




PASSAU UNIVERSITY

MASTER THESIS

DriveBuild
Automation of
Simulation-based Testing
of Autonomous Vehicles

Author:

 Stefan HUBER

Supervisor:

Prof. Dr.-Ing. Gordon FRASER

Advisor:

Alessio GAMBI, Ph.D.

Thursday 16th January, 2020

Abstract

Simulation based testing is the most common technique for testing autonomous vehicles (AVs). For each test a tester needs to describe a scenario, specify test criteria, setup a simulation, connect the artificial intelligences (AIs) under test to it, execute the test, determine its results and collect all generated data e.g. for further analysis or training AIs. This process is tedious and error prone. There is no well-established procedure how to cope with or solve these problems. I present DRIVEBUILD, a research toolkit for simulation based testing of AVs. DRIVEBUILD comes with an abstract scheme to describe tests and provides a scalable client-server-architecture based on micro services. DRIVEBUILD is able to execute automatically generated tests and to connect AIs under test which control AVs in a simulation. It also offers many metrics to analyze AVs and test generators. This thesis shows that DRIVEBUILD automates the process of setting up simulators, distributing test runs across a cluster, frequently checking test criteria during a simulation, gathering data and analyzing test results. So it reduces the amount of time which a tester needs to invest into preparing, running and evaluating simulation based tests. There are already students, courses as well as research groups that are interested in DRIVEBUILD and use it for their own purpose.

Contents

1	Introduction and Motivation	4
2	Problem Statement	4
3	Background	6
4	State of the Art	7
4.1	Formalization of Environments and Criteria	7
4.2	Generation of Environments	9
4.3	Simulators	10
4.4	Simulation Infrastructure and Platforms	11
4.5	Comprehensive Approaches	11
4.6	Implementation of AIs	12
5	Methodology	13
5.1	Test Case Formalization	13
5.1.1	Formalization of Environments	15
5.1.2	Formalization of Criteria	17
5.1.3	Formalization of Participants	18
5.2	Test Life Cycle	19
5.3	Runtime Verification	19
5.4	Cluster Architecture	21
6	Implementation	24
6.1	Used Tools, Frameworks and Libraries	24
6.2	Communication	25
6.3	MAINAPP	26
6.4	SIMNODE	30
6.5	DBMS	36

7	Evaluation	36
7.1	Experimental Settings	38
7.1.1	Seminar	38
7.1.2	Submissions	38
7.1.3	Technical Specifications	39
7.2	Evaluation of Generality	39
7.2.1	Challenge Test	40
7.2.2	Interfaces for Challenge Test	41
7.2.3	Feature Coverage	41
7.2.4	Efficiency of Test Generators	44
7.2.5	Complexity of Test Cases	44
7.2.6	Effectiveness of Test Generators	51
7.3	Evaluation of Scalability	53
7.4	Experience Report	57
7.4.1	Process	57
7.4.2	Experience	57
8	Conclusions	59
9	Future Work	59
10	Acronyms	61
11	References	62
A	Appendix	67
A.1	Code Example Snippets	67
A.2	Determine Target Position for A0	70
A.3	Used Tools	70

1 Introduction and Motivation

The progress in developing autonomous vehicles (AVs) over the past years is impressive and the effort taken for testing them is amazing. The Cruise program of general motors (GM) drove over 1 million miles [1], Uber drove over 3 million miles [2] and cars of Googles Waymo project even drove over 10 million miles autonomously on public roads. Additionally Waymo drove over 7 billion miles in simulations [3]. However, many more miles have to be driven autonomously since the process of testing and training AVs requires AVs to drive hundreds of millions of miles autonomously [4] to assure a high reliability on their safety. This results in an increasing importance of simulations ([5, 6]). To simulate AVs instead of driving real AVs on public roads allows to drive many more miles within a certain time interval, avoids accidents and injuries, vastly reduces the costs of testing, allows to test AVs in predefined situations and enables testers to reproduce test results and faulty behaviors, i.e. to debug AVs. The setup of simulations and the interaction with AVs in a simulation are complex which makes simulation based testing tedious and error prone. There is currently no well known scheme of abstractly specifying test criteria in the context of AVs and no well known software architecture which is geared towards simulation based testing of AVs. Furthermore there is at the moment no tool that provides an abstract interface for testing and training AVs as well as for supporting test generation.

