning and timing when generating test cases that comply to the scenario he wants to test. The tester can focus on **what** he wants to test by declaring that and handing it as an input to the system.

2 Background and Related Work

2.1 Declarative Programming

Declarative programming is a programming method that focuses on what should be computed rather on how it is computed [39]. Programming languages that use that approach are functional languages like Haskell or logic languages like Prolog. On the other side are imperative programming languages like Java or C that focus on how a problem is solved rather than describing the problem.

This thesis is titled "**Declarative** Test Case Generation for Autonomous Cars" because the system that is developed focuses on what the test case and test scenario should be like and what it should not be. The approach i propose in this thesis allows tester to focus on **what** they want to test e.g., a front to rear crash where one car shifts its lane and lands in front of another car, rather than on how this scenario is achieved in terms of scenario set up like timing or trajectory set up.

2.2 Finite State Machine

Formally an FSM can be defined as a 5-tuple <Q, Σ, σ, q_0, F>[9].

- Q is the set of states
- Σ is the input alphabet
- σ is a set of transition functions
- q_0 is the starting state
- F is a set of end states

A finite state machine consists of states and transitions. It has one start state and can have several end states. It receives inputs from the input alphabet and based on these inputs it transitions to other states if a transition function from the current state to another state with the given input is defined. The FSM determines if an end state is reached.

A FSM can be extended with output values and guard conditions, resulting in an *extended finite state machine* (EFSM). Here upon transitioning to another state it is verified that guard conditions are fulfilled [32].

State machines are part of the specification of a test case and are used to model the behaviour of dynamic objects such as moving cars during test execution. It is used to describe the test case on an abstract level and can be used to verify the correct sequencing during test execution.

2.3 Reactive Test Case

Reactive test cases are test cases that can react to a predefined set of properties of the system under test (SUT) [25]. These test cases vary the input for the SUT dependent on the observed state during test execution. In the domain of testing automotive cars reactive test cases can be used to control the movement of none ego car characters (NECs), which are traffic participants that are not controlled by the AI under test. For example a NEC car can be moved out of a parking spot when the ego car (EC) reaches a certain location and/or speed.

Preconditions in general describe all conditions, that must be met before test execution for the test execution to be valid [16]. In this work the test preconditions are defined upon the environment where the test execution takes place. That means correct placement of roads, pedestrians, cars and trajectories. Observable properties e.g.,current location or current velocity, of vehicles are preconditions as well. They are checked before the AI takes over control, this guarantees that the EC is manoeuvred into the desired driving situation that is tested.

The test oracles determine whether the test case passed or failed. A test oracle defines the results of a test, compares the actual outputs to the defined results and evaluates if the outputs were sufficient enough to pass the test case [27]. In the context of testing autonomous cars quite general test oracles would be that the car does not get damaged and does not cause damage and that the car reaches a certain destination point during a given time interval.

The test cases generated by this thesis are not reactive in the form that test executions play out differently dependent on actions taken by the system under test. NECs start their movement dependent on the position of the EC. This timing is critical for vehicles to meet their preconditions. That means that vehicles react to the position of the EC before the AI that is tested takes over control of the vehicle. This phase before preconditions are met is called test set up. During test set up all NECs react to the EC, meaning the EC functions as a reference point according to which other cars start their movement. This will be explained in greater detail in the following sections.

2.4 Search Based Testing

When testing a system like automotive cars the set of possible test cases (simulation scenarios and their configurations) is infinite. Therefore, it is important to find test cases that stress the system under test adequately. This is where search based testing comes into play. When using search based testing, objectives, that the test case should fulfil are defined and fitness functions, that evaluate how close the test case is to reach these objectives need to be defined. With the use of these fitness functions it is possible to search in the set of all possible test cases for test cases, that are closer to the desired goal by optimizing the results of the functions.

Search based testing is commonly used in combination with genetic algorithms. Genetic algorithms are used to solve optimization problems by evaluating the fitness of every entity in an initial population, which can be generated randomly. The fittest entities are then used to generate a new population using crossover and mutation. Crossover means, that two parent entities are combined to a child entity. Using mutation, one or more variables are changed to ensure

diversity and generate different children for the new population. This is an iterative process, which is repeated until a satisfying fitness level is reached.

This thesis will make use of search based testing in form of local search without a genetic algorithm. It is used for the test case generation itself by searching for an environment and trajectory set up that allows the EC to satisfy the test preconditions. A start solution is used and a fitness value is assigned to it. Then a neighbour of that start solution is picked and its fitness is evaluated. If its fitness is better than the fitness of the current solution, the neighbour is picked to continue the search. This process is repeated until a sufficient fitness value is reached [13]. Genetic algorithms are not used because to apply a fitness value to a solution it needs to be simulated and the approach presented focuses on a fast way to generate test cases and executing simulations is a time consuming process. Assumptions based on the search environment are used to guide the search to improving solutions.

2.5 Bezier Curve

Bezier curves are commonly used in computer graphics to draw curves. They provide an easy way to describe a curve with only three given points. A bezier curve is defined by a start point an end point and one up to several additional control points. The start and end point lie directly on the curve. Control points influence the shape of the curve but do not directly lie on the curve itself. The Bezier Curve is a curve that is represented by the following polynomial:

$$B(t) = \sum_{i=0}^{n} \binom{n}{i} \cdot (1-t)^{n-i} \cdot t^i \cdot p_i \text{ with } t \in [0,1] \tag{1}$$

In Equation 1, n is the number of points used to interpolate the Bezier Curve minus 1, because i starts at 0. p_i are are coordinate values (x, y) of these points. The point p_0 is the start point of the curve and p_n is the end point. Those two points are the only ones, that lie directly on the curve. The points $p_1, p_2, ..., p_{n-1}$ do not lie on the curve, but influence its shape [17], like shown in figure 1. The two Bezier Curves depicted in figure 1 have three control points. With p_0 and p_2 determining the start and end position of the curve only p_1 has influence on the shape of the curve. Bezier Curves which are determined by three points are called quadratic Bezier curves. Filling the value $n = 2$ in equation 1 results in the following equation 3, that describes Bezier curves with three control points.

$$B(t) = \binom{2}{0} \cdot (1-t)^2 \cdot p_0 + \binom{2}{1} \cdot (1-t) \cdot t \cdot p_1 + \binom{2}{2} \cdot t^2 \cdot p_2 \tag{2}$$

$$\Leftrightarrow B(t) = (1-t)^2 \cdot p_0 + 2 \cdot (1-t) \cdot t \cdot p_1 + t^2 \cdot p_2 \tag{3}$$

Quadratic Bezier Curves are used in this work to model curves in trajectories of traffic participants and cars. They provide a simple way to describe curves, because they only need one additional point to generate a curve from a straight line. It is easy to modify the shape of these curves only by moving the control point without touching start or end point. That allows for simple mutations in case the shape of a curve is infeasible for a car to follow.

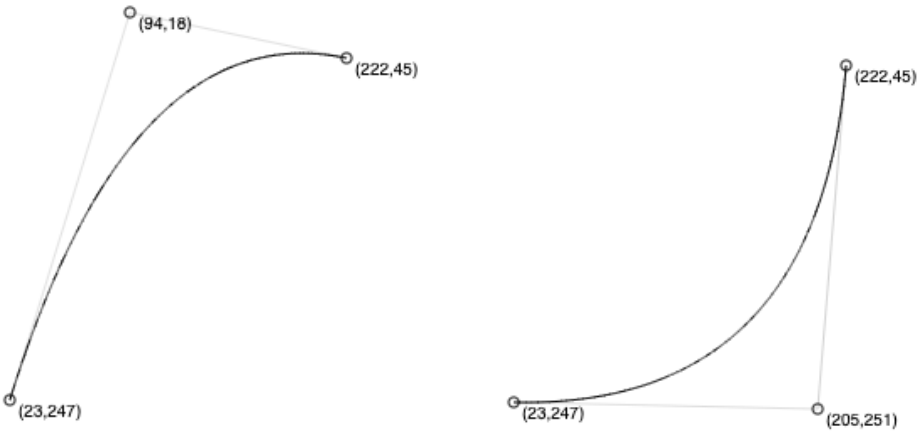

Figure 1: Example of two Bezier Curves with n = 3. p_0 and p_2 are equal for both curves. Source: [17]

2.6 Related Work

This thesis proposes a novel approach to automatically generate test cases for self-driving cars in critical driving scenarios. Autonomous cars can be tested in several different ways, like running tests on source code level, X-in-the-loop testing, simulation testing and real traffic driving tests [36]. The system proposed by this thesis focuses on simulation testing. In order to generate a scenario set up, that runs in the simulation as planned fitness functions are optimized.

Till Menzel et al. [40] show what requirements a scenario description has to fulfil during different steps of system development of the ISO 26262 standard. They differentiate between three different abstraction levels for scenario descriptions. Functional scenarios, which are the most abstract ones. They are represented in natural language and it is not needed for a machine to understand them. Logic scenarios contain every parameter needed to define the scenario and value ranges for these parameters. The least abstract description is the concrete scenario. It describes the scenario with concrete values for every parameter and can be used for test case generation. This thesis uses the level of abstraction of logic scenarios for describing its driving scenarios.

Lionel Briand et al. [31] discuss how systems that continuously interact with the environment should be tested. It proposes that the level of abstraction is raised and that models of behaviours and properties of systems should be tested instead of operational systems/implemented systems. By simulating whole driving scenarios the level of abstraction is raised from actual source code tests to tests, that verify the behaviour of autonomous cars in critical driving situations.

A system for generating simulations of real car crashes from police reports using databases like the NHTSA crash viewer [14] is proposed in [33] (AC3R) . It uses natural language processing and part of speech tagging to map the information from the description to a domain-specific ontology. After the needed information is extracted the intercepting trajectories of the cars are calculated and the simulation is implemented in BeamNG [28]. The result of AC3R are similar to the results of the system proposed in this thesis, but differs in terms of input. AC3R uses police

reports and reconstructs the described driving situations. Those are situations where human drivers failed due to various reasons, hence critical scenarios are the result. The system proposed in this thesis allows for a user defined input and description of the critical situation.

With the paper "Automatically Testing Self-Driving Cars with Search-Based Procedural Content Generation" [34] Alessio Gambi et al. propose a system for automatic test case generation with the target of testing lane keeping capabilities of autonomous cars (AsFault). A road network is generated by placing several roads, which are procedurally generated by stitching road segments together, on a big map. It is checked, whether the generated road network represents a valid road system. Then search based testing is used to find road networks, that expose failures in lane keeping, with a fitness function that rewards road networks, that made the system under test drive further away from the middle of their lane, above other road networks. AsFault and the system described in this thesis, both automatically generate test cases for a simulation environment. While this thesis focuses on the generation of test cases, that lead to a failure without intervention independent of an AI, AsFault focuses on exposing failures of specific systems under test.

Galen E. Mullins et al. present an approach for the "Automated Generation of Diverse and Challenging Scenarios for Test and Evaluation of Autonomous Vehicles" [41]. Their approach categorizes the actions taken by an autonomous vehicle to fulfil a mission into disjunct performance modes. Then a search algorithm is used to find scenario configurations, from which the results are as close as possible to the boundaries of the performance modes. This results in test cases that produce diverse behaviour of the system under test and can give insight into what might influence the decision making of the autonomous vehicle.

Florian Hauer et al. try to tackle the problem of "methodological challenge of creating suitable fitness functions" [35]. They provide templates for fitness functions that "ensure correct positioning of scenario objects in space, yield a suitable ordering of maneuvers in time, and enable the search for scenarios in which the system leaves its safe operating envelope" [35]. The templates they present for correct positioning of cars and for the correct timing of events measure the distance from the measured data to the desired outcome. With 0 being the result if the actions and placements were done correctly. This thesis applies fitness functions for trajectory calculation and timing of vehicles, that use a similar approach in calculation.

Aitor Arrieta et al. [25] uses search based testing to find optimal reactive test cases for cyber-physical-systems (CPS). Fitness functions are applied for guiding the search towards requirements coverage, test case similarity, and test execution time. Using these fitness functions, one crossover operator and three mutation operators the NSGA-II algorithm is applied, which is a multi objective search algorithm. In another paper [26] he applies a weight based search algorithm in order to prioritize the execution order of test cases in a test suite. Weight based search can be used to convert a multi objective problem into a single objective one. Objectives used for prioritizing test cases are test execution time and fault detection capability.

Andrea Arcuri et al. [24] deals with black box testing of real time embedded systems. The main focus lies on the test case selection comparing random testing, advanced random testing and search based testing in combination with a genetic algorithm. Rakesh Kumar et al. [37] compares search based testing and random testing concerning automatic test case generation, when trying to test boundary values as input parameters. Findings are that search based testing outperforms random testing.

Joel Lehman et al. [38] and Mohammed Boussaa et al. [29] [30] use search based testing to search for novelty. It basically means that it is not tried optimize a fitness function regarding a certain objective, but to optimize a novelty function to search for solutions that are different from each other, in order to cover a large variety in the search space. Novelty is measured as the distance between test cases [29] [30]. This distance is calculated using the Manhatten Distance between all input parameters of all methods under test. Such a method can be used to optimize generation of test case that are different from each other for the same input, when applied to our system. How this can be incorporated will be further discussed in the future work section.

Matthias Althoff and Sebastian Lutz [23] propose an approach to make traffic situations more critical in terms of collision avoidance, by measuring the criticality using the drivable area the EC has for its maneuvers and shrinking that area. This is achieved by changing the initial placement of traffic participants or changing velocities of them and the EC. By reducing the drivable area the solution space for correct behaviour gets smaller and an immediate correct behaviour of the EC is necessary to avoid a collision. This method can be used to try to create scenarios that are more critical than what is currently created by the system proposed in this thesis, when implemented in our system. Further discussion about that in the future work section.

3 Method

In this thesis i propose a novel approach to automatically generate critical driving scenarios suited for execution in a simulation environment for testing autonomous cars. Critical means, that the driving scenario generated would lead to an accident (critical event) if no intervention happens. Hence, the Ai under test will be handed over control of the EC, when the driving scenario is already set up and the critical event is about to happen. I developed a system that automatically generates test cases based on the critical event that is supposed to be tested. Generating the test case means that a value is assigned to every test case property. Test case properties are further described in section 3.1. We focus on driving scenarios, that lead to a critical event, like crashes or road departures. Figure 2 shows an example trajectory setup for a crash at an intersection. This is an abstract representation of an actual test case set up. Roads are missing in this figure. They are placed beneath the car's trajectories and are adjusted dependent on the lane the cars drive in. The blue lines symbolize the test preconditions, both cars have to pass those lines with a certain velocity for the preconditions to be fulfilled. The straight lines up to the precondition lines represent the set up phase. During this phase the system under test has no control over the EC yet. The set up phase is responsible to take all vehicles to the needed location and lets them accelerate to the needed velocity. Behind the precondition lines the execution phase starts. During this phase the actual test case execution takes place. The system adjusts the timing of both cars so that they do actually crash at the intersection of their trajectories. It is also guaranteed that the curves are not too sharp for them to drive. The yellow rectangle behind the trajectory intersection describes the success area. This is the area the EC has to reach in order to successfully complete the test case. In most cases it is placed on a straight line behind the location where the critical event is supposed to happen. This crash at an intersection will function as a working example during the remaining thesis. Concepts are exemplified using this set up.

This thesis also introduces a finite state machine (FSM) model for describing vehicle behaviour during test execution. Sections of trajectories of vehicles are split in states of FSMs, this means that a state is used to describe the behaviour of the vehicle during one section of its trajectory. When the vehicle leaves this section the FSM transitions to the next state (further described in section 3.2). It exists one FSM for each vehicle present in the test case. We use this FSM model to categorize the trajectory of vehicles, this enables us to treat sections of their trajectory differently during the generation of missing values dependent on how the vehicle is supposed to behave e.g., accelerate, travel or crash.

Scenarios are specified by the tester in a declarative manner, which means that the input provided by the user describes what the scenario should be rather than how the scenario is created. To achieve this, the system needs two input files for generation purposes. An input file with the specification of the desired test scenario and a configuration file that contains information about test case properties and their boundaries that are not defined by the specifications. The input file contains the type of the critical event as a mandatory value. For example the scenario in figure 2 can be generated by solely receiving *TwoRoadAngle* as an input. *TwoRoadAngle* is in this case the type of critical event. It is a angled crash involving two roads (hence it happens at an intersection). In addition to that the tester can already assign values to test case properties, if he needs those values to be present in the finished test case. The configuration file contains value ranges for test case properties that did not get a value assigned in the input file. The tester can use the configuration file to guide the test case generation into a direction he wants to test

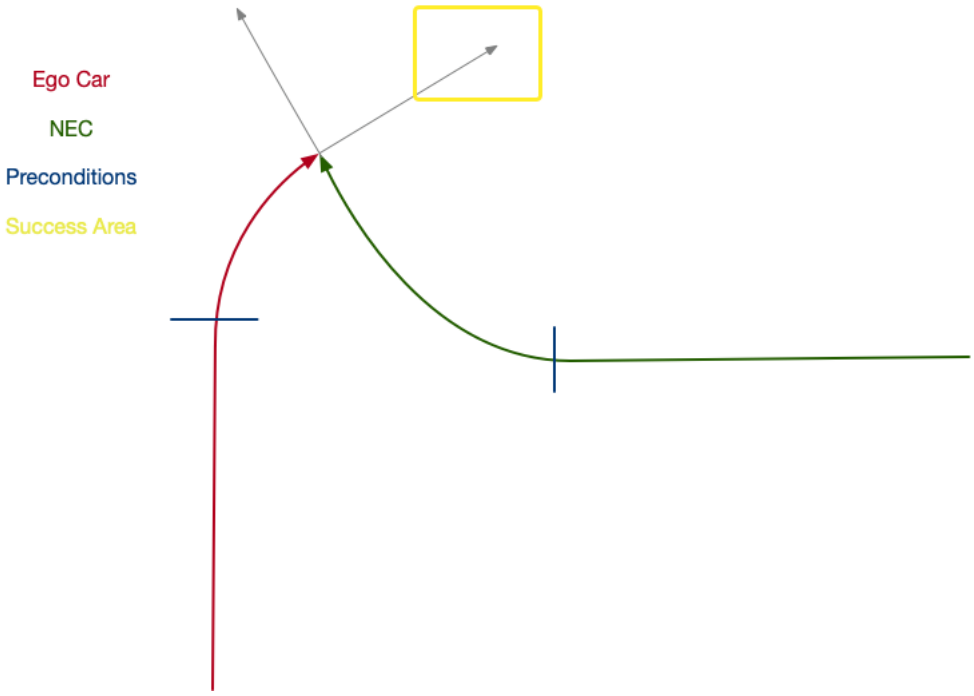

Figure 2: Example trajectory setup of a two road crash

e.g., only right curves are supposed to be generated. These input files are described in section 3.3.

In figure 3 the approach on generating the test case is depicted. The general approach is going top down from abstract to concrete, which means that based on the user input a partially specified test case is generated that lacks on property values dependent on the accuracy of the input. The system then generates step by step missing properties and adds them to the test case, making it concrete. It does that by randomly picking values for properties from the allowed value ranges that are determined by satisfying constraints, which are introduced by the critical event, for already set values. For instance constraints applying for the crash in our TwoRoadAngle example are the following:

- EC.position = NEC.position (positions need to be equal at the crash location)
- EC is on road (EC trajectory is only valid when placed on a road)
- NEC is on road (NEC trajectory is only valid when placed on a road)

If no value is yet assigned to any test case property, our system randomly sets them in a predefined order. Each time a value is set, the constraints listed above are satisfied, this leads to a limitation of possible values for the remaining properties that have no value assigned. For instance, if the waypoint at the crash location gets set for the EC, the position of the waypoint for the NEC is limited to the same location because of the position equal constraint. The possible waypoints

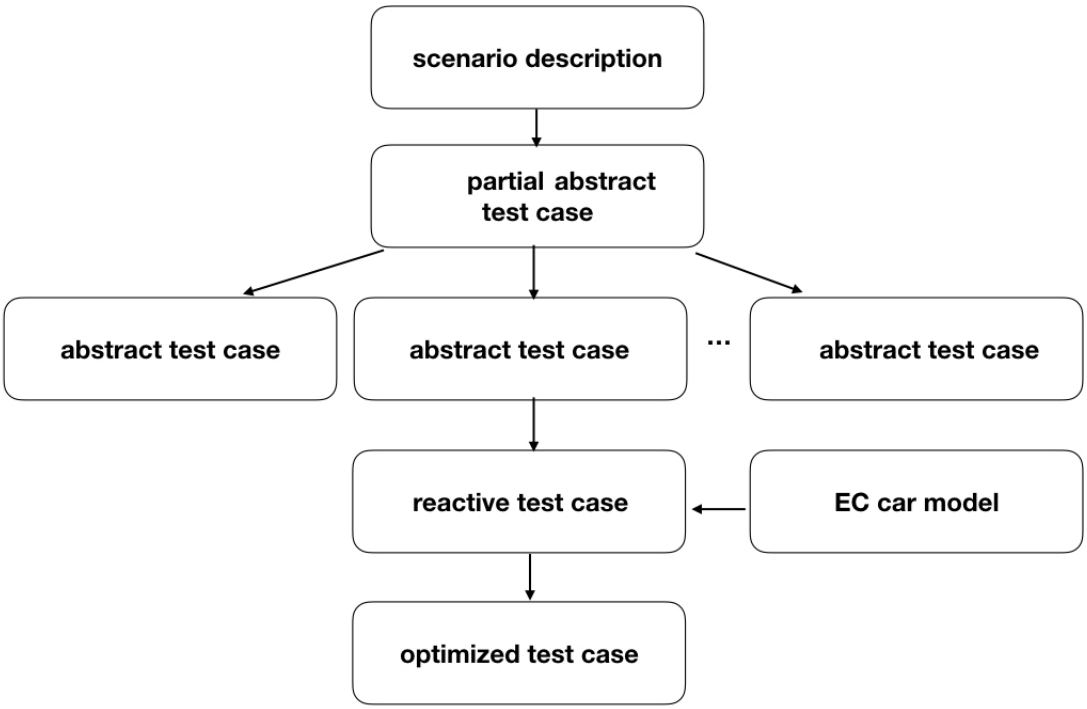

Figure 3: Generation steps of the test case

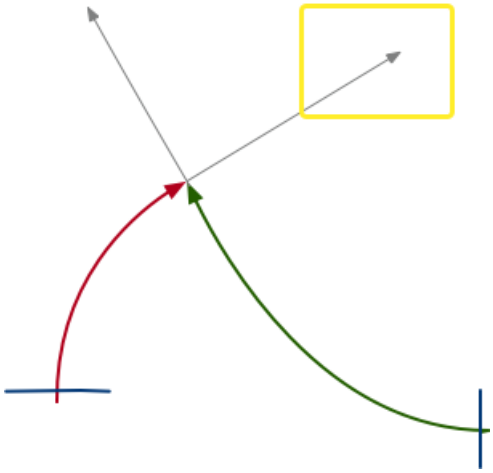

Figure 4: Example set up of an abstract test case

for the road trajectory of the EC is limited as well, because the EC needs to drive on the road. The same applies for the road waypoint of the NEC respectively.

Hence more inputs provided by the tester constrain possible other values more than less inputs. Therefore, the system can generate a larger number of diverse test cases with a less specified input than with a more specified input. A fully specified input always results in the same test case. A by-product of this generation is the abstract test case. The abstract test case already contains all properties needed for describing the preconditions of the test case, but has no information on how to enable cars to meet these preconditions. It also contains every information about the accident, that is needed. Figure 4 depicts a representation of the abstract test case corresponding to our example in figure 2. The straight trajectory leading up to the precondition lines is still missing. As mentioned before this straight trajectory is the set up phase. It is needed to accelerate and drive vehicles into the correct position for test execution. Therefore, the set up phase contains information about how vehicles meet their preconditions. The set up phase is what separates the abstract test case from the concrete test case. The whole generation process is further explained in section 4.

The concrete test case results after all missing values have been generated by our system but is not necessarily the finished test case, because the timing for the crash (cars miss each other), the trajectory set up (curve might be too sharp) or the reached velocities (cars are too slow) can be insufficient. To guarantee that neither is the case we execute the scenario in a simulation environment and track position and damage data of all vehicles. In case the simulation can not be executed as specified in the test case, we optimize our test case using distinct local search algorithms with fitness functions for timing, trajectory and velocity respectively (see section 4.5). After that our system generates an output file, with which the test case can be executed in a simulation environment.

3.1 Test Case Properties

A test case consists of test oracles, test preconditions and information about test execution. First the needed properties for test execution are described in the following paragraphs. These properties are listed below:

- type of critical event
- road properties
- environment properties
- traffic participants

In the following subsections every property is described in detail.

3.1.1 Type of critical event

The type of the critical event provides general information about the tested scenario. Possible values for that are defined by the "Guidline Model Minimum Uniform Crash Criteria" (MMUCC). MMUCC delivers a minimum set of variables that describe a motor vehicle crash and is the standard used by the NHTSA. It was developed to encourage greater uniformity in the data that states collect in their State crash data system [1]. Examples of the critical event are "Front to Rear", "Front to Front" or "Lane departure", with which the car crashes can be identified and categorized. This type of critical event basically is what is called a "logical scenario" by Till Menzel et al. [40]. The type of crash provides an initial set of conditions about the movement of the vehicle involved in the crash. The type of the critical event on its own is sufficient for the system to generate a test case, because it provides enough information to pick suitable values for the other properties. In our example above in figure 2 the type of critical event can be identified with what is called "Angle" in the MMUCC.

3.1.2 Road properties

Road properties generated by our system are the road width, the number of lanes and the road trajectory. The road width is the width of the whole road and is evenly distributed amongst all lanes of that road. A road with width six meters and two lanes has a lane width of three meters as a result. The road trajectory is a list of waypoints that describe the course of the road.

3.1.3 Environment properties

Environment properties are lightning and weather conditions. Possible values are as well specified in the MMUCC. The daytime is a result of the lightning conditions.

3.1.4 Traffic Participant

Traffic participants are the dynamic objects in the test case. This work focuses on vehicles as traffic participants, but in theory the model is also applicable to unprotected road users e.g., cyclists, pedestrians or animals. It is differed between two different kinds of Traffic Participants. The Ego Car (EC), which is the car, that is supposed to hand over control to an AI, eventually, and Non Ego Cars (NECs). There is exactly one EC in every test case and any number of NECs (current implementations support one NEC). Traffic participants have the following properties:

- maximum velocity
- acceleration power
- trajectory
- direction

Maximum velocity and acceleration power are properties, that are determined by the car model used and are retrieved as an input for the scenario generation. They are needed for the generation of the set up phase of each traffic participant. For example the length of the acceleration phase has to differ dependent on the acceleration power of a vehicle.

The trajectory is a list of nodes. These nodes contain a coordinate and a velocity and describe the movement of the traffic participant during test execution. When the traffic participant is supposed to travel on a curve between two nodes rather than on a straight line, a bezier curve is used (see section 2.5). For that purpose an additional coordinate is added, that represents the bezier control point. Therfore, a curved trajectory is represented by two nodes and an additional coordinate. Coordinates are represented by a tuple (X, Y). Hence the trajectories of cars are placed on a two dimensional coordinate system.

As shown in our example in figure 2 both cars approach the crash location from different directions. The EC drives from the bottom to the top and the NEC from the right side to the left side. The trajectories of the cars are placed on a common coordinate system with the Y-axis being the vertical axis and the X-axis being the horizontal axis. Hence the EC drives in Y direction and the NEC in opposite X direction, which is called RX. The opposite Y direction is called RY respectively. The driving direction of a traffic participant can either be X, Y, RX, or RY. During this thesis i refer to a *coordinate in driving direction*. By that the coordinate of the coordinate tuple (X,Y) that has the biggest value difference between two waypoints of the trajectory of a car is meant. For instance Y is the coordinate in driving direction when the car drives in Y or RY direction. *Orthogonal driving direction* denotes the direction, that is not driving direction. It can have values X and Y. X is the orthogonal driving direction to Y and RY and Y accordingly to X and RX.

3.1.5 Test Oracles

Test oracles are a critical part of a test case and are used to decide whether the system under test has passed or failed the test. This system uses three test oracles, that can lead to a failure.

- the ego car got damaged
- the ego car has left the road
- the simulation timed out

The test execution is ended, if one of the following two conditions is fulfilled. Either a certain time frame has passed or the EC has reached its last waypoint of the trajectory. If the last waypoint has been reached, after the execution has ended and the EC is not damaged and did not leave the road, the test execution is considered a success. Every other case is considered a failure.

3.1.6 Test Preconditions

Test Preconditions are conditions, that need to be met before executing a test case. They ensure, that everything is in place for test execution. Our system generates driving situations, that lead to an accident, if it is not intervened by the system under test. Hence, preconditions of these test cases are, that all traffic participants are at a certain place with a certain velocity, that will lead to the desired event, before the AI actually takes over control of the EC. So preconditions, provided by our system, of a test case are a waypoint that needs to be passed and a velocity that needs to be reached for every Traffic Participant included.

3.2 TestCase Model

The previous section described all properties of the test case. For generation purposes, some properties of traffic participants and roads are modelled within finite state machines (FSMs). These FSMs are used to verify, whether the cars followed their trajectory as planned. Every traffic participant has its own FSM, that describes its movement. They are called *Sequence Plan* in this work. The sequence plan has three different kinds of states:

- several setup states
- one execution state
- one success state

Every state describes the movement of its traffic participant between two nodes of its trajectory. Setup states are used to describe the trajectory before the critical event takes place. Hence, setup states are responsible to enable each traffic participant to fulfil the preconditions. The critical event in our example is the crash at the road interception. The execution state is the state, that is active during test execution. It leads directly to the critical event. The success state provides a trajectory for NECs behind the critical event, so they do not stop driving, when the crash location is reached and an area for the EC, that ends the simulation, when reached and is fundamental for a successful test execution. A state has the following properties:

- type: delivers a basic description of the state, that determines how it is treated during the generation process
- 2 traffic participant waypoints, between which the traffic participant drives during that state
- 2 control points, functioning as the control points for the bezier curves. One for the traffic participant and one for the road (only needed for the execution state).
- the velocity the object should have, when leaving the state
- the lane number on which the car drives during this state
- 2 road waypoints that describe the road trajectory this car drives on during that state. Those road waypoints are not necessarily the same as the traffic participant waypoints, because road waypoints mark the middle lane of the road while the traffic participant often drives on another lane.

Possible types for setup states are *Park*, *Accelerate* and *Travel*. The park state marks the initial position of the traffic participant. The accelerate state is used to accelerate the car up to the needed velocity. The travel state is the part of the trajectory a traffic participant drives on before it reaches the execution state. We need it to adjust the timing between traffic participants by shortening or elongating the travel state. A FSM is specified by the quintuple $(Q, \Sigma, \sigma, q_0, F)$. Q contains every state of the sequence plan of the traffic participant. As an input alphabet Σ every tuple of a waypoint and velocity of this cars trajectory is used. σ is the transition function, that takes Σ as an input and transitions dependent on the received input and the current state to the next state in the sequence plan. q_0 is the initial state of the FSM. In the sequence plan that state is always the park state. F is the set of final states, it includes the *success state*. State transitions generally happen, when the last waypoint of the current state is received as an input. For every other input the sequence plan remains in its current state. There are differences in the sequence plans of ECs and NECs which are explained in the following paragraphs.

EC The Ego Car (EC) is the car that will be controlled by the AI under test. So the control of the car needs to be handed over to the AI when all preconditions are fulfilled. This action can be modelled as an additional state transition from the *execution state* to an *AI control state*, that is added to the set of states Q. Additionally a *fail state* is added to Q and F, that is entered, when the test execution times out. Signals, that indicate, that all preconditions are fulfilled and that the test did time out are added to the input alphabet Σ. Transitions from the *executions state* to the *Ai control state* and from the *Ai control state* to the *success state* and *fail state* are added. Figure 5 shows an example sequence plan of the EC for our Two Road Angle event.

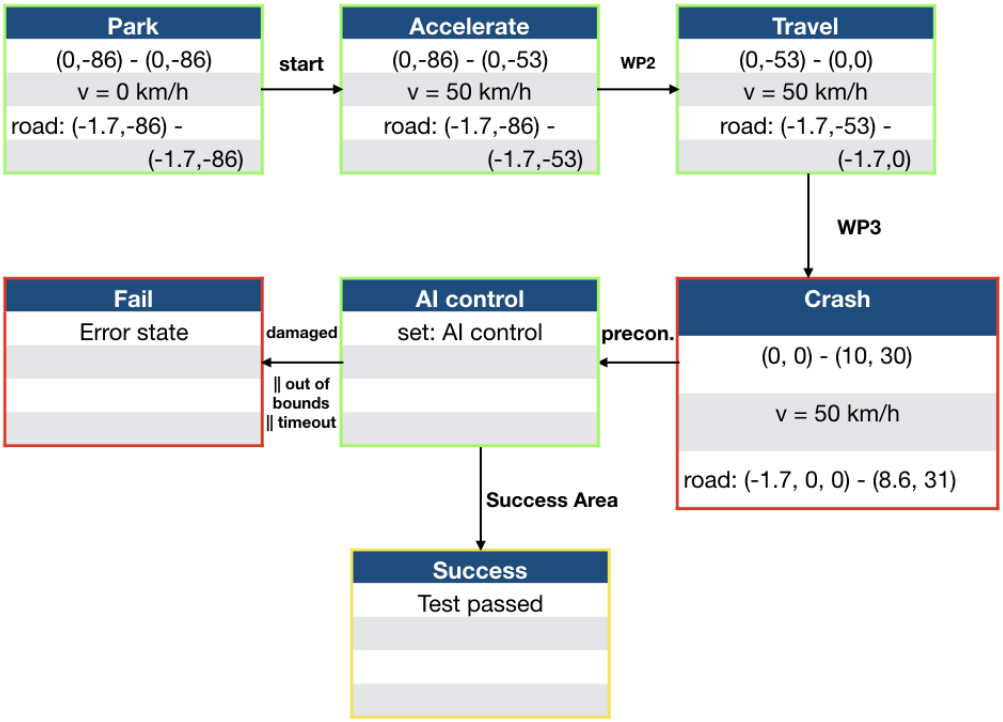

Figure 5: Example sequence plan

Park, Accelerate and *Travel* are *Setup states*. They define actions the EC has to take in order to reach the desired velocity and position. The *Crash* state, with the red border, is the execution state. This figure gives an idea on how trajectories are defined and how they are incorporated in the state machine model. The EC drives on the right lane, that is why the road waypoints are shifted to the left in relation to the traffic participant waypoints.

NEC NECs do not hand over control to a system under test, hence they have no AI control and fail state. They do have a success state, but this state is used only for describing their trajectory past the critical event and has no impact on the success of the test case. The challenge for NECs is to synchronize their sequence plan. We are dealing with separate concurrent FSMs, that need to be at a certain point of execution synchronized. An easy example for this is a crash. If the critical event is some sort of crash, all cars, that are part of that crash, need to reach the crash location simultaneously. This is enabled within the sequence plan of NECs. This is achieved by starting their movement, when they need the same amount of time to the specified location as the EC. So the EC is a reference point for every NEC in the simulation and they react to its position. Figure 6 illustrates this concept. The NEC (in green) waits until the EC (in blue) has reached a location from which it needs the same amount of time (15 seconds in the figure) to arrive at the crash destination at the intersection. To fit this into our FSM model, we need to

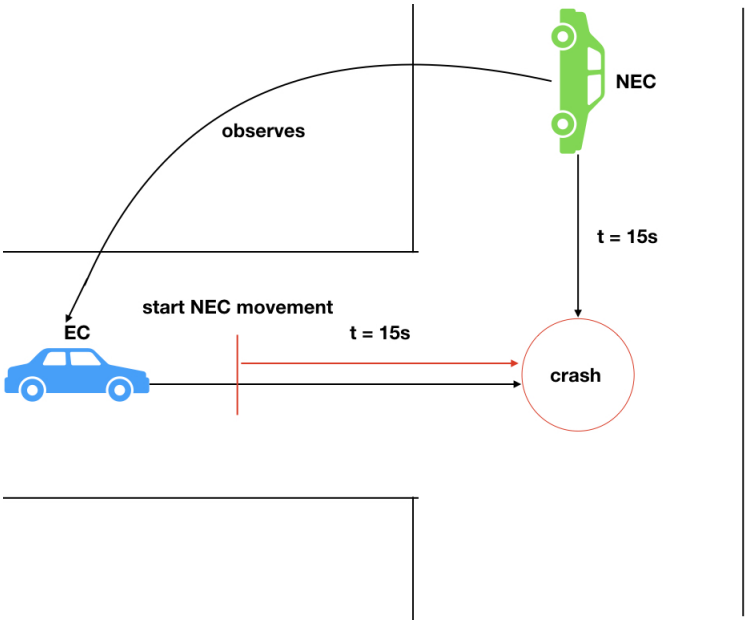

Figure 6: NEC starts movement dependent on the position of EC

add the waypoint at which the NEC should start its movement to the input alphabet and add a transition from the *Park state* to the *Accelerate* state with this waypoint as input.

Let us apply this model to our example of the two road angled crash. Figure 7 shows what parts of the trajectory is part of which state of the sequence plan. Setup states are marked by the green blocks next to the trajectory and the execution states by the thick red line. The Park state only marks the initial position. After that both cars accelerate to the demanded velocity, then they further travel on a straight line until they pass their precondition location. At this point the AI can take over control of the EC (red trajectory). If the AI does not intervene both cars crash at the location where their trajectories intersect.

3.2.1 Test Oracles and Preconditions

Besides the traffic participant properties, that are described in the previous section, the test preconditions and partly test oracles are present in the sequence plan model as well. Test preconditions are present in the first waypoint of the execution state and the velocity of the execution state of each traffic participant. That means before the EC is allowed to hand over control to the system under test and the actual test execution can start, every traffic participant present in the test case has to have reached its execution state. Test oracles present in the sequence plan are transitions from the *AI control state* to either the *Fail state* or *Success state*. Test oracles missing in the sequence plan are *the car does not leave the road* and *the car does not get damaged*. These missing oracles are evaluated after test execution, meaning they are no criteria to stop the execution of the test case, but can let it fail when being evaluated. Criteria to stop the test execution are only a time out or that the EC has reached its success area. The reasoning behind this is, that the EC can reach the success area in some cases even though it

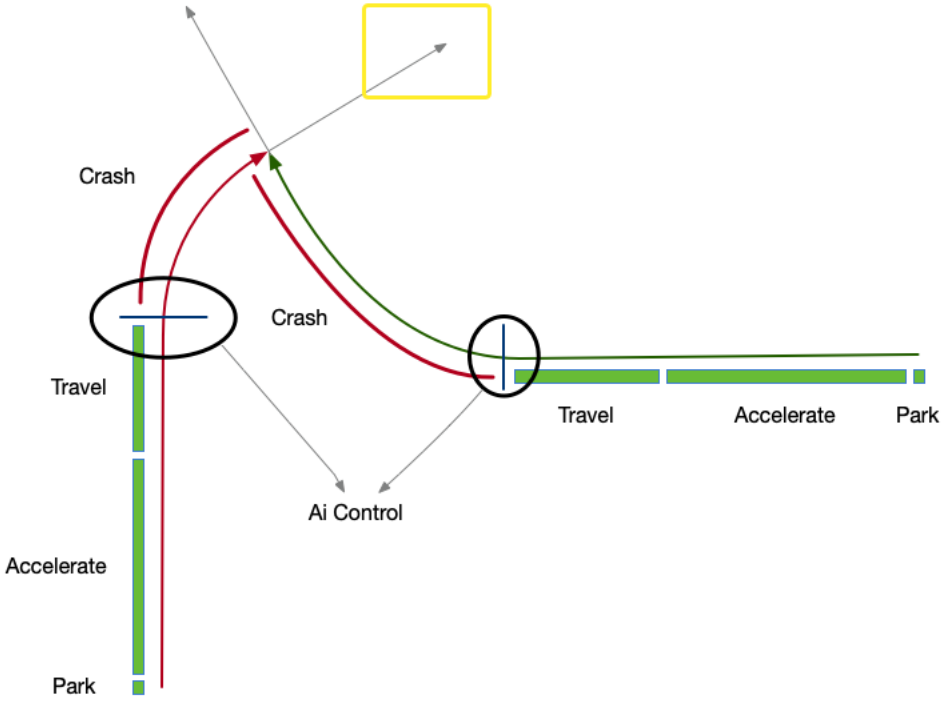

Figure 7: Sequence plan is applied to our example crash

got damaged or has slightly left the road. The test oracles applied are generic and can result in a fail even when manual inspection would consider the test case a success e.g., when it was the correct thing to leave the road to mitigate greater damage. To enable manual inspection after a failed test execution the test execution is only stopped by time out or when reaching the success area.

3.3 Input and Output Data

After describing the properties a test case consists of and how they are put into a FSM model, this section will explain what is used as input by the approach i propose to generate the test case and what the output data looks like. For both (input and output data) a XML document is used, because it is easy to read and easy to convert into java classes and can easy be generated from java classes. There are formats that already describe driving scenarios like Commonroad [5] and OpenDrive [15]. They provide a format for describing roads and Commonroad also provides a format for modelling trajectories in form of states. But they are lacking in the description of driving actions and critical events, which are needed for this system to generate the test case.

3.3.1 Input Data

The input files is where the user *declares*, what the finished test case should look like. For this task the system takes two files as input. One *configuration* file and one *input* file. The

configuration file contains information for running the simulation program (see attachments B). The configuration file also contains generation related information in form of value ranges in which the system is allowed to randomly pick data and the time that is allowed to pass during test execution until the test times out. These value ranges are the following:

- execution state length in meter
- road shape in meter
- velocity range in km/h
- bezier control point range in percent

The interval execution state length defines the interval for the length of the execution state in driving direction. Road shape defines the interval for the coordinate of the second road waypoint of the execution in orthogonal driving direction relative to the first one. Hence it defines possible directions for the curve and how big it is. The bezier control point range defines a percent interval, that describes fractions of the execution state length at which the bezier control point is allowed to be set in driving direction. For instance, if the execution state length is 100 meter and the bezier control point range is 20 to 60, the control point can be set between 20 meter to 60 meter in driving direction from the first road waypoint of the execution state. Where the bezier control point is set in driving direction influences how sharp the curve is. The interval for the velocity range simply defines the range for velocities, which are picked by the system during test case generation.

The *input* file is the place where the user can declare what the finished test case should be like. In other words, he can pick values for properties described in section 3.1. The system will then generate missing values for the remaining properties. The only mandatory input is the *Critical Event*. Properties, that can receive values via the input file are the properties of the *Execution states*, besides the bezier control point, the directions of the cars and general road properties. This input data is provided in a XML file to the system, which looks like the one reported in figure 8. This input can lead to a scenario set up like in our example of the two road angled crash. The EC passes waypoint (0,0) and crashes at (10,30) with the NEC. The EC approaches the crash location in Y direction and drives on a right curve. The NEC has no property specified besides velocity, hence other trajectory set ups like depicted in our example are possible as well.