I present DRIVEBUILD, a research toolkit that automates the process of setting up simulations, executing tests on a cluster, verifying test criteria during simulation time, determining test results and collecting training data. This automation avoids dealing manually with tedious and error prone tasks. DRIVEBUILD implements a unified process to test AVs and to collect training for them which reduces the required effort for testing AVs and gathering training data. DRIVEBUILD provides a declarative and extensible extensible markup language (XML) based domain specific language (DSL) to formalize test cases which allows to verify test cases and reuse test definitions. This work does a critical discussion of the domain and presents a user study to reveal the main requirements of the DSL. The work contains an extensive evaluation that shows the generality and the scalability of DRIVEBUILD. This evaluation develops a scheme to test AVs against test generators.

The thesis is organized as follows: Section 2 introduces detailed descriptions about the problems this work aims to solve followed by Section 3 which explains the basic concepts this work utilizes or orients on and Section 4 which discusses current and related approaches. Section 5 raises the applied methods and strategies, Section 6 describes the implementation in detail and Section 7 shows the capabilities of the test formalization provides, the metrics it offers, analyzes the scalability and discusses how supportive DRIVEBUILD is for testers.

2 Problem Statement

In this work a test case is a specification of an environment and a test setup. The environment describes the curvatures of roads and the placements of static obstacles. The test setup describes the initial states of vehicles and the test criteria. The resulting number of possible test cases is too huge to create a systematic way to define test cases i.e. a formalization that allows to describe arbitrary test cases without losing the level of detail which is required to specify concrete test cases. So a formalization that is supposed to define concrete test cases can only treat a

subset of the whole test case space. Currently there is no standardized subset which specifies a comprehensive but sufficient test suite that ensures the safety of AVs to a high degree.

Concerning a subset of test cases which explicitly target safety critical advanced driver assistance systems (ADASs) e.g. adaptive cruise control (ACC), lane centering, emergency brake or collision avoidance the number of possible test cases is still too large for a formalization. A simple option is to reduce the subset to a number of certain ADASs. This reduces the generality of the formalization and raises the problem of reasoning about which ADASs should be supported. Another option is to subsume ADASs into groups. This raises the problem of determining shared characteristics of ADASs that separate ADASs. Again if the number of groups is too large a formalization can not handle all of them and if it is too low a formalization may not be able to express all the specific details which are needed to specify concrete test cases.

Testing an ADAS is complex. Each ADAS requires certain input metrics to operate e.g. current positions, distances or speeds of the AV to which the ADAS is attached to or of other participants. There are no standards to define which metrics ADASs require and how they have to be tested. Further input metrics may be properties about the AV like damage, steering angle or the state of certain electronic components e.g. the headlight. Depending on the implementation of an ADAS it may not require certain metrics as its input directly but other data like camera images or light detection and ranging (LiDAR) data which further increases the variety of input metrics. In order to test the results of ADASs even further metrics that e.g. represent a ground truth are required. This yields the problem that a formalization has to support many different kinds of metrics in order to provide ADASs with input metrics and to test them. The more metrics a formalization supports the more complex it may get.

AVs under test are controlled by artificial intelligences (AIs). Concerning a subset of test cases which evaluate the efficiency of a given AI the problem raises that the execution time of the frequent verification of test criteria, the overhead of the underlying simulator and the discrepancy between the hardware used for testing and the actual hardware used with a real AV falsify time measurements. Given all the metrics which an ADAS that is attached to an AV requires a simulation needs to exchange these possibly highly diverse metrics with the AI that controls the AVs. Since AIs differ greatly in their implementation they can not be included in the simulation directly and have to run separately. Additionally a tester may not want to expose the implementation of an AI to DRIVEBUILD. Hence AIs have to run externally i.e. not within the internal architecture of DRIVEBUILD. In case of an external AI the communication with a simulation and the network latency further falsify time measurements. In case of external AIs there is also the problem of creating a communication scheme which allows to request and exchange lots of diverse data and which exposes mechanisms to implement interactions between AIs and a simulation.