Like mentioned in the previous section 3.1.1, the *Critical Event* comes with restrictions to value ranges of properties in form of relations between them. This means, that not every combination of input values is conform to the type of the critical event. This is the case if the received inputs violate constraints provided by the critical event. The system will tell the user, when it received a non conform input file during the generation process. To avoid non conform inputs here are some recommendations for picking input values:

- define the second waypoint for only one of the participants of a crash and not both
- define either a road- or a trajectory waypoint in combination with the driving lane for one point of the cars trajectory, if the car is supposed to drive on the road
- an input for the direction is only useful if not both execution state waypoints are defined

```xml
<?xml version="1.0" encoding="UTF-8"?>
<inputValues>
    <criticalEvent>TwoRoadAngle</criticalEvent>
    <EgoCar>
        <velocity>50</velocity>
        <wp1>(0,0,0)</wp1>
        <wp2>(10,30,0)</wp2>
        <exLane>1</exLane>
        <setLane>1</setLane>
    </EgoCar>
    <NEC>
        <direction></direction>
        <velocity>70</velocity>
    </NEC>
</inputValues>
```

Figure 8: Example XML file that can be used as an input for the system

Only one waypoint for the crash location should be defined, in order for the system to be able to generate a crash at this location. A waypoint for the trajectory of the car is fixed by a road waypoint and the corresponding lane the car drives on and a road waypoint is fixed by a car waypoint together with the lane, hence, to guarantee, that the car actually drives on this road it is recommended to define either road or trajectory waypoint. Both together with the lane number. The driving direction of the car is only needed if only one or zero waypoints of the cars trajectory are defined in the input file. With both waypoints given, the system will calculate the resulting driving direction and override a faulty one if needed.

3.3.2 Format of Input

After describing what can be received as an input, this section will briefly describe in what format the input values can be obtained:

- direction: X, Y, RX and RY, with RX and RY being the opposite direction to X and Y
- waypoints (road or trajectory): (x,y,z) with x, y and z being double numbers
- velocity: double number
- number of lanes: 1 and any other positive even integer (roads with an uneven number of lanes are not supported)
- road width: double number
- driving lane: Positive lane numbers are the lanes on the right side of the road and negative lane number lanes on the left side, starting at 0, which is the middle of the road.

3.3.3 Output Data

The test case generator produces two different kinds of output files. The scripts, that were used during the generation process to simulate the scenario and a XML file, which contains all generated test case properties with their values. The XML file can be used later to execute the test case. It has the same form as the input file, which is used for generating the test case, but contains additional properties, which are needed for execution like the trigger coordinates for NEC movement and information about setup states.

3.4 Method Summary

It is the goal of the system to generate an executable test case for autonomous cars, that matches all inputs provided by the user. That means, that the test case includes all these inputs and that the parts generated by the system complete these inputs to obtain a fully specified test case. That involves, that all cars content of the test case are able to meet the preconditions. The minimal input the system needs to operate is the *type of critical event*. The critical event delivers a shell in form of constraints in which the system will then generate missing values. With each missing value that was generated, value ranges for other properties can be limited. In this manner the system can pick values for each property, without violating constraints. A high priority during this process is, that user inputs are not changed and that they are present in the resulting test case.

The resulting test case will be executable in a simulation environment and provides test preconditions as well as test oracles. Dynamic objects have a finite state machine called sequence plan, which models their behaviour during test execution and provides trajectories for roads as well. The sequence plan contains two different types of states the *execution state*, which describes the critical event, and several *setup states*. Setup states are only used to enable vehicles to reach the preconditions, which are defined by the execution state.

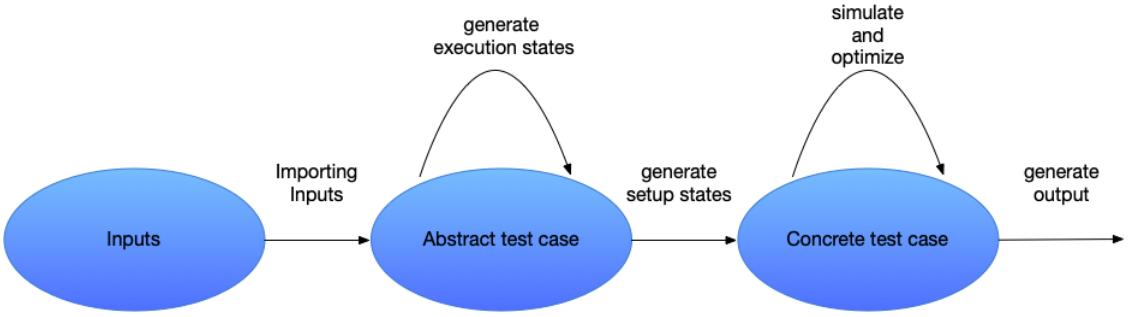

Figure 9: different steps of the generation process

4 Test Case Generation

After showing what is content of a test case and how this content is modelled for generation purposes, this section focuses on the generation process itself. The general idea is going from abstract to concrete by generating missing values step by step. The test case is generated using input provided in a XML format like described in section 3.3. This generation process is split in four major parts. First filling that input into the test case model, which is described in section 3.2. Second the abstract test case is generated. This includes generation of execution states for each traffic participant and other values, that do not depend on characteristics of the car model. Then, using the abstract test case and details of the car model, a concrete test case is generated. The fourth step is optimizing that concrete test case towards its fitness goals, which are velocity, trajectory and synchronicity, using local search algorithms. Figure 9 shows the different generation steps and their outcome. During the generation process the test scenario needs to be simulated several times, because we need the vehicle's positions during simulation to assign fitness values to the current version of the test case. For simulation beamNG research[28] is used. For that a python script using the beamNGpy[3] api is generated by the system.

The general approach for setting values of the execution states is picking them randomly from the set of allowed values in a predefined order. We limit the set of allowed values by satisfying constraints. Each time a new value is assigned to a property constraints are satisfied, this ensures that no values are picked that contradict a constraint.

During the next chapters some phrases are written in *italic* letters, while they are explained when they occur an explanation to many of them with according mathematical operations can be found in the attachments A at the end of the thesis.

Constraints Constraints are statically encoded in the type of the critical event, hence the test case objects and properties that are part of the relations are not generated when constraints are defined. Constraints are used to describe relations between test case properties of the form $v_1\ op\ v_2 + m$. v_1 and v_2 represent test case properties and m is a modifier variable. op is the comparative operator, which can be $<, >, =, \leq, \geq$. The only exception are direction constraints. They support the operations $=$ and $\neq$. The system only supports constraints on the second waypoints of executions states of test case objects (roads or traffic participants), on the velocity of traffic participants and the direction of test case objects. Hence, the constraint has two ids,

one for every test case object which is part of the equation, the modifier property, that describes the offset, the comparative operator and a direction variable, which specifies the coordinate that is constrained. Constraints with the = operator between traffic participants and roads do not need a direction variable, because the operator simply signals that the car drives on the road.

Now satisfying a constrained can be further explained. Test case properties have an interval of possible values they can have. Satisfying a constraint means that the interval boundaries are updated in order to fulfil the relation defined by the constraint. When trying to satisfy a constraint three cases are differed.

- both properties, that are part of the relation, have no value assigned
- only one property has a value assigned
- both properties have values assigned

If both properties have no value assigned, the constraint is not satisfied, but boundaries for possible values can be adjusted dependent on given boundaries. This ensures, that the system does not pick values, that contradict that constraint.

If one property has an assigned value, boundaries of the value ranges of both properties are updated. For instance, if the relation is the following: $vEC > vNEC$ (EC drives faster than NEC) and the velocity of the NEC is already set to a value. The upper boundary of the velocity interval of the NEC is set to the assigned value and the lower boundary of the velocity interval of the EC is set to the assigned value + 1. Figure 10 shows this process. One bar represents the whole value range. The grey coloured area depicts the range of values, that are allowed to be picked.

If both values are assigned both interval boundaries are updated as well, so that if one value needs to be changed during test case generation the interval in which it is allowed to be set is already defined.

An exception to this process of satisfying constraints is the = relation between a traffic participant and a road. The = operator in this case means, that the car has to drive on the specified road. Two ensure this during test generation a *onRoad* flag for that traffic participant is set. The constraint can not be satisfied, if an already set waypoint of the traffic participant does not lie on the road.

Constraints are provided by the type of critical event, which is received as the only mandatory input.

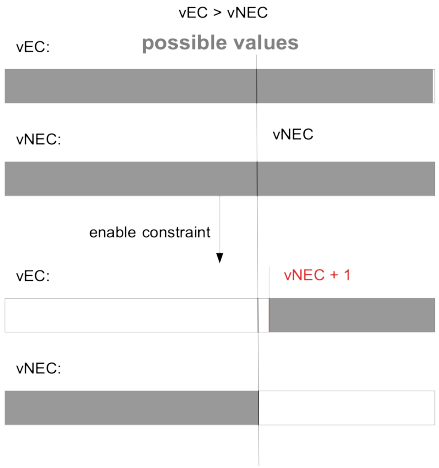

Figure 10: satisfying the vEC > vNEC constraint with vNEC already set. The grey areas depict the possible values.

Critical Event Properties of the critical event are the minimum needed amount of traffic participants and roads and a list of constraints it comes with. The critical event itself is all the system needs for test case generation. With making use of constraints its task is to guarantee, that the resulting test case is conform to the desired scenario. As described in the paragraph above a constraint provides a relation between two test case properties. For instance the relations listed below are used to describe the *TwoRoadAngled* crash of our example set up.

- number of cars: 2
- number of roads: 2
- EC.position = NEC.position in orthogonal driving direction
- EC.position = NEC.position in driving direction
- EC = Road1 (EC drives on the road)
- NEC = Road2 (NEC drives on the road)

As it is shown, the position in orthogonal driving direction and driving direction is equal for both cars. This makes sure that their trajectories intersect at wp_2 of their execution states (this is where the crash happens). Both cars drive on individual roads, this will lead to a crash at an intersection like in our example. Position constraints are differed based on direction, because it is possible that one coordinate needs to be equal and the other can differ in its value e.g., a *Front to Rear* crash demands that the position in orthogonal driving direction is equal but in driving direction one car has to drive slightly before the other.

Ego Car
(0,0) - (10,30)
50 km/h
Road Waypoints: WP1 - WP2
lane: 1

NEC
WP1 - WP2
70 km/h
Road Waypoints: WP1 - WP2
lane: -

Figure 11: Values of execution states after inputs have been added.

4.1 Importing the Inputs

At this point we are at the beginning of the generation and a blank test case is generated and missing traffic participants and roads are added dependent on the type of critical event. In the case of our Two Road Angled crash example we need two traffic participants and two roads. Now the content of the input file can be imported into the test case model. That means that properties receive their value defined in the input and are marked with a predefined flag. Properties marked with this flag are not changed by the system during test case generation in order to guarantee that the input is present in the finished test case. If any combination of the first road- or traffic participant waypoint and second road- or traffic participant waypoint of the execution state is received as an input, the driving direction of that traffic participant is calculated and set. When we import the example inputs shown in figure 8 we obtain the partially specified execution states shown in figure 11. The EC has received values for wp_1 and wp_2 hence we can calculate its driving direction, which is Y, because the difference in the Y coordinate is bigger than the difference in X coordinate.

4.2 Generating the Abstract Test Case

After filling the input data into the test case model, the actual generation process can start. First the abstract test case is generated. The Abstract Test Case has all the properties the concrete test case will have, but some are not specified. It is an intermediate step in the process

of the test case generation, which describes the test case without taking a specific car model into consideration. This means that properties of traffic participants other than the sequence plan are not specified. Waypoints of *Setup* states have no defined location as well, because they may have to be placed differently dependent on the used car model. This separation in abstract test case and concrete test case makes it possible, that the same abstract test case can be used to generate various concrete test cases by using different car models.

Starting point for this step is the result after adding the inputs to the test case. That means only values received by the user are present in the system at this point and based on that the remaining properties will be generated, while not changing any of the input values. Generating the abstract test case consists of the following steps executed in the order in which they are listed:

1. generating missing road properties aside from road trajectories
2. setting driving directions of cars
3. setting environment properties
4. generating values of the execution states of every car
5. adding setup states to all sequence plans
6. adding a success state to every sequence plan

Important to note is, that all generation steps, that are described in this chapter are only executed, if no input was received for the corresponding property.

4.2.1 Roads

Roads are generated after all traffic participants are added. If a road waypoint is marked as predefined, a road is generated and a reference to that traffic participant is set. Every road has a reference to the traffic participant in whose sequence plan its trajectory is defined. After adding missing roads their properties need to be defined, if they are not received as input. The road trajectory will be defined in a later generation step, when the execution states of traffic participants are generated. Hence values that are generated at this step are road lanes and width.

The system supports only even numbers for road lanes and only one lane. Reasoning for that is the calculation that determines the length that a waypoint needs to be shifted to its lane. If uneven road numbers are necessary only the algorithm, that calculates the shift length needs to be adjusted. When picking a lane number for a road, first the highest lane defined for a traffic participant that drives on that road is determined. If it was not received as an input no lane is set at this point. Lanes are numbered upwards from zero (middle lane) with positive numbers to the right and negative numbers to the left. For instance a four lane road has following lanes from left to right: $-2, -1, 1, 2$, this is shown in figure 12. In order to pick the amount of lanes a road has in total our system picks a random even number of lanes from the interval of the minimum amount to a upper boundary, which is received as an input in the configuration file. The minimum amount is determined by finding the highest already set lane of any traffic

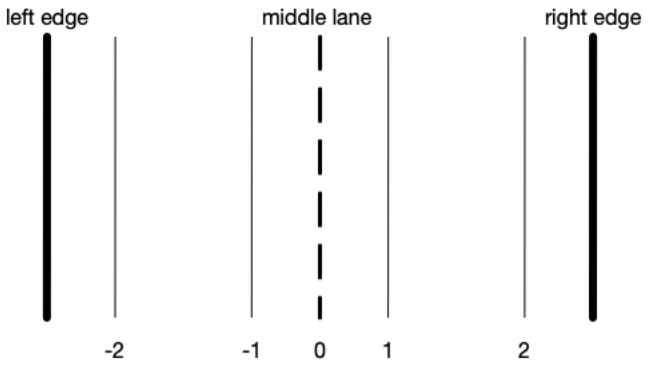

Figure 12: distribution of lane enumeration

participant that drives on this road and doubling the absolute amount of that lane number. In our example in figure 11 the lane defined for the EC is 1 that means that the road needs to have at least 2 lanes (-1 and 1), so that the EC can drive on lane 1. Hence the minimum amount is set to 2. For the NEC was no lane defined therefore there are no constraints on the amount of lanes of its road, so here 1 lane is used as the minimum amount.

After that the system can set the road width, which is dependent on the amount of lanes the road has. The system picks a random number between 2.5 meters and 3.8 meters as a width for one lane. After that it multiplies that number with the number of lanes the road has to receive the road width.

4.2.2 Set Driving directions

Each traffic participant has a set of possible directions. This set contains every direction (X, Y, RX and RY) as default. Before any direction is set, constraints on directions are enabled in order to limit the set of possible directions if necessary.

When enabling a direction constraint with operator $\neq$ it is checked whether the direction is set for one of the test case objects part of the relation. If so, that direction is removed from the set of possible directions for the other one.

When enabling a direction constraint with operator = the procedure is similar. If one direction is set, all other directions are removed from the set of possible directions of the other test case object. In case no direction is set, it is checked whether any combination of the first waypoints (road or traffic participant) and second waypoints (road or traffic participant) of any test case object that is part of the relation is set. Both test case objects need to have the same direction, therefore, their direction can be calculated, if a pair of first and second waypoint is given. Then the set of possible directions is set to that one direction for both objects.

After the set of possible directions was constrained to allowed values, the system picks a random one for every traffic participant from its set of possible directions. After that the system can set the orthogonal direction for every participant.

In our example we have no constraints on the directions of vehicles, that means that missing directions are picked randomly. The EC has already a direction assigned, because we were able to calculate it using both predefined waypoints. The NEC has no direction so a random one is assigned to it. In our example as depicted in figure 2 the NEC was assigned direction RX in this step.

4.2.3 Environment properties

Environment properties are simply set by picking a value for weather and lightning out of a set of possible values randomly, if no input was received.

4.2.4 Execution state

The idea, when setting execution state properties is, that the system picks values, that do not violate the constraints given by the *critical event*. For that task constraints are satisfied every time a value was set. That eliminates values that would violate a constraint from the set of possible values beforehand.

Properties are set in the following order:

1. velocity
2. driving lanes
3. traffic participant waypoints
4. road waypoints
5. duration

Velocity The velocity has as well as other properties an interval with allowed values. This interval is limited by constraints and the input received in the configuration file. When deciding at which velocity the traffic participant should drive in its execution state, a value is picked at random from that interval. In our example both velocities are already defined so no value is picked for them.

Driving lanes In this generation step the driving lanes for the execution state and setup states are set. The lane needs to be set before any waypoints are determined, because waypoints need to be set according to the defined lane number. Lanes for executions state and setup states are set independently and in the same way. If the number of lanes of the road is 1, the lane gets set to 0, what is the middle of the road. If the number of lanes is greater than 0, the system picks a random number between 0 and $\frac{numberlanes}{2}$. After that the sign of the lane is determined. The sign determines the side of the road on which the car drives on. The system picks the sign randomly with chances for a positive sign being 75 % and a negative sign 25%. The system prefers the right hand side over the left side, because that is the common road side to drive on. The vehicles perform a lane switch if the lane for their setup states differs from the execution state lane. The vehicle enters the execution state on the lane defined for the setup states and performs the lane switch to the execution state lane during the execution state. The EC in our example has already a lane assigned so only lanes for the NEC get picked at this generation step.

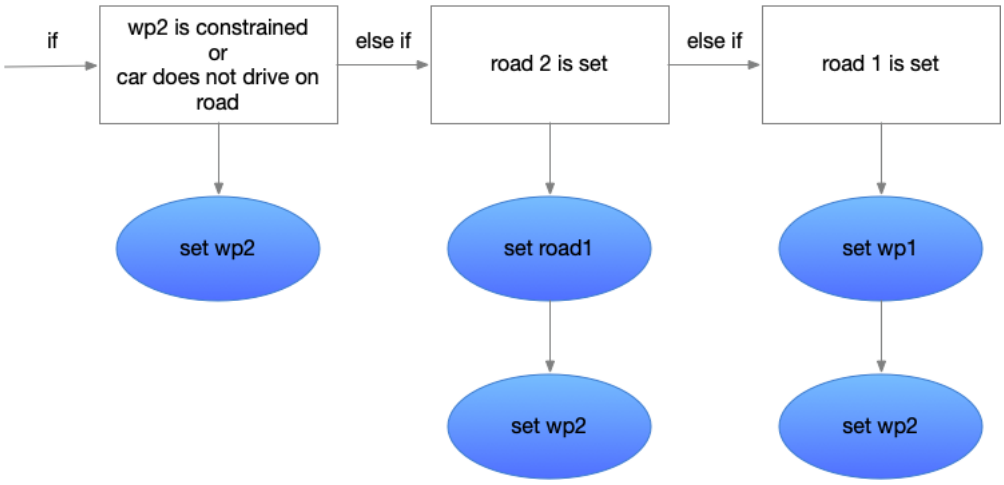

Figure 13: generation order of waypoints dependent on input

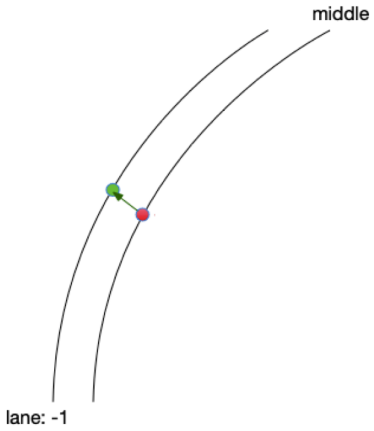

Figure 14: shifting a waypoint to lane -1 from the middle lane

Traffic Participant Waypoints The waypoints of the trajectory of traffic participants are two dimensional values. They have a X and Y coordinate. For setting a waypoint, its X and Y coordinate need to be set.

The order in that waypoints are set depends on the already set values of test case properties. It can be necessary to pick values for road waypoints before traffic participant waypoints can be determined. It is tried to set traffic participant waypoint 2 (wp_2) first. Figure 13 illustrates the order in which values are set and what influences this order. wp_2 is set first if its boundaries are constrained, if the car does not drive on the road or if there aren't any road waypoints (r_1 or r_2) defined already. wp_2 can not be set at first, if any road waypoint is defined. In case r_2 is already given, r_1 is set next, because wp_2 needs to be *shifted to its lane* and that is only possible, if the whole road segment is defined. Shift to its lane means that a waypoint is moved orthogonal to the trajectory at that point until it is placed on the according lane. Figure 14 illustrates this process. Important to note is, that in case wp_2 is constrained, it is always set according to

these constraints independent of r_2. In case only r_1 is predefined, the next waypoint set is wp_1, because r_1 constrains wp_1 in its position and wp_2 has to be set in relation to wp_1.

In our example no values for the waypoints of the EC need to be set, since they are already present. When generating the execution state for the NEC we still need to find values for wp_2. Its wp_2 is constrained by the already set wp_2 of the EC. As shown in figure 13 we start with picking values for wp_2, because it is constrained.

The algorithm for actually picking values for wp_2 differs dependent on whether wp_2 is constrained and if r_2 is defined or not. It is always set based on its constrained if it is constrained. Otherwise the way values are picked depends on whether r_2 has been defined already.

If r_2 is set, r_1 gets set next and the road *bezier point is calculated*. Then wp_2 can simply be *shifted to its lane* from r_2.

When picking values for wp_2 based on constraints it is started with the coordinate that is the most constrained. Meaning it is started with X, if for example the X coordinate has a defined minimal boundary and the Y coordinate has no boundaries defined at all. It is distinguished in three different cases.

- no boundary is specified
- only one boundary is specified
- both (min and max) boundary is specified

If no boundary is specified a reference trajectory in the cars direction is generated. The coordinate in driving direction is chosen randomly from the *execution state length* input received by the configuration input file. The coordinate in orthogonal driving direction is randomly chosen from the *road shape* interval, which is also received by the configuration input file. In case wp_1 is already defined, those chosen values are added to the corresponding coordinate values of wp_1 to obtain values for wp_2. If wp_1 is not defined, those values are simply taken as coordinate values for wp_2.

If one boundary (min or max) is given the interval of possible values for the coordinate is generated using this boundary. For instance if min is given: $max = min + x$. If max is given: $min = max - x$. x is describing the interval length. A greater x leads to more diversity in generated test cases using the same inputs. After setting the second boundary a random value from the interval is picked. If both boundaries are specified a random value from the interval is picked and used. This process is repeated until all coordinates of $wp2$ have assigned values.

In our example wp_2 of the NEC is constrained in both coordinates (X and Y) hence the order in which they are set is random. The position equal constraint that is applied limits upper and lower boundary and sets them to the same value. Hence one value is left for the system to pick and these are in our example the same as wp_2 of the EC. The resulting execution states after wp_2 and lanes have been set are depicted in figure 15

For setting wp_1, it is differed whether the traffic participant is *road defining* or not. The road defining vehicle is the traffic participant in whose execution state the road waypoints for the road

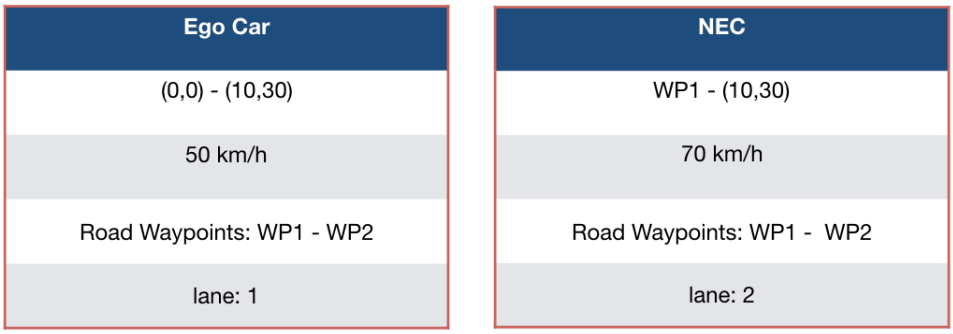

Figure 15: Values of execution states after waypoint 2 and lane was generated.

it drives on are defined. A traffic participant is not road defining, if it shares a road with another traffic participant and these road waypoints are defined in the other ones execution state. In our Two Road Angled crash example both traffic participants are road defining, because they drive both on their own road. If the traffic participant is *road defining*, it is distinguished whether the first road waypoint is already set or not. If it is set, the first waypoint of the traffic participant is simply *shifted to its lane*. This is possible without knowing r_2, because when reaching r_1 the traffic participant is travelling straight in driving direction. Therefore, the lane shift can be executed straight in orthogonal driving direction. If the road waypoint is still undefined, a reference trajectory is generated again. The X and Y coordinates from this reference trajectory are subtracted from already set wp_2. The result is used as coordinates for wp_1.

When finding values for wp_1 in our example we only have to define it for the NEC because the EC already has values for wp_1 defined. The NEC has no values set for the road waypoint so we need to randomly generate values from the *execution state length* and *road shape* interval that were received as an input in the configuration file. Those values are added to wp_2 dependent on the driving direction. In our scenario the NEC drives on a right curve in direction RX to the crash location. So for wp_1 (60,-10) could have been picked.

In case the traffic participant is not *road defining* wp_1 needs to be placed on the road, the traffic participant drives on. It is assumed, that both road waypoints of that road are set already. The EC is always a *road defining* vehicle, hence this generation step only is executed for NECs. When picking the distance between wp_2 and wp_1 the system tries to match the duration of the execution state to the duration of the execution state of the EC, by calculating the length l

dependent on the car's velocity v : $l = v \cdot d$. d is the duration of the execution state of the EC. Now a point on the curve between wp_2 and r_1 with distance l between wp_2 and wp_1 has to be interpolated. For that task a point on the straight line between wp_2 and wp_1 is calculated first, using following equations.

$$l = v \cdot d \tag{4}$$

$$elevation = \frac{\Delta O}{\Delta D} \tag{5}$$

$$\alpha = \arctan(elevation) \tag{6}$$

$$x_{diff} = \sin\alpha \cdot l \tag{7}$$

$$y_{diff} = \cos\alpha \cdot l \tag{8}$$

ΔO is the difference in orthogonal driving direction between wp_2 and r_1 and ΔD is the difference in driving direction between those points. x_{idff} and $y_d iff$ are then subtracted from wp_2 to obtain coordinates on the straight line. This point (wp_t) does still not lie on the bezier curve, that describes the cars trajectory. Hence, we need to use the equation for the bezier curve in order to find a coordinate pair, that is on the curve. For that task the previously calculated coordinate in driving direction of wp_t will be used to *get a point on the road* between r_1 and wp_2. *Get a point on the road* refers to the process of interpolating a point on a bezier curve with a given coordinate value in driving direction. This point is then *shifted to its lane*. As soon as both waypoints have been set, the according *bezier control point is calculated*.

In case l is greater than the distance between wp_2 and r_1, wp_1 does not lie between them. Hence, wp_1 has to be set in front of r_1. For that the remaining distance is added according to the driving direction to r_1 in driving direction. This is possible because traffic participants are assumed to drive on a straight road before entering the execution state.

Road Waypoints Road waypoints are only set in the execution state, if the traffic participant is *road defining*. Contrary to the determination of traffic participant waypoints, the calculation order of road waypoints (r_1 and r_2) does not depend on other inputs. The calculation only depends on whether the *road defining* traffic participant drives on the road. In our example both vehicles are road defining and constraints that demand them to drive on the road are present. A traffic participant is considered to drive not on the road if such a constraint is not present.

If the car drives on the road, both road waypoints can be *shifted to their lane* from the traffic participant waypoints that have been set in the previous step and the *road bezier point is calculated*. Since both traffic participants drive on the road in our example all road waypoints are generated this way resulting in the execution states depicted in figure 16.

In case the car does not drive on the road, r_2 is set based on constraints the same way as wp_2 has been determined. r_1 is *shifted to its lane* afterwards. As soon as both road waypoints are set the according road *bezier control point is calculated*.

There exists the corner case, that the first waypoint (wp_1) of the traffic participant was predefined by the user and no road waypoint was received as input and the critical event does not demand a separate road for that traffic participant. For instance, this can be the case for the "Front to Rear" critical event, when wp_1 of the NEC is received as an user input. Given that case, it is checked, whether wp_1 lies on a road defined by other traffic participants. This is achieved by

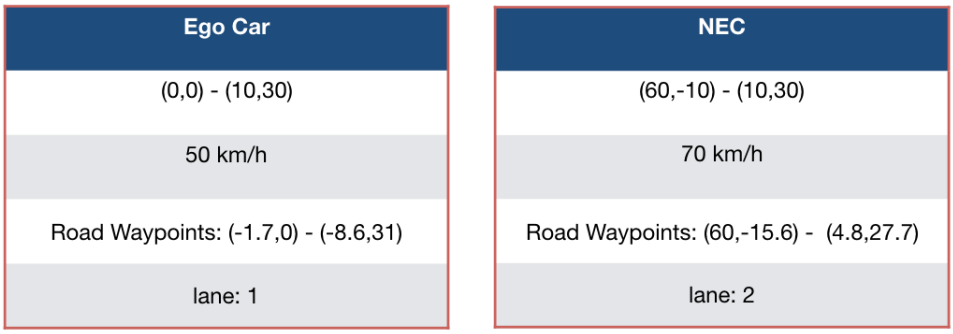

Figure 16: Values of execution states after road waypoints were generated.

calculating the road edges for the coordinate in driving direction of wp_1. For that task, first the *point on the road* is calculated and then *shifted to its lane* with shift length $\frac{width}{2}$ in both directions. If wp_1 lies between the road edges it is on the road. If it does not lie on another road, a separate road needs to be generated so that the NEC in our example does not drive off road. When an additional road is added to the test case, position constraints for that road and its traffic participant are added as well. This guarantees that other waypoints of the trajectory of the traffic participant are on the road as well. After generating this additional road, its trajectory is placed in the same way as for other roads.

Duration The duration of the execution state is calculated by $\frac{1}{v \cdot l}$. v is the velocity of the car and l the distance the car travels during the execution state. l is calculated using the Pythagoras theorem. This calculation can only be used as an estimation of the duration, because it assumes a constant velocity of the vehicle and neglects curves and lane switches. The duration will be used later to determine trigger coordinates for NECs.

Execution state generation order It is important, that the execution states of the different traffic participants are generated in a specific order. The = constraint between a traffic participant and a road has no direct impact on the boundaries for the waypoints of these test case objects. It only sets a *onRoad* flag for the traffic participant. Therefore, a transitive limitation for other traffic participants on their waypoint boundaries is not possible. If waypoints for those traffic participants would be generated first, it could happen, that the system is not allowed to

place waypoints on the road due to other constraints. This can be a problem, if the second road waypoint (r_2) was received as an input and generation starts with a traffic participant that has no road waypoint predefined.

We solve this problem by generating values for the execution state properties of traffic participants who received r_2 as an input first. After that all *road defining* traffic participants are specified, because the waypoints of all roads need to be specified before waypoints of traffic participants get set that have to drive on these roads. Eventually executions state values for none *road defining* traffic participants are calculated.

4.2.5 Setup states

Setup states are added after the properties of the execution states of every traffic participant are defined. Setup states are used to describe the trajectory of traffic participants before they enter the execution state. Which means before the critical event happens. They are needed to enable traffic participants to fulfil test preconditions. Waypoints for setup states are set when the concrete test case is generated. At this point in the generation process they are added to the sequence plan and velocities are set. For adding them the following rules are applied:

- The first state has type Park
- The last state has to be type Travel
- An Accelerate state has to follow the Park state

This results in the following sequence of setup states: Park to Accelerate to Travel. The states are added using this set of rules and not statically in this sequence, because it is possible that it is necessary to add setup states in a later generation process, when already some of them are existing in the sequence plan. With this set of rules no redundant states are added. The only property added to the setup states during this generation step is the velocity. The velocity of the Park state is always set to 0. The velocity of the Travel state is always set to the velocity of its successor state. In most cases that is the execution state. The velocity of the Accelerate state is set to the velocity of its successor state as well, if the successor state is no Accelerate state. In most cases its successor state is the Travel state. If the successor state is another Accelerate state, its velocity is set to a random value within the interval of the minimal allowed velocity and the velocity of the subsequent Accelerate state.

4.2.6 Success states

are added as the last step of the abstract test case generation to sequence plans. They have a road waypoint (r_s) and a traffic participant waypoint (wp_s) as properties. Hence, they extend both trajectories after the execution state or critical event. The ego car has to reach an area around its success state waypoint without triggering a test oracle to complete the test case. For NECs the extended trajectory from the success states is important, because they should not stop after reaching their last waypoint of the execution state. For them the success state is only a way

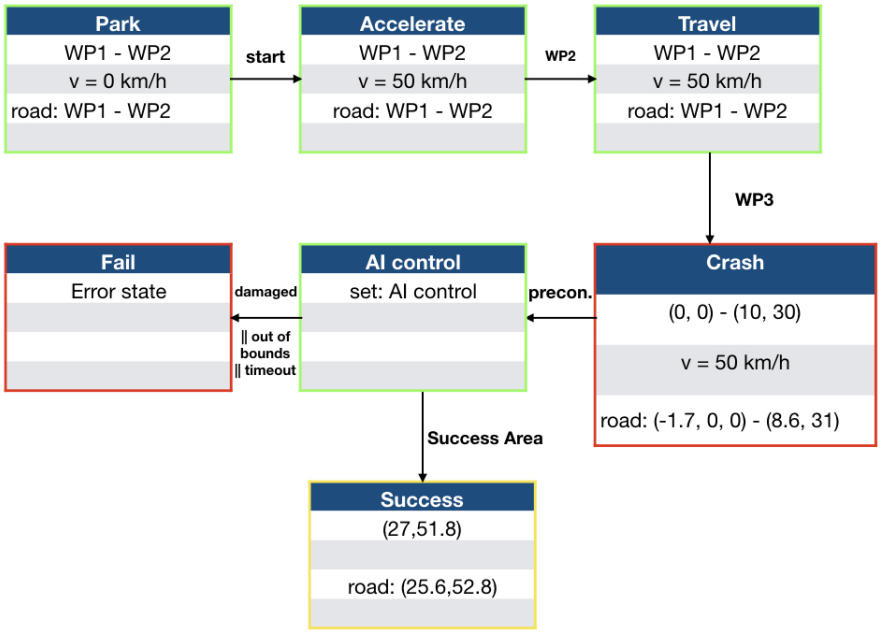

Figure 17: Success state, Setup states and their velocities have been added to our sequence plan of the EC

to describe the trajectory after the critical event, but is computed the same way as for the EC. The success state waypoints extend the trajectories of car and road with a straight line. This line is calculated the same way as described in equations 4 - 8.

If the car does not drive on the road, points used for calculation are the road bezier point and the second road waypoint of the execuiton state (r_2). This results in a straight road behind r_2. Then the traffic participant waypoint is *shifted to its lane*.

In case the car drives on the road the bezier control point (b) of the traffic participant and its second waypoint (wp_2) are used. This will result in a straight trajectory for the vehicle from wp_2 to wp_s without a crinkle at wp_2. This is the case because the vector $\overrightarrow{bwp_2}$ is tangent in wp_2 and this vector is continued as the trajectory to wp_s. Then the road waypoint can be *shifted to its lane* from wp_2.

The only exception to this is, when wp_s already is set on a road trajectory from another vehicle. In this case no value is assigned to r_s, because the car can follow the other road to reach wp_s and no additional road segment needs to be generated.

After adding success state and setup states with the corresponding velocities to the sequence plan of the EC in our example we obtain what is shown in figure 17. If we compare this sequence plan to the example sequence plan in figure 5 we notice that only road and traffic participant waypoints of setup states are missing to complete it. Values for those properties are generated in the next step.

4.3 Generating the Concrete Test Case

After the abstract test case was generated, missing values for properties of the setup states need to be determined. This will result in a fully specified test case, which is called *Concrete Test Case*. It is generated using the abstract test case and a car model as input. Waypoints for setup states are only specified with knowing the car model, because their placement can differ dependent on that car's acceleration. For instance a car with less acceleration power needs more space to reach the desired velocity and therefore, its waypoints for the acceleration state need to be placed further away than the waypoints for a car with more acceleration power. This separation of the generation process in abstract and concrete test case makes it possible to generate similar test scenarios for different car models. The generation of the concrete test case contains two generation steps:

- generating waypoints for setup states
- calculating movement triggers for NECs

4.3.1 Waypoints

Waypoints are calculated in two steps. First relative waypoints are generated for every setup state. Relative waypoints are waypoints that are not aligned with the waypoints of other states. They are used to describe the length and shape of the trajectory for one state, completely independent of other states and the driving direction of the vehicle. Every setup state has one relative waypoint w_r. This waypoint describes the vector from (0,0) to w_r. After those relative waypoints are determined, concrete waypoints can be calculated for road and traffic participant. This is needed because during the generation- or optimization process the length of individual setup states might need to be adjusted. With the concept of relative waypoints in place it is possible to only change the relative waypoint of that state and re execute the generation of the concrete waypoints. Concrete waypoints are generated every time before the scenario is executed, hence it is easy to modify the set up of a vehicle by only modifying its relative waypoints without touching concrete waypoints. This makes changing the set up trajectory simple.

Relative Waypoints When setting relative waypoints a driving direction in Y direction is used. The calculation of the relative waypoints is dependent on the type of setup state, which are Park, Accelerate and Travel. The Park state has its w_r set at (0,0), because the car does not move during this state. w_r of the Travel state is calculated with the following equation:

$$y = v \cdot d \tag{9}$$

The d parameter is an input for the calculation, which specifies the duration the traffic participant should stay in this state. v is the velocity set during the abstract test case generation step. Equation 9 calculates the distance a car travels with velocity v in duration d. The result is used as the Y coordinate of the relative waypoint which is at $(0,y)$. For the duration parameter d, 2 seconds are used for the EC and 1 second is used for NECs. NECs have a shorter travel time in this state, because they need to have a shorter over all duration in travelling through their trajectory than the EC. This is necessary because they need to start their movement dependent

on the position of the EC. d can be used to manipulate the durations the traffic participants need to travel through their trajectory.

So in our example the relative waypoint w_r of the travel state of our EC is $y = (50/3.6) \cdot 2 = 27.8$ and therefore w_r = (0,27.8).

When calculating the distance the traffic participant needs to accelerate to the desired velocity a constant acceleration is assumed. The needed distance s is calculated with the following equations [11]:

$$t = \frac{\Delta v}{a} \tag{10}$$

$$s = v_0 \cdot t + \frac{1}{2} \cdot a \cdot (t^2) \tag{11}$$

In equation 10 the time t the car needs for accelerating is calculated. Δv is hereby the difference in velocity from entering the acceleration state to leaving it. Equation 11 calculates the distance it needs to accelerate. v_0 is the velocity the car has, when entering the acceleration state. In most cases v_0 equals 0, because the car accelerates from the park state. The relative waypoint of the acceleration state is set at $(0,s)$. Calculating s for our EC with $\Delta v = (50/3.6)m/s$ and an acceleration $a = 3m/s^2$ and $v_0 = 0$ leads to an s of about 33 meters. Hence w_r of the acceleration state is (0,33).

Concrete Waypoints After setting all relative waypoints the concrete waypoints are generated. This step is repeated every time a scenario needs to be executed. First the road trajectory is set. Here it is distinguished whether the traffic participant is *road defining* or not. If it is *road defining*, the relative waypoints generated in the previous step are chained together dependent on the driving direction of the car starting at the first road waypoint of the execution state. A setup state has two road waypoints. The second (r_2) is equal to the first road waypoint (r_1) of the successor state. r_1 is the result of the addition of r_2 and the relative waypoint, dependent on the driving direction. The relative waypoint has the form $(0,y)$. Dependent on the driving direction different vectors are added, as equations 12 to 15 show.

$$\text{Y: } r_1 = r_2 + (0, -y, 0) \tag{12}$$

$$\text{RY: } r_1 = r_2 + (0, y, 0) \tag{13}$$

$$\text{X: } r_1 = r_2 + (-y, 0, 0) \tag{14}$$

$$\text{RX: } r_1 = r_2 + (y, 0, 0) \tag{15}$$

Figure 18 shows an example of this process in direction RX.

If the traffic participant is not *road defining*, which means that the car, for which we set the road waypoints, shares the road with that other traffic participant, references to those road waypoints are set in this sequence plan. It is ensured that the road waypoints are already set in the other sequence plan.

After setting road waypoints the waypoints of the trajectory of the traffic participants are set in their sequence plans. Again, it needs to be differed, whether the traffic participant is *road defining*. If it is road defining, the concrete waypoints are obtained by *shifting them to their lane* at the coordinate in driving direction of the road trajectory. In case the traffic participant is

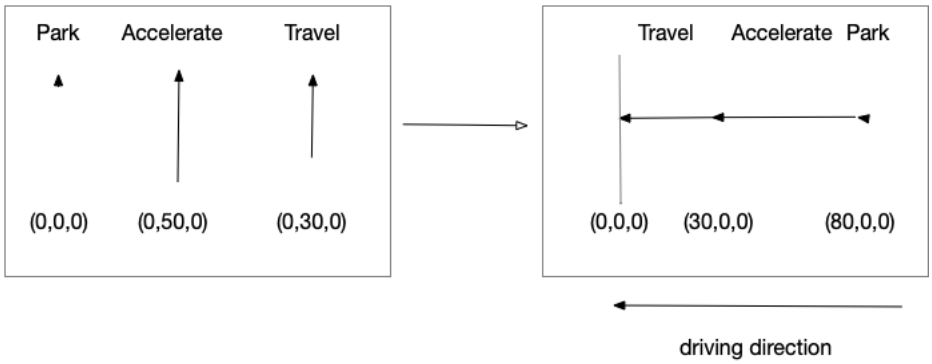

Figure 18: aligning relative waypoints to obtain concrete waypoints

not *road defining*, it is not as simple to place its trajectory on the road as described in the next paragraph.

A setup state has two waypoints: wp_1 and wp_2. The second waypoint equals the first waypoint of the successor state. In case of the last setup state, that is the execution state. For setting wp_1 the distance that needs to be travelled during the state, which is obtained from the relative waypoint, is compared to the distance to the next predecessor road waypoint from wp_2. If the distance between the relative waypoints is greater than the distance to the next road waypoint, the difference is added to the road waypoint like described in equations 12 to 15 dependent on the driving direction. If the difference is less than the distance to the road waypoint, wp_1 has to be placed between wp_2 and the road waypoint. If the road waypoint does not belong to a execution state, it is assumed, that the car travels on a straight line. That means, that dependent on the driving direction only one coordinate needs to be changed compared to wp_2, hence the remaining distance can be added to wp_2 in opposite driving direction to calculate wp_1. In case the road waypoint belongs to an execution state, it is possible that the car does not drive on a straight line in exactly one direction. Therefore, we need to interpolate a point on the bezier curve like described in previous chapters. If there are great differences in the velocities of the two cars it is possible that the trajectory of more than one Setup state lies on the road segment of another execution state, because a high velocity leads to longer states and a low velocity leads to shorter states. For exapmle in a "Front to Rear" critical event in which the EC goes with $100\frac{km}{h}$ and the NEC with $30\frac{km}{h}$, the NEC will likely drive during its setup states on the road of the execution state of the EC.

In our example the road waypoints of both cars are calculated by chaining relative waypoints together. Traffic participant waypoints are then obtained by shifting them on their lane from the road waypoints. This is possible because both traffic participants define their own road. Figure 19 shows the resulting sequence plan of the EC. As shown every property has now a value assigned, but they might be updated in a later generation step.

After setting every concrete waypoint for the setup states, the duration of that state is calculated. This might seem trivial because the duration was used to calculate the length of the setup states, but this only applies for the first time concrete waypoints are set. As mentioned before concrete waypoints are calculated every time a simulation is executed with the use of relative waypoints. The relative waypoints might have been changed since the first iteration and with that the duration of the setup states has changed.