The more subsets a formalization has to consider and the bigger they are the higher is the diversity of metrics as well as test criteria which are required to define test cases. This results in the problem of an increasing complexity in the definition of test cases plus in the validation and evaluation of test criteria. There is no standardized set of test criteria which are sufficient for many test cases. There is also no standardized way of how to declare test criteria and how to specify reference tests and their expected results [7]. Testing ADASs that are not explicitly considered during the creation of the formalization may need test criteria which the formalization does not provide. To allow an user to introduce additional criteria on the client side for the purpose of introducing further criteria leads to the problem of distributing test criteria over the underlying platform and the user and thus divides the corresponding responsibilities of the verification of test criteria.

However, any subset of test cases involves AIs which control AVs. These AIs have to be trained before they are able to control an AV suitably. Therefore a tester wants to use training data which is collected in the same environment that is in place to run tests for an AI. This yields the problem of manually controlling participants in a simulation to efficiently generate training data.

In order to ensure a high degree of safety of AIs many test executions are required. In order to execute many tests simultaneously they may be distributed over a cluster. When distributing test runs across a cluster a common goal is high utilization of its provided resources. This leads to the problem of finding a strategy to distribute test executions based on their predicted load and their estimated execution time. Therefore characteristics of formalized test cases have to be determined that deposit in the resulting load and the actual execution time.

The goals of this work are the creation of a scheme which formalizes test cases, the support of training AIs, the specification of a life cycle for handling the execution of tests and the actual implementation of DRIVEBUILD. The formalization shall focus on ADASs and be able to describe static elements (e.g. roads and obstacles), dynamic elements (e.g. participants and their movements), test criteria and sensor data which AIs require.

3 Background

A common execution strategy for simulations is **synchronous simulation**. This strategy avoids simulations to be influenced by network latency and the current load of the underlying hardware. As a drawback this strategy does not support real-time simulations.

This strategy makes a simulation process wait frequently for AI processes which control AVs in the simulation to reach a certain state. Only if the AI processes reach this state both the simulation process and the AI processes continue.

Since synchronous simulation makes simulations pause frequently the real time does not correspond to their simulated time. Further is the speed of the simulated time dependent on the current load of the underlying hardware. To solve this the simulation time is separated from the real time by using **Ticks** [8]. Ticks define logical time units [9] on which both simulations and test cases in my work are based on. A single tick is a time interval of predefined length in which a simulator calculates the changes to the environment plus to all traffic participants and applies them to the simulation. A tick is the smallest considered logical time unit and can not be divided.

The **three-way handshake** [10] is a communication protocol that allows to establish a connection between a client and a server on an unreliable channel. Therefore the initiating client sends a **SYN** (“synchronize”) package to the server to request a connection. The server responds with a **SYN-ACK** (“**SYN** acknowledge”) package that informs the client that the server opened a connection. The client answers this package again with a **ACK** (“acknowledge”) package which confirms that the client knows about the opened connection. This establishes a connection.

The most well-known application is transmission control protocol (TCP) which extends the protocol with additional properties including error detection. TCP builds the basis of hypertext transfer protocol (HTTP) requests. Nevertheless, a three-way handshake is suitable to implement simple request mechanisms to exchange data on unreliable channels.

Micro services is an architectural design pattern which splits an application into small and autonomous services and connects them with light weight protocols [11]. In contrast to other

architectural design patterns like service oriented architecture (SOA) micro services introduce a very high level of resilience, the possibility to scale services independently instead of the whole application and easier composition of heterogeneous technologies. As a drawback micro services increase the complexity of the development and the deployment of an application. Further it requires more communication between its components which the application more dependent on network latencies.

Complex objects can often not be stored or transferred over a network as they are. **Serialization** is a technique to convert a complex object to a textual or binary representation and back [12]. These textual or binary representations can be stored or transferred over a network. Further serialization may implement additional properties which allow to check whether a serialized object is broken e. g. after transferring it over an unreliable connection.

The **master slave pattern** is a communication model which allows to distribute similar or identical computations over multiple nodes and parallelize them [13]. The pattern consists of a number of slave nodes that do the actual computations and exactly one master node that distributes tasks among the slave nodes, organizes them and collects their results. On the one hand this architecture is simple and introduces tolerance for faults of computations on slave nodes. On the other hand it also introduces the master node as single point of failure. Further this architecture does not allow direct communication between slave nodes without involving the master node which increases the required communication in case the slave nodes need to exchange data.