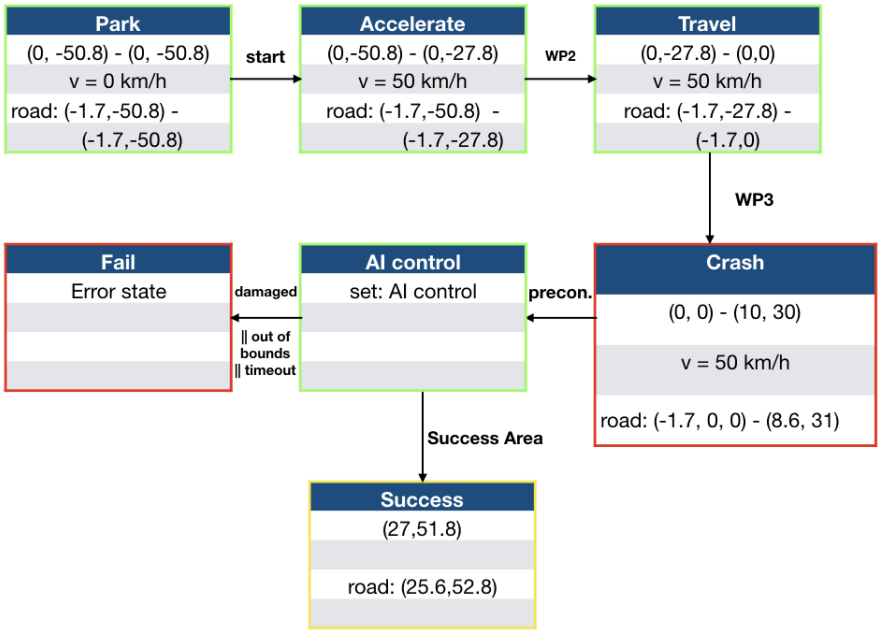

Figure 19: Values of the sequence plan after concrete waypoints for set up states have been added.

The duration of the Park state is always 0, because the car moves to the Accelerate state, after its movement is started and does not remain in the Park state. When setting the duration of the Accelerate state, first the distance the car needs to accelerate is calculated using equations 10 and 11. Then the difference of that distance and the distance of the Accelerate state is calculated. The duration set for the Accelerate state is the time the car needs to accelerate (equation 10) added to the time it needs to travel the remaining distance ($time = \frac{remaining}{velocity}$) after accelerating in the Acceleration state. In most cases that remaining distance is close to 0. The duration of the Travel state is calculated by: $time = \frac{distance}{velocity}$.

4.3.2 Movement trigger

Movement trigger are set in the last step of the concrete test case generation, because a fully specified sequence plan is needed in order to calculate them. The first car that starts its movement is always the EC. NECs start their movement dependent on the position of the EC. This position is what is called the movement trigger. It is the coordinate (X,Y) of the EC at which a NEC starts its movement and is part of the specification of the NEC. The idea is, that if NEC and EC have to be at the same time at a certain location, the NEC starts its movement through the sequence plan, when the EC has reached the position from which it needs the same time to its destination as the NEC.

For calculating this position, at first the duration of the EC and NEC that they need to travel through their sequence plan to the synchronized location ($egoDuration$, $necDuration$) is calculated. For that the durations of the individual states, which have been set in previous generation steps, are added up. Then the delay is calculated with: $delay = egoDuraion - necDuration$. This delay is the time the NEC needs to wait until it can start its movement.

If $delay$ is negative, the NEC needs longer to move through its sequence plan than the EC. This is likely to happen, if the NEC has a greater velocity than the EC and needs more time for acceleration. If that is the case, the setup states need to be refined. That means that either the duration of the NEC needs to be shortened or the duration of the EC needs to be longer. The better option here is to shorten the time the NEC needs, because by changing the duration of the EC, movement triggers of every NEC need to be recalculated. The setup states that are changed for that cause are the Travel states, because they do not demand a minimal length unlike the Acceleration states. As default the time the NEC spends in the Travel state was set to 1 second. If $-delay < 1$ it is possible to shorten the Travel state of the NEC by $-delay$. If $-delay > 1$ the duration of the EC is elongated by $-delay$. For changing the setup states, the steps described above for setting concrete waypoints are executed again with the new duration for the travel state as input. This process of refining the setup states is repeated until $delay = egoDuration - nexDuration > 0$.

With $delay > 0$ the movement trigger can be calculated. As a first step, the setup state during which the trigger will be placed is found. This is done by comparing the $delay$ to the durations of the setup states. The trigger calculation depends on the type of setup state. If the trigger state is a Park state, the initial position of the EC is used as a trigger.

If the trigger state is a Travel state, the duration of the previous setup states is added up (d_p) and the delay in the Travel state itself is calculated by $delay = delay - d_p$. Now the distance s the EC travels during $delay$ is computed with $s = velocity \cdot delay$. Dependent on the driving

direction of the EC the movement trigger is set by adding s to the corresponding coordinate wp_1 of the Travel state.

For setting the movement trigger during an Accelerate state, the distance s the EC travels during $delay = delay - d_p$ needs to be calculated as well. This is achieved by using equation 11 and substituting t with $delay$. Now s is added to wp_1 of the setup state dependent on the driving direction of the EC.

When we calculate the trigger for the NEC in our example we calculate the duration for both traffic participants first. Results are: $egoDuration = 8.96$ and $necDuration = 10.82$ so $delay = egoDuration - necDuration = -1.84$. The delay is negative so we need to refine our setup. The delay is negative because the NEC drives faster than the EC and needs more time to accelerate. $-delay >= 1$ in our case, which is longer than the NEC needs to drive through its travel state, hence we can not shorten it enough and need to elongate the travel state of the EC. This resulted in $egoDuration = 10.82$ and therefore $delay = 0$. $delay$ is still not greater than zero, so we need to refine the setup again. This time $-delay < 1$ so we can shorten the duration of the NEC to be able to calculate a trigger. With the delay being nearly 0 the NEC starts its movement together with the EC. The calculated trigger is (0, -85.9). Now we have finished generating values for all properties of our TwoRoadAngled crash test case. Figure 20 depicts the finished sequence plan of our EC. When we compare that to the previous iteration shown in figure 19 we can notice that the concrete road and traffic participant waypoints of the setup states have been moved. This happened due to the refine steps we had to take in order to elongate the duration the EC needs to travel through its sequence plan. We have no guarantee that the cars can follow the trajectory that was calculated and that the timing of the crash is correct, hence we need to simulate our scenario and optimize it if necessary.

4.4 Simulating the scenario

At this point of the generation process all properties needed for simulating the scenario in beamNG [28] have been generated. To be able to optimize the test case toward its fitness goals, it is necessary to simulate it beforehand. A simulation for each traffic participant on its own is generated. This is done to check if the traffic participant runs through its sequence plan as expected. After that the whole scenario is simulated with each traffic participant. For simulating the scenario the beamNg python api beamNGpy[3] is used. This api provides methods to easily generate and run a scenario in beamNg using a python script. The system creates and executes this python script.

Cars are added to the simulation at their initial position, which is defined by the Park state. A damage sensor is attached to each car in order to track at which position during simulation they have been damaged. Trajectories of cars and roads are added to the python script as they are specified in the sequence plans of the traffic participants. When dealing with a curved trajectory, points of the bezier curve are calculated using the quadratic bezier curve (equation 3). Values for t are chosen in 0.10 steps from the starting t to the end t. Those values for t are calculated using equation 25 and the corresponding points of the trajectory that are on the curved sequence (beginning and end waypoints of the part of the trajectory, that lies on the curved road).

The road trajectory is a list of 4 tuples containing three coordinate values for X, Y and Z and the width of the road at that position. The trajectory of the traffic participant is a list of

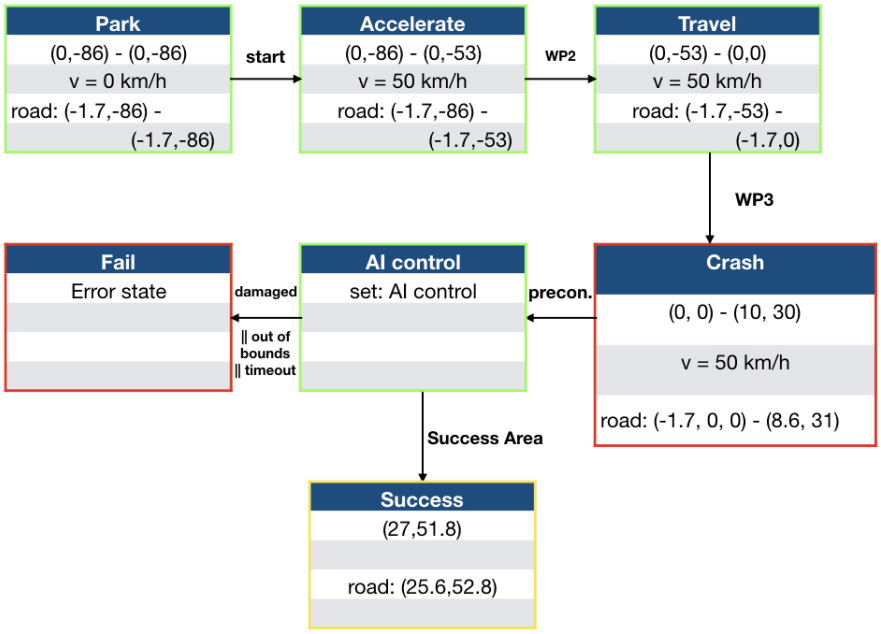

Figure 20: Finished EC sequence plan

dictionaries containing two entries. The position as a 3 tuple and the speed of the car, when passing the position. The waypoint of the park state is not content of this trajectory, because the car is placed there initially and does not need to drive there. For running the cars through their trajectory the *ai_set_line(trajectory)* method of the beamNgpy api is used. It takes the list of dictionaries as an input parameter and starts the movement of the car following the specified trajectory with the specified velocity.

During simulation execution it is checked in which state of the sequence plan each traffic participant is by comparing its coordinate in driving direction with coordinate of the first waypoint of the next state. Movement of NECs is triggered the same way. Here the position in driving direction of the EC is compared to the coordinate of their movement trigger. While the simulation is running the process continuously prints messages. These messages are read by our system. There are two different messages. A message that provides information about the state, position, velocity and damage of the traffic participant (m_{state}). This message is used to track the behaviour of the traffic participants during simulation. The second message is sent, when the EC reaches a location, that is concurrent to another traffic participant (m_{conc}). This message contains the position of the other vehicle at that time. This message is used to assess, if the other vehicle has indeed reached the specified location at the intended time.

The system for test case generation reads the output of the simulation and parses the messages. Messages m_{state} are converted into nodes. A node has the following properties:

- velocity

- position coordinate
- damage flag

Values for velocity and position coordinates are used from the simulation. The damage flag is set, if the damage sensor of the traffic participant provided values greater than 0, which means that the car is damaged. m_{state} also provides information about the state the car was in, hence, the resulting list of nodes can be assigned to the according states in the sequence plan.

Messages m_{conc} are converted into a coordinate, which is stored within the vehicle, that has to be concurrent to the EC at a specific location. This coordinate contains the position of the vehicle to the time, when the EC entered the concurrent location.

4.5 Optimizing the Test Case towards its Fitness Goals

The previous steps of test case generation have generated every missing property of the test case, but it is not guaranteed, that the traffic participants can actually follow the generated trajectory with the intended velocity. In addition to that calculations regarding acceleration distance or regarding movement triggers can be faulty, because they are based on the assumption, that the car has a constant acceleration and drives with constant velocity. Hence, it might be necessary to update the already generated scenario in these regards, to ensure, that everything runs as expected and needed during test execution. For that task three different fitness goals have been defined.

- every traffic participant needs to reach its defined velocity
- every traffic participant can run through its defined trajectory
- sequence plans are synchronized

The first two goals are self explanatory. Sequence plans are synchronized means, that cars, that have to reach locations at the same time, do that as planned. In most cases these locations are the same or at least so close that the physical car bodies overlap, because these cars are supposed to crash.

To find a test case set up that sufficiently fulfils every fitness goal local search is used. Each fitness goal is optimized individually, so three local search algorithms are implemented, which are explained in the following sections. As a starting point for the search the test case generated in the previous steps is used. Local search is a suiting algorithm, because the system aims to keep the finished test case as close to the inputs as possible, because the goal is to generate a test case, that is close to received inputs. In case it is infeasible to fulfil the fitness goals with the received inputs it is possible to allow the search algorithm to mutate input values. By picking only neighbours of the predefined value it is guaranteed that the resulting value stays close.

For each fitness goal a fitness function is used to describe how close the simulation is to fulfilling the goal. f_v to describe the velocity goal, f_t to describe how well the car runs through its trajectory and f_c to describe how close the cars are to fulfil their synchronicity goal. The closer

the values of the fitness functions are to 0, the better fits the simulation. As described in section 4.4 every state has a list of nodes, that describe the movement of the car in the simulation during this state. State transitions in the simulation take place, when the car passes a certain coordinate in driving direction. This list of nodes of every state is used to calculate the fitness values in combination with the specification in the sequence plan. Every fitness function is optimized independently. First f_v is optimized, because the velocity of the traffic participants influence their ability to run through their trajectory. Second f_t is optimized. Eventually, when every traffic participant runs through its trajectory with the specified velocity, f_c is optimized. f_c is optimized last, because it is not necessary to change waypoints or velocities to optimize it. For this step only the movement trigger coordinates need to be changed. Therefore, no re evaluation of f_v and f_t is necessary when making changes to the scenario setup due to f_c optimization. Fitness values f_t and f_v are calculated for each state of the sequence plan individually. The fitness value of a traffic participant for f_t and f_v is the worst value one of its states has.

4.5.1 Velocity optimization

The fitness function for velocity is defined differently for setup states and the execution state. For a setup state the fitness function calculates the distance between the specified velocity of the sequence plan and the actual velocity the car had, when leaving the setup state. The reasoning behind this is, that setup states are used for enabling the car to meet the requirements of the execution state, which are test preconditions. In order to fulfil this task it is not necessary that the traffic participant has always the required velocity while running through its setup, but it is necessary that he runs through the sequence plan as planned. Therefore, it is important that the car reaches the specified velocity at least when leaving a setup state. When determining f_v for the execution state, it is important that the car enters the state with the defined velocity. A correct setup guarantees that the car has reached the needed speed, when it enters the critical event and the system under test potentially takes over control of the vehicle. Equation 16 calculates f_v. Dependent on the kind of state, setup or execution state, the last or first node of the state is used to determine v_{node}. v_{plan} is the velocity defined in the sequence plan for the state.

$$f_v = v_{node} - v_{plan} \tag{16}$$

If the traffic participant is too fast f_v is greater 0. If he drives too slow it is negative. Assumptions when updating the test case regarding f_v towards the simulation programm are:

- the car does not keep accelerating after it has reached its specified velocity
- the car does not slow down, when driving on a straight lane, if it is not demanded

Based on these assumptions, the car should always enter the execution state with its required velocity, if enough space for acceleration is available during setup states. In addition to that, the car should not be able to go too fast, because it stops accelerating when the specified velocity is reached. Hence, if the car goes too slow and because of that the fitness goal is not fulfilled, the distance of the acceleration state is extended. The new distance is calculated the same way as the original described by equations 10 and 11 with changes to Δv, which are $\Delta v = \Delta v - f_v$. With a negative f_v the new Δv is bigger than the original, that leads to a longer distance for the accelerate state, when applying the same calculation with the new Δv.

4.5.2 Trajectory optimization

The fitness function for the car running through its trajectory evaluates how close the traffic participant stays to its specified trajectory. It is calculated the same way for each state of the sequence plan. For determining this fitness value each car is simulated on its own, because otherwise other cars might prevent the traffic participant from staying on its trajectory. This could happen for example if they crash, and with synchronicity not optimized yet, it can happen unpredicted. Therefore, this fitness function is optimized with each car simulated independent of other cars excluding possible interference from them.

Calculating f_t The fitness for a state is calculated the following way:

$$f_t = max(|f_{t_1}|, |f_{t_2}|, |f_{t_3}|, \cdots, |f_{t_n}|) \tag{17}$$

Here f_{t_k} is the distance of the car to its planned trajectory for a certain position in driving direction. With $k \in \{1, \cdots n\}$ and $n = |nodes|$. Nodes describe the trajectory, the car had during simulation. To calculate f_{t_k} the two waypoints of the car's trajectory between which the k_{th} node lies are determined. These waypoints are not necessarily wp_1 and wp_2 of this state, because road waypoints can be between these waypoints. If that is the case, the trajectory the car follows during this state includes these road waypoints. After the two waypoints (t_1 and t_2) of the trajectory, between which the k'th node lies, have been determined, it is differentiated, whether the car travels on a curved trajectory between these waypoints or on a straight line. Important to note here is, that the car can travel on a curved line even if the current state is a setup state, because it might share the road with another vehicle, that has defined a curved road at this location. In the following paragraphs a d in the indices means that it is the value of a coordinate in driving direction. An o in the indices indicates the value of the coordinate that is not the driving direction. For instance, t_{2_o} is the X coordinate of the second waypoint, if the driving direction of that traffic participant is Y. In both scenarios (straight or curved trajectory) the value of the coordinate not in driving direction, where the car should have been (c_o) needs to be determined. c_o is then compared to where the car actually was (v_o) for the same coordinate in driving direction (v_d).

If the car travels on a straight line between the two waypoints, values for the linear equation, that describes the straight line between t_1 and t_2 have to be found. This equation has the form:

$$c_o = m \cdot v_d + t \tag{18}$$

$$\text{with } t = -(m \cdot t_{2_d}) + t_{2_o} \tag{19}$$

m is the elevation between t_1 and t_2. c_o is the coordinate we are looking for, that lies on the straight line between t_1 and t_2 at position v_d. v_d is the value of the k'th node in driving direction.

If the car travels on a curved trajectory c_o has to be calculated using the bezier curve. For that task a point on the bezier curve that shares v_d needs to be determined. A quadratic bezier curve is defined by three points. The start point of the bezier curve is the first waypoint of the state (wp_1). The end point is the second waypoint of the state (wp_2). The third point is the bezier control point set in the execution state. Using these points as the parameters of the bezier curve the t value at the point v_d can be calculated. With t the corresponding point on the curve is calculated and c_o is received. In case the state, for which f_t is calculated, is a setup state, wp_1 and wp_2 are substituted with road waypoints of the execution state *shifted to their lane.*

Eventually we can calculate f_{t_k} with the following equation:

$$f_{t_k} = c_o - v_o \tag{20}$$

v_o is the value of the node in orthogonal driving direction. f_{t_k} has positive values, if the car drives too much to the right and negative values if the car drives too much to the left. The overall fitness regarding the trajectory of the state is determined like described in equation 17. It is the maximum deviation of the car from its trajectory during the state.

Optimizing f_t When dealing with f_t values that are not sufficient, the actions taken depend on the state. During setup the traffic participant travels on a straight line, hence deviations from the trajectory are unlikely and no changes to the trajectory or velocity are made. If the fitness value of the execution state is not sufficient, there are two options to make changes. First changing the position of the waypoints and second changing the velocity. It is tried to change the trajectory first, because the velocity has impact on the criticality of the scenario regarding the criticality measure time to action [42]. Trajectory or velocity can be changed only if they are not predefined by the user input.

Manipulating the test case results in a so called neighbour of the local search algorithm. Some neighbours are neglected during this search, because of assumptions made based on the search domain.

- the vehicle can always follow a straight line
- the vehicle is unable to follow a trajectory because the curve is too sharp or the vehicle is too fast

Therefore, neighbours with sharper curves or higher velocities are not considered, when looking for a scenario configuration. Our system has the following operations to manipulate the test case for optimization purposes:

- change the shape of the curve (move bezier control point)
- move waypoint two of the execution state (wp_2)
- move waypoint one of the execution state(wp_1)
- lower velocity by 5 km/h

When changing the shape of the curve, the bezier control point is moved by f_t meters in opposite driving direction. This results in a flatter curve. The system stops moving the bezier point, when half the distance between wp_1 and wp_2 is reached. A straight line woud be reached, when the coordinate in driving direction matches the corresponding coordinate of (wp_1).

For updating wp_2 the system tries to move the waypoint in driving direction for $0.5 \cdot |f_t|$ meters and $0.5 \cdot f_t$ meters in orthogonal driving direction. These actions are taken independent of each other, meaning, that if the system is not able to move wp_2 in one direction, it can still mutate

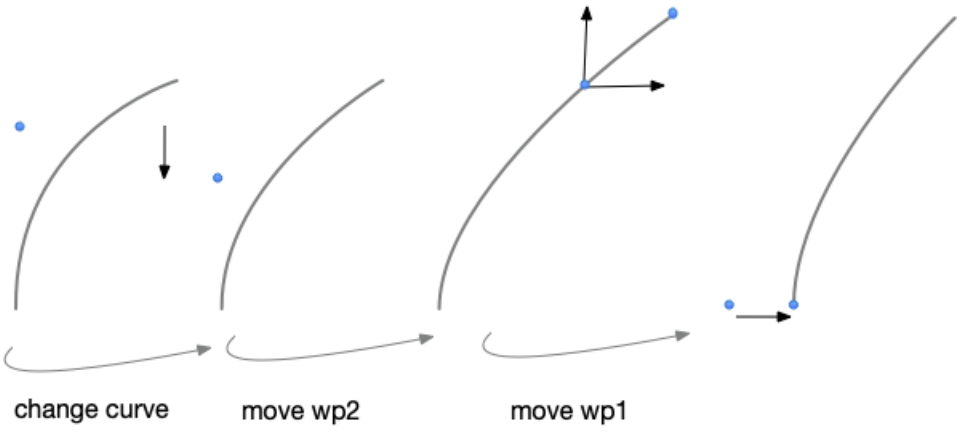

Figure 21: implented mutations to optimize f_t

the other one. This results in a flatter curve, hence it is unlikelier for the vehicle to drift off its trajectory.

When wp_1 is updated, it is moved by $0.5 \cdot (-f_t)$ in orthogonal driving direction. It is moved in the opposite direction as when wp_2 is updated, this results as well in a flatter curve.

Figure 21 depicts the effect of every mutation on a curve. As it is shown every mutation results in a slightly flatter curve, which makes it easier for the traffic participant to follow the trajectory.

The ability to change a test case property can be limited by two factors:

- a constraint is violated
- the property is predefined

This results in three degrees of manipulating the test case, listed below from weakest to strongest:

- changes that do not affect constraints or predefined flags (degree 1)
- changes that violate constraints but not predefined flags (degree 2)
- changes that violate predefined flags (degree 3)

The algorithm for updating the test case tries to optimize f_t using manipulations of the weakest degree possible, but if necessary it can even remove predefined flags, if allowed by the user. Therefore, the algorithm can be categorized into three stages, one for every degree of manipulations it tries to execute. For every mutation step, only one property is changed.

Mutations are tried to execute in the following order:

1. change wp_2 (degree 1)
2. change curve (degree 1)
3. change wp_2 (degree 2), only if EC is mutated
4. change wp_1 (degree 1)
5. change velocity (degree 1)
6. change velocity (degree 2)
7. change wp_2 (degree 2)

In case the system was not able to perform any of those mutation operations, the test case can not be mutated further because predefined flags prevent the system to change properties. Then the user will be asked, whether he allows predefined flags for the car that is mutated right now to be removed. If he agrees, predefined flags of velocity, traffic participant waypoints and road waypoints are removed and the mutation process is continued. If predefined flags are not allowed to be removed, the system will stop mutating the according traffic participant and continues the optimization process with other traffic participants.

Whenever one action could be performed, the system stops mutation and runs the simulation again to recalculate fitness values. And restart the mutation process if necessary. This whole process is repeated, until every traffic participant runs through its trajectory as planned.

4.5.3 Synchronicity optimization

After f_v and f_t have been optimized for every traffic participant, f_c is optimized. It is important for this step, that all cars run correctly through their sequence plan and no changes to their trajectory need to be made at this point of test case generation. For evaluating the sychronicity during scenario execution, the scenario with all traffic participants is simulated.

Calculating f_c

The content of the message m_{conc} described in section 4.4 is used to calculate f_c. This message is sent, when the EC reaches a location, that needs to be synchronous with another vehicle. Being synchronous means, that when the EC reaches that certain spot, the other vehicle has to reach a certain location as well. Most times both locations are the same, because both cars crash into each other at this location. m_{conc} contains this location of the other vehicle. Like described in section 3.2 and as depicted in figure 6 synchronicity is enabled by starting the movement of the NEC at the right time. That means that the NEC starts running through its trajectory, when it needs the same amount of time to reach the synchronous location like the EC from the trigger coordinate. Using f_c the distance from that is measured. It is calculated the following way:

$$f_c = pos_{sim} - wp \tag{21}$$

$$f_c = pos_{damaged} - wp \tag{22}$$

Equation 21 is used, if the cars did not crash before they reached the defined locations. This happens for instance, if both cars should crash in the middle of an intersection, but the movement of the NEC car got triggered too early or too late. The result of that is, that the NEC did not reach the crash location, when the EC arrives or has already passed it. pos_{sim} is the coordinate in driving direction the NEC had at that time and wp is the coordinate, where it should have been. If the NEC drove too far the sign is positive and if it has not reached the location yet the sign is negative. Equation 22 is used if the cars crashed before they reach the specified location. This can happen for instance, if both cars drive in the same direction on the same road and the NEC movement got triggered too late, so it did not reach the crash destination in time. $pos_{damaged}$ is the coordinate in driving direction, where the NEC got damaged first. This coordinate is obtained by the message m_{state}, which indicates if the car is damaged or not.

Optimizing f_c

Dependent on the results of f_c the movement trigger is updated. For that purpose the delay d, the NEC needs to be delayed additionally, is calculated. d can be positive, if the car movement needs to be further delayed, or negative, if the car needs to start moving earlier. Using d, the movement trigger itself is recalculated with the same method as for the first calculation of the trigger described in section 4.3.2. The original delay for that calculation was $delay = egoDuration - necDuration$. Hence, the new delay for the trigger calculation is $delay = egoDuration - necDuration + d$. The trigger coordinate is not simply moved by f_c in the corresponding direction, because the distance the trigger needs to be moved is dependent on the state of the EC during which the movement of the NEC is started. For instance, during the Accelerate state the same delay results in a bigger distance than in the travel state, because the car has not reached its full speed, yet. In addition to that it is possible, that the setup needs to be refined like described in section 4.3.2, if $d + necDuration > egoDuration$.

So updating the movement trigger narrows down to finding the correct value for d. For that purpose a slightly modified binary search algorithm was implemented. Binary search picks an element in the middle of the search space and continues the search with the upper half, if the value of the searched element is in the upper half, or with the lower half, if the searched element is in the lower half. This process is repeated until the searched element was found or the search space is empty [4]. The difference of the implemented algorithm to binary search is that we do not simply split the search space in half, but calculate a new value for d with $d = d + \frac{f_c}{v}$ with v being the cars velocity. The initial value of d is 0. In case the new value for d does not lie in our search space we come back to using a traditional binary search algorithm and cut the search space in half to obtain the new value for d. The sign of the fitness function f_c tells us whether we need to continue the search in the upper or lower section.

In case the setup states need to be refined during this optimization process the current search space boundaries and delay are reset. The following pseudo code describes the algorithm for picking new values for d and the upper- and lower boundary. The lower Boundary refers to the search boundary closer to zero and upper boundary to the boundary further from zero than the lower boundary ($|lowerBoundary| < |upperBoundary|$). The $newDelay$ is always set between $lowerBoundary$ and $upperBoundary$. It is calculated with $newDelay = d + \frac{f_c}{v}$. In case this $newDelay$ does not lie between the boundaries, the value in the middle between them is picked for $newDelay$. Figure 22 shows an example of finding the correct value for $newDelay$.

if $d < 0$ **then**
  if $f_c < 0$ **then**
    lowerBoundary $= d$
    $d = newDelay$
  if $f_c > 0$ **then**
    upperBoundary $= d$
    $d = newDelay$
if $d > 0$ **then**
  if $f_c > 0$ **then**
    lowerBoundary $= d$
    $d = newDelay$
  if $f_c < 0$ **then**
    upperBoundary $= d$
    $d = newDelay$

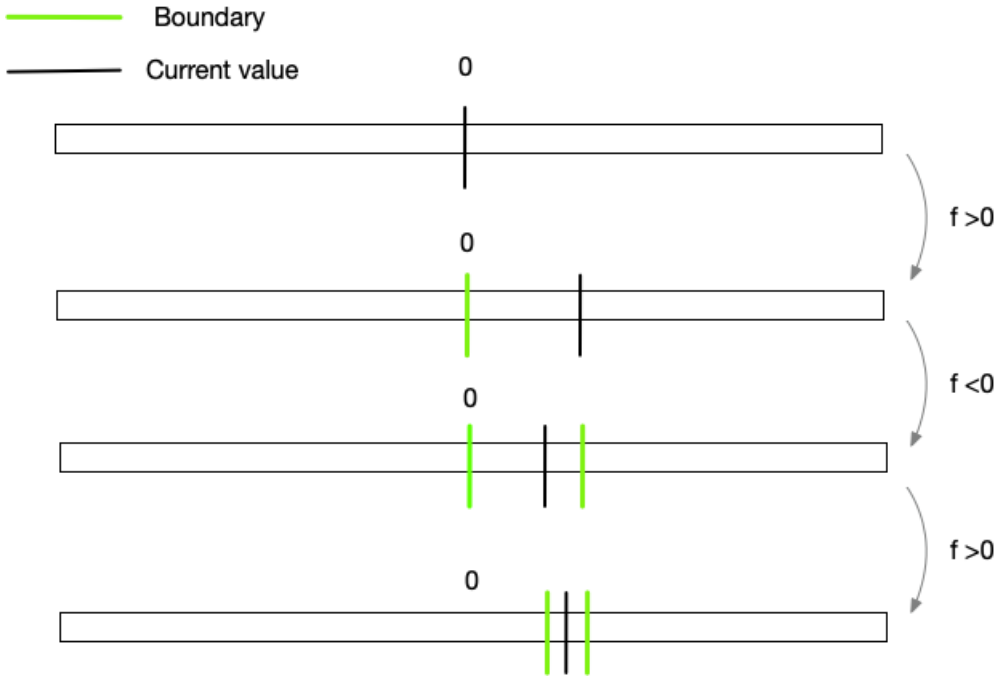

Figure 22: Example of an optimization process of f_c. Modifications to the delay are made based on the sign of the fitness function (f)

5 Evaluation

The use case of "Declarative Test Case generation for autonomous cars" is the automated generation of critical scenarios that contain specification received by the user suited for testing autonomous cars in a simulation environment. The system will be evaluated regarding the scenarios it produces as output and the generation process. The evaluation of the output scenarios focuses on the quality of the test case while the generation process will be evaluated regarding its duration.

5.1 Experimental setup

The system implements three different *critical events*: *Right Departures*, *Front to Rear* crashes and *Two Road Angled* crashes. Each critical event is evaluated individually. For this task input data for every critical event type was semi random generated. This means that inputs were generated randomly but input combinations that can not satisfy the constraints provided by the type of critical event were excluded. Every critical event type is evaluated using the following input files:

- only the critical event type executed 10 times
- 5 inputs with one additional random car property each executed 5 times
- 10 inputs with two additional random car properties each executed 3 times
- 10 inputs with between two and fully specified random car properties each executed 2 times
- 10 fully specified inputs each executed once

This results in five different input configurations(different amount of input values) and 36 total input files for each critical event type and 95 generated scenarios. In case the system received an infeasible input, which means that the predefined trajectory setup could not be followed sufficiently, two test cases have been generated for some of these cases. One with removed predefined flags for that trajectory and one with predefined flags intact but insufficient fitness values (only if the car could reasonably follow the trajectory). So in total at least $3 \cdot 95 = 285$ test cases have been generated. Every traffic participant used the same car model in these test cases in order to keep the generation duration consistent regarding the acceleration of vehicles. All input files used for generation, resulting test cases and a *csv* file with results for each critical event are in the attachments of the thesis. All test case generations were executed on the same hardware, with the same configuration file as input. This ensures, that variations of the execution time only depend on the received user input data and the generation process itself. The configuration file used can be found in the attachments as well. Velocity ranges for random generation were 25 - 100 km/h. The execution state length was set between 25-70 meters and the shape of road interval from -15 to 15 meters. The threshold for trajectory fitness was set at 1 meter, the threshold for velocity fitness at 2 m/s and the threshold for synchronicity fitness at 2 meters. This means, that the simulation had to score fitness values with absolute values below those thresholds in order to be accepted by our system.

5.2 Evaluation of the generation process

The generation process itself was evaluated in terms of generation duration. For that task the setup in 5.1 was used. The duration of every scenario creation was measured and averages for every partition of input data were calculated. Additionally it was evaluated which input values have impact on the duration and which do not.

Data recorded for each test case generation were defined velocities of all vehicles, their velocity difference, as well as iterations of the optimization process, categorized into velocity iteration, trajectory iteration and synchronicity iteration. Last but not least the duration of the generation process was tracked in seconds. This duration includes every step of the generation process as well as every simulation execution needed. Additionally each test case generation was observed personally. Independent of the type of critical event, the number of simulations strongly correlates to the whole duration, because the time needed for abstract- and concrete test case generation is completely overshadowed by the duration that is needed to run one simulation. Hence differences in generation duration are caused by the different amount of simulation executions needed to complete a test case. A simulation is executed each time a test case has been mutated due to the optimization algorithms. Therefore, we can conclude that the amount of optimization iterations needed to complete a test case is the critical factor when it comes to generation duration.

5.2.1 Right Departure

Figure 23: Initial positions

Figure 24: preconditions reached

Figure 25: lane departure

For the right departure critical event 95 test cases were generated. Figures 23 - 25 show key images of an example *RightDeparture* event. The average duration for generating one test case was 39 seconds and 1.25 simulations were needed per test case. As described in section 5.1 there

	Zero	One	Two	Several	Full	Total
Averages	33	33	34	31	92	39
SD	4.03	8.66	19.37	1.93	93.3	36.12

Table 1: averages and standard deviation of the generation duration for the different input categories

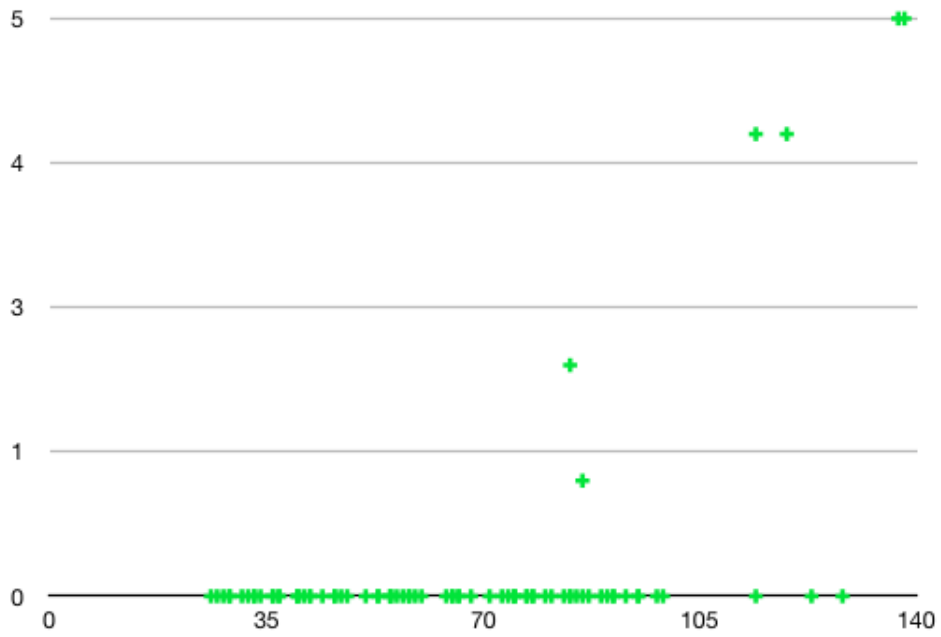

Figure 26: This figure shows the relation between velocity(x-axis) and trajectory optimization iterations needed (y-axis) for lane departure events

are five categories of input data with different amounts of specified input values. Table 1 shows the average generation duration and standard deviation for each category. It is striking that the duration generation is nearly 3 times as long for a fully specified input as for the other ones. With a standard deviation (SD) of 93 for that category we can conclude that this is the reason because single test cases needed a lot longer(longest 258 seconds) to be generated than test cases in the other categories. This happened, because some of these test cases had to be optimized several times regarding their trajectory, because the predefined waypoints were poorly placed (scored bad fitness values). The correlation between the generation duration and trajectory iterations in this partition has a correlation coefficient of 0.96. With 4 of the 10 test cases being optimized regarding their trajectories. While only 6 % of all test cases in the right departure category had to be optimized regarding their trajectory at all. The reason why fully specified inputs had to be optimized more often is a combination of high velocities and predefined waypoints, that the car could not follow. This can be a result of random selection of input values. The low SDs of the other categories suggest that the system can consistently generate lane departure events in about 36 seconds, if no infeasible input was received. The overall correlation between the number of input values and the generation duration is 0.37, which suggests, that there is no relevant correlation. Figure 26 shows, that the trajectory had to be optimized only for test cases

with velocities above 84 km/h, with very few entries above 100 km/h, hence we can conclude that high velocities indeed can increase the generation duration by causing optimization iterations, because it is harder for vehicles to follow their trajectory with high velocities. This can be as well a reason, why the duration was that much longer for the fully specified inputs, because only in that, test cases with velocities above 100 km/h were generated.

5.2.2 Front to Rear

For the 35 input files have been 97 front to rear test cases generated. Figures 27 - 29 show key images of an example *Front to Rear* event. For fully specified inputs, that the system marked

Figure 27: Initial positions

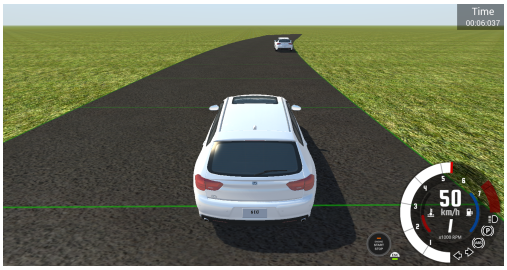

Figure 28: preconditions reached

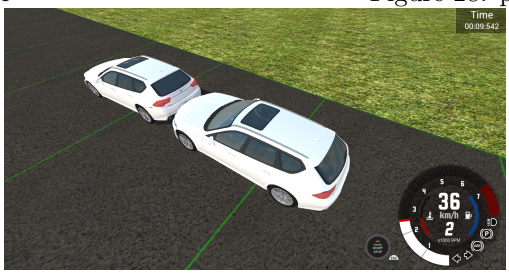

Figure 29: front to rear crash

as infeasible, but were close to their fitness goals, two test cases have been generated. One without breaking up predefined flags and one with breaking them up and allowing the system to further manipulate the test case. This is the reason we have 2 more test cases, than in the right departure category. Out of the 97 generations, the system did not finish on 8 of them. Here the concurrency optimization was caught in an endless loop and the generation process was stopped manually. The reason for that is, that the damage sensor in beamNG did not measure any damage to the vehicles, even though they did crash. This delivered false fitness values to the optimization algorithm. This happened because the difference in velocity between the two vehicles was so low, that there was barely any impact, when they crashed, with the highest Δv being 4 km/h in these cases. When calculating averages for the test case generation, the cases where the system did not finish were not considered. The resulting average generation duration is 204 seconds and 6.8 simulation executions were needed per test case on average. Table 2 shows the average generation durations and SD for the 5 different input categories and in total.

By looking at this table it is noticeable, that the average duration raised, when comparing test cases, that received a higher number of inputs, to test cases, that received a lower number of

	Zero	One	Two	Several	Full	Total
Average	165	155	135	247	293	204
SD	75.07	103.72	100.06	205.12	174.97	145.57

Table 2: Averages and SD of the generation duration of Front to Rear events of the different input categories and in total

	Duration	Velocity Iterations	Trajectory Iterations	Synchronicity iterations
V sum	0,5263	0,5963	0,4187	0,1891
V diff	-0,1145			-0,2657
V max	0,4051	0,5417	0,4007	0,0581

Table 3: Front To Rear Correlations

inputs (zero, one and two). A big factor influencing this is, that the system has less options to optimize the trajectory, if more input values are predefined, because it is not allowed to change predefined values. Additionally, it's optimization algorithm is more limited in the front to rear event, compared to the right departure event, because the position equal constraint is more strict than the position greater constraint, which is used for the right departure event. That results in mutating wp_1, if velocity and curve mutations are not allowed. The mutation of wp_1 has shown to be the least effective optimization operation during test case generation.

Table 3 summarizes correlation coefficients of the sum of both velocities, the max velocity between both cars and the velocity difference to the generation duration and optimization iterations. An assumption is that the test case duration is influenced mostly by the height of velocities and the placement of trajectories. $Vsum$ correlates the strongest to the generation duration, this supports the assumption, that high velocities can lead to longer generation durations. While observing the generation process of each test case, an impact of Δv between both vehicles on the simulation duration seemed noticeable. The correlation coefficient of -0.265 suggests that there is a small correlation, but not really noticeable. But as table 3 shows $Vdiff$ has still the most impact on synchronicity iterations, when compared to the other velocity values. Figure 30 depicts Δv between the cars on the x-axis and the amount of synchronicity iterations needed on the y-axis. In the area from $\Delta v = 0$ to Δv being about 25 km/h an accumulation of points is recognizable. That supports the assumption, that Δv can have an impact on the generation duration, by causing more synchronicity optimization iterations. A reason for this is that, when cars drive in the same direction with similar speed, the striking car needs longer to close the gap to the other car. This makes it more difficult to adjust the timing of the cars so that they crash at the specified location.

Another obvious difference from *front to rear* events compared to *right departure* events is the overall greater generation duration. For test cases, that have more than one car the synchronicity optimization has to be executed, as well as trajectory and velocity optimization for each additional car, that results in at least one more simulation needed per car and one more simulation needed because of synchronicity optimization. Overall it is safe to say, that with a rising number of cars, the test case generation duration will rise as well.