4 State of the Art

Section 2 shows that simulation based testing of AVs is a complex task and there are many problems which have to be investigated. The following subsections explain and discuss current approaches for solving some of these problems.

4.1 Formalization of Environments and Criteria

Concerning the definition of test environments OPENDRIVE [14] is one of most popular formats for defining very comprehensive and very detailed environments. Many well known car manufacturers (e. g. Audi, Bavarian Motor Works (BMW) and Daimler) and other organizations like Fraunhofer, Technical University of Munich (TUM) and deutsches Zentrum für Luft- und Raumfahrt (DLR) Institute of Robotics and Mechatronics use it. The format is XML based and offers declarations for many different kinds of objects like signs, cross falls, rail roads, bridges and signals which may even dynamically change. Especially the definition of streets and rail roads can be very complex. Additionally these object can be enhanced with meta information. So OPENDRIVE allows to define predecessor and successor lanes, neighbor lanes, complex junctions, parking spaces, acceleration strips, side walks, multiple different types of markings, reference lines for roads/junctions and rail road switches. Although OPENDRIVE has many options and capabilities to define environment elements OPENDRIVE on its own has neither the capability to add traffic participants nor to specify their movements nor to express any kind of test criterion. OPENCRCG [15] is a XML based format which extends OPENDRIVE. It allows to increase the

details of the roads defined in OPENDRIVE by adding bumps and unevenness to road surfaces using curved regular grids (CRGs). Therefore OPENCRG already includes tons of predefined data measured on existing roads.

OPENSENARIO [16] is a XML based scheme to add traffic participants to OPENDRIVE scenarios and bundles them with their physical properties and their dynamic behavior. The behavior is organized in maneuvers which are sequences of abstract actions like change lane, brake, accelerate and adapt the distance to other participants. OPENSENARIO is capable to define conditions which trigger these maneuvers as soon as they are satisfied. The variety of conditions includes time to collision (TTC), time headway, (relative) speed, traveled distance, speed, acceleration or reaching a certain position. Since OPENSENARIO uses XML for the description of the dynamic behavior it can not change during the simulation. Further maneuvers can not do any computations throughout a simulation e. g. calculate steering angles or any other information which the underlying simulator does not directly provide. OPENSENARIO is not able to define any kind of test criteria as well.

Another very popular format for declaring environments is COMMONROAD [17] which focuses solely on path planning problems. COMMONROAD scenarios are only capable to define lanes, obstacles and cars. A car can be associated with a list of states that describe its movement. Each state consists of a logical timestamp and the current position, orientation and speed of the car. The speed as well as the position may be not specified exactly but with an interval which allows to formulate uncertainty of these attributes. However, since COMMONROAD bundles timestamps with positions it does not guaranty that specified movements are realistic. Hence it is needed to check feasibility of movements separately beforehand. The definition of lanes in COMMONROAD is based on lanelets [18] which consist of two sequences of points that describe the left and the right border of a lane. In contrast the simulator I will use in my work (See Section 5.2) describes lanes with a sequence of road center points, the current width at each point and the number of left and right lanes. The simulator interpolates the sequence of road center points as well as the widths to generate a smooth curvature. As a result the simulator can visualize arbitrary lanelets of COMMONROAD scenarios with a guaranteed high accuracy. Concerning the definition of test criteria COMMONROAD is restricted to the definition of simple goal regions. If an AV reaches a goal region the test is considered successful and it failed otherwise.

There is also a model based approach to describe scenarios [19]. In contrast to other approaches this approach focuses on creating a description scheme that is not only comprehensive but also human-readable and abstracts from a scenario to a logical level. A main point is the abstraction in terms of the separation of spatial and temporal information. This separation allows the definition of acts which describe the current situation at some point during a test. Every act defines exactly one event which triggers a transition to another act in order to create a linear sequence of acts. All cars in a test have multiple perception layers of different sizes which surround the car. These perception layers define event triggers. The movements of participants are specified based on a predefined set of maneuvers. So the description of behaviors of AVs and the relation between acts may not be flexible enough to test a wide range of diverse ADASs.