5.2.3 Two Roads

For the *two road angled* crash event 96 test cases have been generated. Figures 31 - 33 show key images of an example *Two Road Crash* event. In two cases the system could not finish the

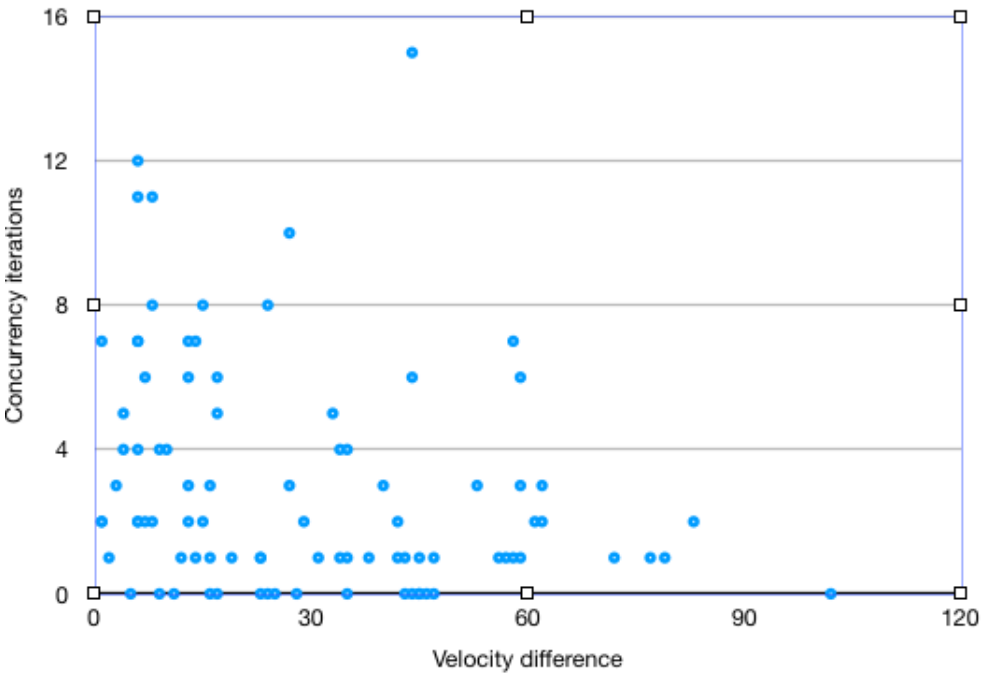

Figure 30: relation of Δv and concurrency iterations

Figure 31: Initial positions

Figure 32: preconditions reached

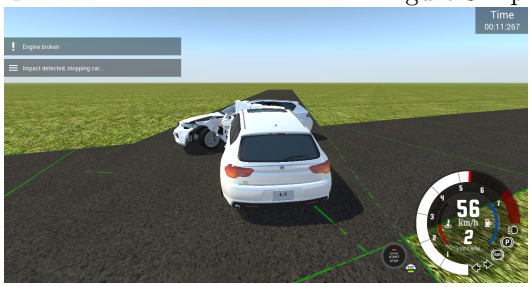

Figure 33: critical event

	Zero	One	Two	Several	Full	Total
Average	129	120	123	142	510	165
SD	49.45	48.05	45.51	107.65	340.9	170.88

Table 4: Averages and SD of the generation duration of Two Road Angled crashes for each category and in total

	Duration	Velocity Iterations	Trajectory Iterations	Concurrency iterations
V sum	0,454	0,564	0,414	0,066
V diff	-0,097			-0,041
V max	0,508	0,572	0,440	0,104

Table 5: Two Road Correlations

generation process. One of them was due to a low $\Delta v = -2$ and both cars driving in the same direction, resulting in the same problem as in the *front to rear* events (no registered damage). The second test case failed to finish had a fully specified input file as input, which resulted in an infeasible scenario setup. The car that was unable to be optimized was the NEC. Its lane for the setup sates was set to 2 and for the execution state to -2. Its velocity is set to 70 km/h. And because of inputs of the EC it only had 15 meters to switch lanes from 2 to -2 during the execution state. This lane switch resulted in a curve too sharp for the car to follow its defined trajectory close enough. The system was not allowed to move wp_2 because it is constraint by inputs of the EC and a method to mutate lane changes or the position of wp_1 in driving direction is not implemented, hence the system could not optimize this case.

Without taking the two failed generations into account the average duration for generating one test case was 165 seconds and 5.8 simulation executions were needed per test case. Table 4 summarizes the durations and SD for the different input categories and in total.

Quite striking here is that the fully specified inputs lead to higher durations. As the SD shows the reason for that is that some test cases needed in comparison to others a way more time. Reason for this is a high number of trajectory iterations with the highest being 38, which resulted in a generation time of 1270 seconds. With only 10 test cases in that category those exceptions have severe impact on the averages. When it comes to the reason for the high number of trajectory iterations, it can be narrowed down to high velocities and poorly placed waypoints in the input files, what makes it very difficult for the vehicle to follow the defined trajectory. Those poorly placed inputs with the slow optimization by mutating only wp_1 of the corresponding execution state lead to many iterations that were necessary until the result was satisfying.

Another thing worth mentioning is, that besides of the duration for the fully specified inputs, the averages are lower, than for the *front to rear* event. This can be explained by the number of synchronicity iterations, that were needed per test case on average. A *front to rear* test case had to be optimized towards synchronicity for 3 times per test case on average, whereas the average for the *tow road* crashes only lies at 1.41 times. Each additional iteration has an additional simulation as consequence. One simulation needs between 20 and 30 seconds to run on average. Hence, the difference in synchronicity iterations can explain the difference in execution times.

For investigating whether the velocity of traffic participants influences the duration of test case generation, the correlation coefficients listed in table 5 have been calculated: Here the difference

in both velocities had no noticeable impact on the duration. The strongest correlations shows $Vmax$, which is the maximum velocity of the traffic participants. The fact, that $Vdiff$ shows barely any correlation suggests, that the velocity of traffic participants has impact on the duration independent on each other. So the faster the cars go, the longer the generation will likely take to finish. The reason for this is not that it is harder to place cars that go faster at the same time at the same location since neither $Vsum$ nor $Vmax$ show any significant correlation to the synchronicity iterations needed. The reason why higher velocities lead to higher generation duration is that it is likely that the length of the acceleration phase needs to be adjusted, because we calculated with a constant acceleration and with higher velocities that is not accurate enough. Additionally cars are less likely to be able to follow their trajectory if they go faster, because they are unable to drive curves as sharp as the ones they can drive with lower velocities.

5.2.4 Summary

To conclude the evaluation on the generation process, we can take away following findings:

1. The amount of simulations needed have the biggest influence on the generation duration. For each optimization mutation one additional simulation needs to be executed. On average 4.6 simulation needed to be executed per test case.
2. The amount of traffic participants elongates the generation process.
3. One test case generation took on average 136 seconds, hence this approach provides a fast way for test case generation.
4. The system did not finish the generation in 10 cases. In 9 of those cases because of a low directional Δv. Hence it is not suited to create Front to Rear crashes with $\Delta v < 4km/h$.
5. Right Departure events were generated the fastest and Front to Rear the slowest.
6. Front to Rear events needed more synchronicity iterations on average than Two Road crashes.
7. Infeasible fully specified inputs result in long generation durations.
8. High velocities lead to longer generation durations, because the system needs more trajectory and velocity optimization steps.
9. Δv has small impact on the duration for Front to Rear events.
10. High velocities ($>$ 125 km/h) lead to velocity optimization steps.

Point 5, 6 and 9 suggest, that the duration of the generation process is also influenced by the traffic participant's direction to each other. Δv showed to have more impact for Front to Rear events, where both cars drive the same direction, than for Two Road crashes. Front to Rear events also needed more concurrency iterations. Hence we can conclude, that it is more difficult to adjust the timing of vehicles if their directional Δv is low.

5.3 Evaluation of the Output Scenario

Test cases generated during the evaluation of the generation process are used to evaluate the qualitiy of the output scenario. The produced scenario is evaluated regarding correctness. Correctness means that the scenario need to be conform to the received inputs. To give an example: A "Front to Rear" crash should lead to a front to rear crash and not to a "Front to Side" crash. Also any other given user input has to be content of the scenario. Further it will be evaluated how well the car is able to run through its trajectory with the demanded velocity and if the critical event occurs at the specified location.

Whether the type of the critical event in the simulation is conform to the critical event of the input is evaluated by manual inspection. In order to evaluate how well the car runs through its trajectory and whether received inputs are present, the position and velocity of the car is tracked every 5 frames during simulation. As a metric on how well the car runs through its trajectory the maximum deviation from its defined trajectory and velocity, during the execution state before the crash, is used. Only data from the execution state is used, because setup states are fully generated by the system itself and their only purpose is to enable the car to meet the preconditions defined by execution state properties. Hence, by running correctly through the execution state the setup states have fulfilled their task and no further evaluation is necessary. As a metric for evaluating the presence of the input in the finished test case, the metric for the presence of the input is incremented by one for each missing input. To evaluate the position of the crash, the distance of the first occurrence of damaged vehicle parts in the simulation to the defined crash location will be used. This results in five metrics used for evaluating the correctness of the scenario:

- Conformance of critical event
- max distance from defined trajectory
- max difference in velocity
- missing input values in finished test case
- distance from crash location (only if a crash is part of the critical event)

If the conformance of the critical event fails the scenario is considered a failure. As an overall metric the sum of the individual metrics will be used. Every metric used for evaluation indicates a good result with low values and a bad result with high values.

5.3.1 Right Departure

The test cases generated were not conform to the critical event in 12 cases, resulting in a 12.6% failure quote. Reason for this is the way the critical event constraints are defined. The constraint responsible for the car to run out of bounds is the following: $wp_2 > r_2 + 2$ in orthogonal driving direction. r_2 is the road waypoint, which lies in the middle of the road, and wp_2 is the traffic participant waypoint. In case the road is broader than 4 meters it can happen that wp_2 is still placed on it. A similar problem arises, in case the road takes a turn. Meaning, if the road does

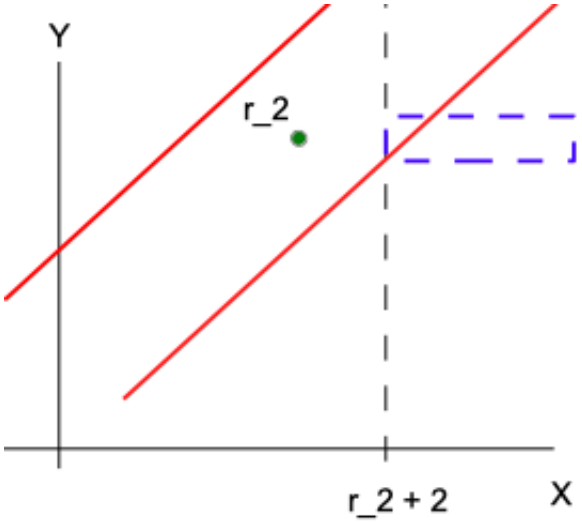

Figure 34: This figure illustrates the problem when placing the waypoint on the right side of a road(red lines are road edges) based on its constraints. The blue rectangle symbolizes the area where the waypoint could be placed.

	Conformance	Trajectory	Velocity	Missing Input	Crash location	Overall
Averages	0.126	0.499	1.511	0.084	-	2.095
SD	-	0.312	1.14	0.3	-	1.50
Worst	fail	1.933	6.0	1	-	8.933

Table 6: Worst scores, average scores and SD for each category of the right departure event.

not align along one axis (Y or X), but goes crooked. The intersect through this road alongside an axis is wider than the width of the road. This is not taken into account by the constraint. Figure 34 illustrates this problem. Here the road is edges are depicted by the red lines. r_2 is the the road waypoint in the middle of the road. To satisfy the constraint wp_2 needs to be placed on the right side of the dashed vertical line. As depicted by the blue dashed rectangle a small area on the road remains as a possible location for wp_2.

Table 6 shows the average scores, worst scores and SD, that were obtained for each category. The averages of the trajectory and velocity scores are both within the set generation thresholds. Together with the small SD we can conclude that sufficient Lane Departure events can be generated with consistent quality.

The overall score of a test case is the sum of trajectory, velocity and missing inputs. All of the worst scores are from the same test case. It was generated using a fully specified input file, that was not feasible, which resulted in removing the predefined flags. This is the reason why one input value is missing in the finished test case. That property was mutated during the generation process.

	Conformance	Trajectory	Velocity	Missing Input	Crash location	Overall
Averages	0.179	0.492	1.510	0.126	4.040	5.982
SD	-	0.24	1.50	0.33	4.03	4.24
Worst	fail	1.207	8.857	1.0	29.816	34.639

Table 7: Average scores, worst scores and SD for each category of the front to rear event.

5.3.2 Front to Rear

The test cases generated with the *Front to Rear* critical event were 16 times not conform to it. Not conform in this case means, that ego car did not hit the rear of the NEC with its front. Compared to the *Right Departure* events, that is a higher number. The cars did indeed crash in every of the 16 cases, the reason they are not conform is, that it was not front to rear. This happened often when both cars did not drive on the same road or same lane. For instance with the use of an input file which had only r_1 of the NEC as an input , 4 out of 5 generated test cases were not conform to the critical event. A predefined r_1 leads to a separate road for the NEC, that caused the crash to be angled and therefore to a front to side crash.

Test preconditions were not reached in one case. This was because a vehicle slowed down during driving through a curve. Hence, it dropped below the required velocity and could not fulfil its preconditions.

Besides that it was not possible to assign a crash location score to each test case, because the damage sensor used to measure the crash location did not track any data when the vehicles had velocities, that are very close to each other.

Table 7 shows the average scores, worst scores and SD, that were obtained for each category: Interesting here is, that in about 12% of all generated test cases an input value is not present. 5 out of the 11 test cases were an input value was missing were generated from full inputs. Here the missing input value is caused by mutation of the infeasible input file. In every of the 6 remaining cases either r_1 or r_2 of the NEC were not conform to the received input. r_2 was not conform, because the system started the generation process falsely with the EC and not the NEC, like it should, when it has r_2 predefined, because of a software bug that has been fixed later. r_1 was not conform, because the system did not generate a separate road for the NEC, this resulted in placing the NEC on the road of the EC and therefore, to a change of r_1. Again a software bug that was fixed later was the reason for that. The worst overall score is caused by the worst crash location score. The crash location score with 29.8 is outstandingly bad. The SD of 4.03 of the Crash location suggests that this is rather an exception than the norm.

5.3.3 Two Road

The generated *Two Road* crashes were not conform to the critical event in 5 cases. There were no requirements besides that the cars have to crash, hence, the cars missed each other in 5 cases. When looking at the test cases, where the cars missed, it can be noticed, that they are very close to each other. Meaning the timing would need to be adjusted. Re executing those test cases did lead to a crash in some of them. This shows that the test execution itself is not totally consistent.

	Conformance	Trajectory	Velocity	Missing Input	Crash location	Overall
Averages	0.053	0.428	1.101	0.212	2.191	3.817
SD	-	0.28	1.14	0.62	1.08	1.58
Worst	fail	2.224	4.787	3.0	5.510	9.537

Table 8: average scores, worst scores and SD of each category for the Two Road Angled crash event

The preconditions could not be reached in one case for the same reason as in the front to rear test case, where the car slowed down in a curve.

Table 8 summarizes the average scores, worst scores and SD for each category. One can notice that the amount of missing input values is higher on average, compared to the other two critical events. Reason for that is, that the system overrides the car's direction, if waypoints get placed contradicting to it. This happened often, if a direction as well as wp_1 was predefined for the NEC. It'swp_2 had to be placed at the location of the EC's wp_2 due to constraints. This resulted in a different direction than specified. On top of that the number of fully specified inputs, that were infeasible was higher than for the other test cases.

5.3.4 Summary

After every critical event was looked at individually, we compare them to each other in terms of their scores. Table 9 summarizes all average scores across each evaluated critical event. Best overall scores were reached by the *Right Departure* test cases, but this is only because, they have no crash location score assigned since there is no crash. If the crash location score is neglected, the *Two Road* crashes obtained the best scores followed by the *Front to Rear* crashes. The reason why trajectory and velocity scores are worse for the *Right Departure* event is, that curves are likely to be sharper. It can be noticed, that the system struggles in regards to the timing of *Front to Rear* crashes. This correlates to the findings of the evaluation of the generation process, where *Front to Rear* events needed the most concurrency iterations. The system succeeds in generating trajectories, which can be followed by the traffic participants. Average scores for trajectory are throughout way below the threshold of 1 meter, with the worst overall trajectory score being 2.224 meters. The system does succeed in generating test cases, that's preconditions can be fulfilled in every test case generated for the evaluation except for two.

Striking is, that the Crash location score does not lie under the synchronicity threshold towards which the test cases were optimized. This threshold was at 2 meters, but both average crash location scores are worse than that. This has two reason:

1. For optimization, the position of the NEC was used and not the position of the EC.
2. Test executions do not run consistent and small deviations of the start of the NEC movements can cause big changes on the crash location.

Regarding point 1: Even if the test generation did finish with a perfect score on the concurrency fitness, the EC itself is likely to be at least 1-2 meters away from its wp_2 (crash location), because

	Conformance	Trajectory	Velocity	Missing input	Crash location	Overall
Right Departure	0.126	0.499	1.511	0.084	-	2.095
Front to Rear	0.179	0.492	1.510	0.126	4.040	5.982
Two Road	0.053	0.428	1.101	0.212	2.191	3.817

Table 9: Average score of each category for the according critical event

the position optimized was the position of the NEC. It is physically impossible for both cars to be at the exact same location, hence this causes insufficiencies in the crash location score.

Regarding point 2: Test execution and test generation do not run 100% deterministic. Reason for that is, that the state of the car during simulation is received every 5 steps using the function *bng.step(5)* of the beamNgpy api and not continuously. Hence, the information present to decide whether a car should start its movement and to calculate fitness values can vary from execution to execution. This causes inconsistencies between each test case execution and the final simulation of the test case generation process. When comparing the conformance it shows that these inconsistencies have little impact, when both cars are simply supposed to crash, but can cause insufficient test cases, when specific parts of cars have to crash into each other. In summary one can say, that the system is suited for generating critical events, that should result in a car crash, but if the car crash is supposed to happen at an exact position or specific car parts are supposed to hit in each other results are less sufficient.

Another problem worth noticing is, that the cars do not follow their trajectory consistently with the specified velocity. This was mentioned before, because it caused in two test cases, that preconditions could not be met. Figure 35 shows all velocity scores of the Front to Rear event as entries in the diagram. The fitness goal is marked with the black line at 2 km/h. During test case generation it was optimized, that the car enters the execution state with the defined velocity. The values shown in figure 35 have all been measured after the execution state was entered. Therefore, the car has lost velocity during the execution state, which resulted in bad evaluation scores. Preconditions can still be reached consistently, hence, this is not necessarily a problem. In theory the trajectory optimization can be modified to enable the vehicles to run through their trajectory without losing speed. A drawback to this would be that test cases lose variety, because more trajectories with flatter curves would be resulting.

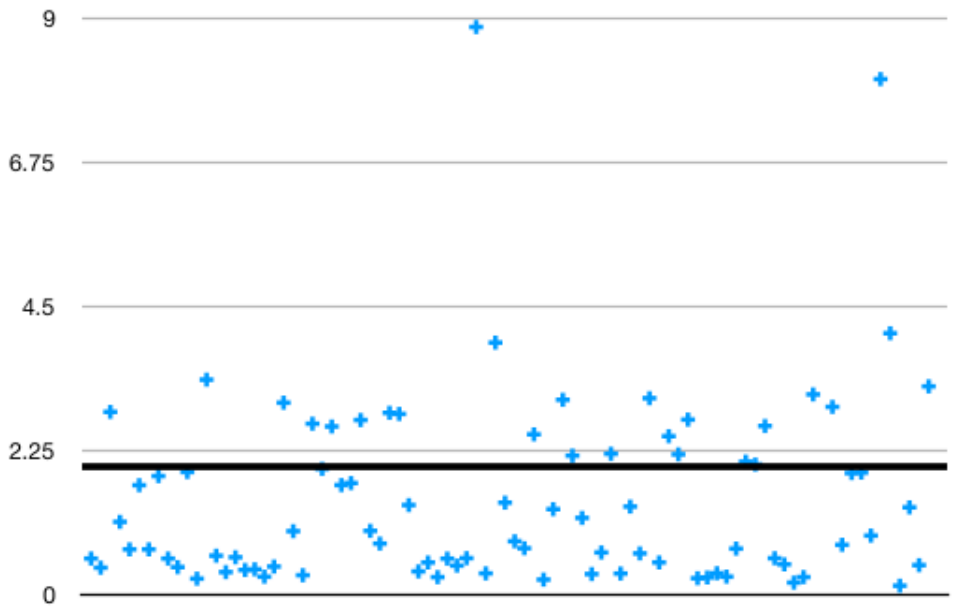

Figure 35: Velocity score of the front to rear event. Threshold marked at 2 km/h

6 Conclusions and Future Work

This thesis presented an approach for the automatic generation of critical test cases based on partial specification from the user. User inputs specify criteria in form of values for test case properties, that are content of the finished test case. A prototype system was implemented, that can generate *Lane Departure* and *Front to Rear*- and *Two Road* crash events. The evaluation of this system showed that it can automatically create test cases that meet the user partial specifications in minutes most of the time. Additionally it tells the user when an input that does not satisfy the critical event or specifies infeasible trajectories was entered. This section concludes the contribution of this thesis to the problem of time consuming test case generation for autonomous cars and discusses what improvements can be made to the system based on the results of the evaluation and what future work is possible based on this system.

6.1 Conclusion

Autonomous cars will be part of everyday traffic in the future and are to a small degree already present on public roads. With human life being at risk if software malfunctions, extensive software testing is a necessity. Self driving cars need to be able to handle critical driving situations. Here the problem of feasible test case generation and execution arises. With the infinite amount of scenarios an AI needs to handle to navigate through traffic manual test generation and relying only on real world tests becomes infeasible. In addition to that test case generation itself remains a time consuming factor. This thesis tackled this problem by presenting an approach for

automatic test case generation suited to be executed in a simulation environment. The tests generated put the system under test into critical driving situations, where an accident will happen without intervention. These tests are generated in a declarative manner, meaning the system generates scenarios that complete the user input. This enables the generation of diverse driving scenarios that all test the capability of the system under test of handling the situation defined by the user specification.

First a test case model was introduced, that organizes the test case properties into setup- and execution states. Using this model the generation process was divided into small steps and values of test case properties, that were not part of the received input, were automatically generated. For that task we applied constraints to the range of possible values for unset properties and picked a random value from the remaining ones. The resulting test case was used as a starting point for local search algorithms, that optimized the test case toward its fitness goals. Fitness goals were, that cars can run through their trajectory, reach the specified velocity and reach synchronous positions with the correct timing. A prototype software that supports three different types of critical events was implemented. The test cases and their generation process was evaluated individually for each type and eventually a comparison was done. Findings are, that the system succeeds in generating scenario set ups, that enable vehicles to reach test preconditions. Vehicles sufficiently follow their trajectory and demanded velocities get reached. 88.9% of generated test cases resulted in the specified critical event. The presented approach showed insufficiencies when it comes to timing of vehicles, due to inconsistent test executions. The evaluation also showed, that the system keeps the resulting test case close to the received input, even if that input was infeasible. The system needed on average about 136 seconds to finish one generation process, hence the proposed approach provides a fast way for automated test case generation.

Overall it can be concluded, that the proposed method of test case generation provides a fast way to generate test cases, that comply to the user input and can be executed in a simulation environment. Hence, this thesis provides a contribution to solve the problem of time consuming test case generation. This allows to concentrate rather on finding challenging scenario setups for the system under test, than on worrying about feasibility or generating those scenarios.

6.2 Future work

Future work will be separated in two section. First the proposed approach can be optimized based on the findings of the evaluation and second new future research is now possible based on this method of automated test case generation.

6.2.1 Improvements to the system

Improvements to the system can be made in two regards. The generation process can be sped up and the quality of the resulting test cases can be improved.

The generation duration is dependent on the amount of simulation the system needs to sufficiently generate a test case. The optimization of trajectory and velocity of traffic participants is executed one at a time. This leads to a increase of duration per traffic participant. This method was chosen, because trajectories have to be optimized without interventions of other vehicles. In theory both cars can be optimized at once, if intersecting trajectories are shifted apart from each other, without changing the trajectories itself.

Another factor influencing the amount of simulations to be executed are the iterations needed to optimize the movement of traffic participants. While the concurrent optimization of several traffic participants has no impact on the resulting test case, changes to the optimization process itself will likely have an impact on it. The search algorithm is only looking at close neighbours. It changes one property value at a time right now. It can be changed to make bigger changes to that property in order to reach a sufficient state faster or it can change two or more properties at a time, if allowed. This results in the search algorithm checking neighbours further away in the search space, hence the resulting test case is likely to more diverse to the received inputs as in the current implementation.

An obvious approach to speeding up the generation is to improve the quality of the initial test case, that is used as the starting point for the optimization. A better starting point results in less needed optimization iterations. The velocity of the cars are not paid attention to, when the trajectory is generated. If insufficient trajectories can be out of the question in advance, the overall generation process can be sped up. There are physic formulae for calculating the forces of the traction of a car's wheels and the street, dependent on velocity, road surface and the degree of the curve [12]. Using these formulae one can approximate the behaviour of the vehicles and estimate the likelihood of generating the expected test cases. This assumes that these formulae also apply to the simulation environment used. The initial calculation of the trigger can be improved by using actual measured time the car needs to fully run through its trajectory instead of using calculated duration values. These durations can easily be measured during trajectory and velocity optimization.

The qualitiy of the test cases can be improved by either optimizing the generation of the initial test case used as a starting point for the search or by improving the search itself. The evaluation showed that the trajectory optimization scored sufficient results. Velocity scores were often not in the the sufficient threshold, because the vehicles slowed down in curves they couldn't take with their current velocity. Despite that, preconditions were reached for 276 out of the 278 generated ones, resulting in a failure rate of 0.71%. The quality on the velocity score can be improved, if the velocity during the execution state (when the vehicle drives on the curve) is incorporated in the trajectory search algorithm. This will come at the cost of diverse test cases, because the solution space becomes more narrow by eliminating solutions, that would fulfil all preconditions and would deliver sufficient fitness values otherwise.

The biggest point of improvement is the crash location or concurrency optimization, as the evaluation showed, that this is the most lacking part. One can assume that a problem during generation as well as execution is the inconsistency with which the scenarios are simulated, further described in the evaluation section. The system accepts a solution for the concurrency optimization if the goal is reached in the current iteration. It does not re execute this solution to confirm, that the goal can be reached consistently. The fact from which this whole problem arises is that the system only receives a discrete set of positions of the EC to decide when the

NECs are supposed to start their movement. For instance, if known positions in driving direction of the EC are 20 and the consecutive one to that is 25, but the NEC should start its movement at 21 the NEC starts its movement too late because it waits until the EC passed 21. This results in the NEC waiting too long. It needs to be investigated, whether a narrow step size leads to more consistent simulations (step size used during evaluation was 5). Another solution to this problem could be, making the trigger not position but time dependent, assuming the EC needs every simulation execution the same amount of time to reach certain spots. In this case the trigger would be represented by a time delay an NEC needs to wait until it starts moving through its trajectory.

A limitation that was exposed by the evaluation of the current implementation of the approach is that the prototype can not mutate the lanes on which the vehicles drive. This mutation operation is very simple to implement. The recommended implementation is to just move the setup state lane closer to the execution state lane, without making changes to the execution state lane.

The Evaluation showed, that the optimization algorithm could not finish the generation in cases with $\Delta v < 3$ km/h. This was because the damage sensor used to detect the collision did not measure the impact of the vehicles. A solution to that problem is, that the search algorithm is modified to not rely on the damage sensor or that crashes demand a directionalΔv of at least 4 km/h.

6.2.2 Future research

The system for automatic test case generation implemented for this thesis is rather a prototype than a finished product, hence future work can extend the system in functionality. The easiest way to extend the system is adding more critical events, by providing new sets of constraints. Additionally road generation can be upgraded. For instance allowing a road to be defined by more than one vehicle, meaning not that a road can be defined by several vehicles at once, but if the trajectory of one vehicle ends another one can continue to define the road further. Another possibility to generate more realistic road networks is to merge two roads. This can be easily accomplished by cancelling the road defining state of one vehicle, when it enters the road of another vehicle.

This thesis introduced a model for representing driving scenarios in form of execution and set up states. While the system generated for the thesis only supports one execution state per test case this can be extended for future work to generate complex driving tasks that include more than one critical event in the test case.

The method proposed by this thesis provides a fast way to generate test cases with specific inputs. This input needs to be generated by hand. A random input generator was used for generating the inputs for the evaluation, but these were often infeasible especially when fully specified. A field for future work can be to automatically generate inputs, that can be used by our system to generate test cases. Inputs can be generated without needing to execute a simulation, because optimization is done by this system itself, while it tries to keep the result as close as possible to the input. For generating these inputs for a test suite search can be used with different goals. Here search for novelty [38] [29] [30] can be applied to generate diverse test suites or search for criticality for instance with limiting the solution space [23] to find test configurations that are challenging for the system under test. Additionally the search for inputs can be guided to expose failures in autonomous driving software.

A Mathematical operations

This section will explain some phrases that are frequently used in the thesis and their accompanying mathematical operations.

Get point on road Get point on a road refers to the operations executed to interpolate a point (wp_c) on a bezier curve between two waypoints (wp_1, wp_2), with a known coordinate in driving direction. Hence, only the coordinate in orthogonal driving direction has to be found. As a reminder of section 2.5, the following is the equation for the quadratic bezier curve:

$$B(t) = (1-t)^2 \cdot p_0 + 2 \cdot (1-t) \cdot t \cdot p_1 + t^2 \cdot p_2 \tag{23}$$

To find a coordinate pair that lies on that curve we have to determine the corresponding t first. For that task the coordinate in driving direction, that we already know, is filled into equation 23. The whole calculation process is demonstrated with Y as the driving direction. The Y coordinate of wp_1 is filled into equation 23 with $B(t) = y$. Start coordinate for the bezier curve is the y coordinate of wp_1 (y_{wp_1}). End coordinate is the y coordinate of wp_2 (y_{wp_2}). The control point is the y coordinate of the road bezier control point (y_b). Filling those into equation 23 results in the following:

$$y = (1-t)^2 \cdot y_{r_1} + 2 \cdot (1-t) \cdot t \cdot y_b + t^2 \cdot y_{wp_2} \tag{24}$$

This equation is solved for t, which results in the following equation:

$$t = \frac{-(2 \cdot y_b - 2 \cdot y_{r_1}) \pm \sqrt{(2 * y_b - 2 * y_{r_1})^2 - 4 \cdot (y_{wp_2} + y_{r_1} - 2 \cdot y_b) \cdot (y_{r_1} - y)))}}{(2 \cdot ((y_{wp_2} + y_{r_1} - 2 \cdot y_b)))} \tag{25}$$

Now we are able to calculate the missing x coordinate of wp_c by using the quadratic bezier equation 23 with the same start, end and control points like when solving for t, but using their x coordinates and the calculated value for t . This results in the following equation:

$$x = (1-t)^2 \cdot x_{r_1} + 2 \cdot (1-t) \cdot t \cdot x_b + t^2 \cdot x_{wp_2} \tag{26}$$

Shift to its lane *Shift to its lane* refers to a waypoint of a trajectory, that is shifted orthogonal to a trajectory dependent on the lane the associated traffic participant drives on. Waypoints that are shifted are traffic participant waypoints or road waypoints, dependent on the trajectory that is present. The shift length l is dependent on the lane the car drives on. If that lane is 0, l is set to 0. In other cases l is defined by the following equations:

$$l = laneSign \cdot \frac{partLength}{2} + (lane - laneSign) \cdot partLength \tag{27}$$

$$partLength = \frac{roadwidth}{numberoflanes} \tag{28}$$

The *laneSign* in these equations is the sign of the lane the traffic participant drives on. Lanes on the left side have a negative sign and lanes on the right side a positive one.When shifting the waypoint to its lane, it is distinguished in two cases.

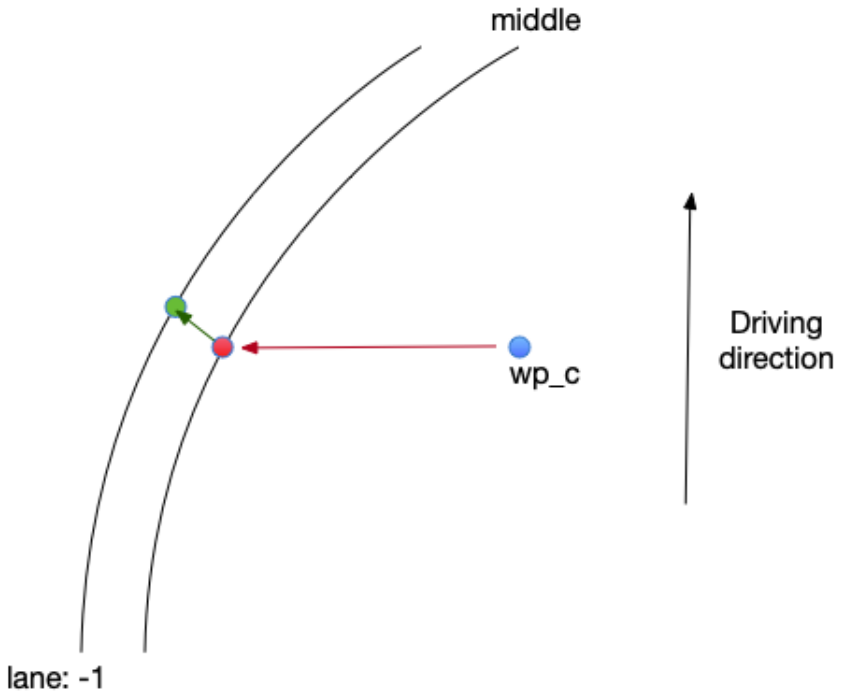

Figure 36: depicting the get point on road (red) and shift to its lane (green) process

First, the waypoint $toShift$ lies before the first waypoint of the execution state or is equal to it in driving direction. In this scenario the way $toShift$ can be moved by l in orthogonal driving direction.

Second, $toShift$ lies between the waypoints according to whose trajectory it is shifted (wp_1 and wp_2). Now the value for t for the coordinate in driving direction of $toShift$ of the bezier curve between wp_1 and wp_2 is determined. Now the values of the derivative of the bezier curve in point t can be obtained. After that $toShift$ is moved by l orthogonal to the vector that is represented by the derivative.

Figure 36 depicts the actions taken for the *get point on road* and *shift to its lane* process. The arrow in red summarizes the get point on road. Here the waypoint is moved in orthogonal driving direction onto a bezier curve. Shift to its lane (depicted in green) moves a waypoint from the middle of the road to another lane (lane -1 in the figure).

Calculating the bezier control point For modelling curves of trajectories bezier curves are used as described in section 2.5. With using a third waypoint (bezier control point b) the shape of the curve between to other waypoints can be described. Each time those other two waypoints have been set, b can be determined accordingly. The vector $\overrightarrow{wp_1 b}$ is a tangent to the bezier curve in wp_1 and the vector $\overrightarrow{bwp_2}$ is a tangent to the bezier curve in wp_2. For setting b it is assumed that the traffic participant travels on a straight line up to wp_1 in driving direction. Hence, to avoid a kink in its trajectory the coordinate in orthogonal driving direction of b is set equal to the

one of wp_1. This guarantees a smooth transition from the straight line to the curved trajectory. The coordinate of b in driving direction is set to a fraction of the distance between wp_1 and wp_2. This fraction is received from the configuration file received as input.

B Executing the System

This section will give a brief introduction on what is needed to run the system and how to execute a test generation.

Prerequisites for running the system are a bemanNg.research_unlimited installation, Java 8 and a python compiler with the beamNgpy api installed. The configuration file described in section 3.3 has to be placed in the same folder as the java application for test generation. In this configuration file the path to the beamNg trunk, the python compiler, that uses the beamNgpy library and the input file has to be specified. In addition the name of the beamNg scenario in which the test should be executed needs to be specified. Keep in mind, that the system has no information about objects already present in that scenario, hence it is recommended, that an empty one like *smallgrid* is used.

For running the system, generation values from which it can pick random values, if no predefined input was received need to be set in the configuration file. There is the option to launch beamNg with running the system (enter *true* to open it or *false* in this field). This showed some troubles, hence it is recommended to launch beamNg in before executing a test generation. Therefore, the python script *launchBeamNg* needs to be executed beforehand. This script is also attached to the thesis. BeamNg does not show roads, when using the default smallgird scenario in its current state, if these roads are generated within the simulated python script. To make roads visible, the *overobjects* flag in the scenarios prefab file, which is generated upon launch needs to be set. For the evaluation a modified version of the *smallgrid* scenario, that can show roads was used. The modified version of *smallgrid* is also attached to the thesis.

While running the system will tell you before the first simulation is executed if the received input was not conform to the constraints provided by the *type of critical event*, but will continue the generation process regardless. During optimization process, it can happen that the system can not perform any mutations on the test case to create a feasible trajectory. When this happens it will ask whether predefined flags should be removed for that vehicle or if they should be kept. Here the user has two options to enter with *false*, predefined flags stay and the generation process continues with the next step. Entering *true* will remove predefined flags and the current traffic participant is optimized until its fitness goals are reached.

C Software

On this CD is the source code of the implemented software as well as input files used for the evaluation and csv files with the tracked data during the generation process.

References

[1] About mmucc. https://www.nhtsa.gov/mmucc-1. Accessed: 11.2.2020.

[2] Autonomes fahren: Digital entspannt in die zukunft. https://www.adac.de/rund-ums-fahrzeug/ausstattung-technik-zubehoer/autonomes-fahren/technik-vernetzung/aktuelle-technik/. Accessed: 12.12.2019.

[3] Beamngpy. https://github.com/BeamNG/BeamNGpy. Accessed: 13.2.2020.

[4] Binary search. https://www.geeksforgeeks.org/binary-search/. Accessed: 13.2.2020.

[5] Commonroad. https://commonroad.in.tum.de/. Accessed: 4.2.2019.

[6] Die db ist vorreiter beim autonomen fahren. https://www.deutschebahn.com/de/Digitalisierung/technologie/Neue-Technologien/Die-DB-ist-Vorreiter-beim-autonomen-Fahren--3376636. Accessed: 12.12.2019.

[7] Dieses muster treibt autonome autos in den wahnsinn. https://www.welt.de/wissenschaft/article202616258/Autonome-Autos-Ein-buntes-Muster-legt-ihr-Gehirn-lahm.html. Accessed: 12.12.2019.

[8] The explosion of the ariane 5. http://www-users.math.umn.edu/~arnold/disasters/ariane.html. Accessed: 28.10.2018.

[9] Finite state machines. https://brilliant.org/wiki/finite-state-machines/#formal-definition. Accessed: 25.12.2019.

[10] Google reports self-driving car mistakes: 272 failures and 13 near misses. https://www.theguardian.com/technology/2016/jan/12/google-self-driving-cars-mistakes-data-reports. Accessed: 13.12.2019.

[11] How to calculate time and distance from acceleration and velocity. https://www.dummies.com/education/science/physics/how-to-calculate-time-and-distance-from-acceleration-and-velocity/. Accessed: 14.2.2020.

[12] Kurvenfahrten. https://physikunterricht-online.de/jahrgang-10/kurvenfahrten-zentripetalkraft/. Accessed: 26.1.2020.

[13] Local search methods. https://www.sciencedirect.com/topics/computer-science/local-search-method. Accessed: 25.1.2020.

[14] Nhtsa crash viewer. https://crashviewer.nhtsa.dot.gov/LegacyCDS/Search. Accessed: 5.11.2018.

[15] Opendrive. http://www.opendrive.org/index.html. Accessed: 4.2.2019.

[16] Preconditions for successful testing. https://reqtest.com/testing-blog/preconditions-for-successful-testing/. Accessed: 09.2.2020.

[17] A primer on bézier curves. https://pomax.github.io/bezierinfo/. Accessed: 10.09.2019.

[18] Road deaths in germany fall to all time low but accidents on the rise. https://www.dw.com/en/road-deaths-in-germany-fall-to-all-time-low-but-accidents-on-the-rise/a-37703282. Accessed: 28.10.2018.

[19] Self-driving car: Levels, benefits and constraints. https://mobility.here.com/self-driving-car-levels-benefits-and-constraints#pgid-2541. Accessed: 10.2.2020.

[20] The self-driving car timeline – predictions from the top 11 global automakers. https://www.techemergence.com/self-driving-car-timeline-themselves-top-11-automakers/. Accessed: 28.10.2018.

[21] Self-driving uber kills arizona woman in first fatal crash involving pedestrian. https://www.theguardian.com/technology/2018/mar/19/uber-self-driving-car-kills-woman-arizona-tempe. Accessed: 28.10.2018.

[22] Uber crash shows 'catastrophic failure' of self-driving technology, experts say. https://www.theguardian.com/technology/2018/mar/22/self-driving-car-uber-death-woman-failure-fatal-crash-arizona. Accessed: 13.12.2019.

[23] Matthias Althoff and Sebastian Lutz. Automatic generation of safety-critical test scenarios for collision avoidance of road vehicles. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 1326–1333. IEEE, 2018.

[24] Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand. Black-box system testing of real-time embedded systems using random and search-based testing. In *IFIP International Conference on Testing Software and Systems*, pages 95–110. Springer, 2010.

[25] Aitor Arrieta, Shuai Wang, Urtzi Markiegi, Goiuria Sagardui, and Leire Etxeberria. Search-based test case generation for cyber-physical systems. In *Evolutionary Computation (CEC), 2017 IEEE Congress on*, pages 688–697. IEEE, 2017.

[26] Aitor Arrieta, Shuai Wang, Goiuria Sagardui, and Leire Etxeberria. Test case prioritization of configurable cyber-physical systems with weight-based search algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pages 1053–1060. ACM, 2016.

[27] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5):507–525, 2014.

[28] BeamNG GmbH. BeamNG.research. https://www.beamng.gmbh/research.

[29] Mohamed Boussaa, Olivier Barais, Gerson Sunyé, and Benoit Baudry. A novelty search approach for automatic test data generation. In *Proceedings of the Eighth International Workshop on Search-Based Software Testing*, pages 40–43. IEEE Press, 2015.

[30] Mohamed Boussaa, Olivier Barais, Gerson Sunye, and Benoit Baudry. A novelty search-based test data generator for object-oriented programs. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 1359–1360. ACM, 2015.

[31] Lionel Briand, Shiva Nejati, Mehrdad Sabetzadeh, and Domenico Bianculli. Testing the untestable: model testing of complex software-intensive systems. In *Proceedings of the 38th international conference on software engineering companion*, pages 789–792. ACM, 2016.

[32] Kwang-Ting Cheng and Avinash S Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *Design Automation, 1993. 30th Conference on*, pages 86–91. IEEE, 1993.

[33] Alessio Gambi, Tri Huynh, and Gordon Fraser. Generating effective test cases for self-driving cars from police reports. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 257–267, 2019.

[34] Alessio Gambi, Marc Mueller, and Gordon Fraser. Automatically testing self-driving cars with search-based procedural content generation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 318–328, 2019.

[35] Florian Hauer, Alexander Pretschner, and Bernd Holzmüller. Fitness functions for testing automated and autonomous driving systems. In *International Conference on Computer Safety, Reliability, and Security*, pages 69–84. Springer, 2019.

[36] WuLing Huang, Kunfeng Wang, Yisheng Lv, and FengHua Zhu. Autonomous vehicles testing methods review. In *Intelligent Transportation Systems (ITSC), 2016 IEEE 19th International Conference on*, pages 163–168. IEEE, 2016.

[37] Rakesh Kumar, Surjeet Singh, and Girdhar Gopal. Automatic test case generation using genetic algorithm. *International Journal of Scientific & Engineering Research (IJSER)*, 4(6):1135–1141, 2013.

[38] Joel Lehman and Kenneth O Stanley. Exploiting open-endedness to solve problems through the search for novelty. In *ALIFE*, pages 329–336, 2008.

[39] John W Lloyd. Practical advtanages of declarative programming. In *GULP-PRODE (1)*, pages 18–30, 1994.

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

[41] Galen E Mullins, Paul G Stankiewicz, and Satyandra K Gupta. Automated generation of diverse and challenging scenarios for test and evaluation of autonomous vehicles. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1443–1450. IEEE, 2017.

[42] Andreas Tamke, Thao Dang, and Gabi Breuel. A flexible method for criticality assessment in driver assistance systems. In *Intelligent Vehicles Symposium (IV), 2011 IEEE*, pages 697–702. IEEE, 2011.

D Eidesstattliche Erklärung

Hiermit versichere ich, dass ich diese Masterarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie, dass ich die Masterarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Passau,

Johannes Müller