GEO SCENARIO [20] is based on the OPENSTREETMAP (OSM) [21] standard DSL to formalize test scenarios. The goal of this formalization is to introduce reproducibility of test cases by enabling testers to describe complex traffic situations. Additionally the formalization focuses on an abstraction from the underlying tools that perform tests to allow to exchange them easily. So GEO SCENARIO allows to reuse well known existing tools like the Java OSM editor (JOSM) [22] with minor changes and can easily utilize existing services like Bing Maps [23] and ESRI Maps [24]. BEEP BEEP [25] is a formal and declarative DSL for complex event processing (CEP) queries on event traces of vehicles. In contrast to other approaches it allows to define custom boolean queries and does not restrict CEP on a predefined set of boolean queries. Further it allows to

describe non boolean queries and compose them using temporal logic based expressions in an exclusively declarative way. This strategy is very similar to the formalization which this work develops. Additionally BEEP BEEP can be used for real time evaluation.

4.2 Generation of Environments

SCENIC [26] is a probabilistic DSL which is geared towards the creation of training and test data sets for machine learning (ML) systems. Its scenarios define an environment model that characterizes scenarios which are e. g. realistic and describe certain types of interesting input. In contrast to other approaches SCENIC allows to specify distributions on the parameters that define environments instead of using concrete values. Furthermore these environment models allow a formal analysis of the properties and the correctness of a given ML system.

The scenario markup language (SML) [27] framework allows to automatically generate complex scenarios which have static as well as dynamic elements. It is designed to author studies on the behavior of drivers in realistic traffic simulations. Therefore a SML script declares events (e. g. accidents) that occur to a driver under predefined conditions and behavior fractions which group a set of commands (e. g. brake, steer and accelerate). Further it can compose these to supervisor elements which influence a simulated scenario. Since a SML script is a XML file the behavior elements are fixed during a simulation which comes with the same problems as OPENSCENARIO at least to a certain extent. Additionally events trigger modifications to a scenario during simulation time. So the SML framework requires a simulator that can change its content during the execution of a simulation.

An approach to implement a test generator is automatic crash constructor from crash report (AC3R) [28] which can reconstruct crashes by translating crash reports into simulations [29]. Therefore AC3R uses natural language processing (NLP) to extract information from a semi-structured police report like provided by the national highway traffic safety administration (NHTSA) database. The extracted information provides the geometry of the required roads as well as the initial positions of the involved participants. To determine the trajectories that led to the crash AC3R applies heuristics. Finally, AC3R generates a BEAMNG scenario that can simulate the crash. Additionally AC3R offers a way of measuring the accuracy of the simulation compared to the input report and is capable to derive test cases which encode similar conditions under which the initial crash happened. This allows testers to check whether their AI would have had an accident under the same or similar conditions too.

Another approach is ASFAULT [30] which uses procedural content generation [31]. ASFAULT focuses on testing lane keeping capabilities of AVs. Therefore it generates scenarios with exactly one road which an AV has to follow to pass the test. The curvature of the road is based on B-splines and a random number generator.

Another approach is evolutionary computation [32] which aims to generate single lane tracks with a high diversity. The paper measures the diversity of a track in terms of its curvature and speed profile. The more different types of curves concerning their length, their size and their angle and the more diverse the speeds an AV can reach on different sections of the generated track the more diverse it is considered. To achieve a high diversity the approach uses multi objective genetic algorithms (GAs) that evolve tracks with an as high as possible variety of turns and straight as well as driving speeds.

4.3 Simulators

OPENDS [33] is a java based cross platform simulator which specializes to simulate AVs. It utilizes the capabilities of the physics library JBULLET [34] and therefore provides a very realistic behavior of participants in the simulation. It bases the generation of the graphical representation for the simulation on the JMONKEYGAMEENGINE [35] framework and uses the lightweight java game library (LWJGL) [36] to utilize the graphics processing unit (GPU). Further OPENDS has the capability to generate scenarios from OPENDRIVE formalizations. However, AVs in OPENDS lack physical properties including a detailed damage report or the behavior of the bodywork and thus a detailed driving behavior.

SIMUL-A² [37] is a WEBOTS [38] based simulator which aims to evaluate ADASs. To increase the realism of the simulations of WEBOTS SIMUL-A² introduces multiple agents that control and influence AVs regardless of any action of the user. These agents use a communication scheme which can not only implement communication with WEBOTS but also exchange of data with real cars. Therefore SIMUL-A² uses tools and libraries of the robot operation system (ROS) project [39]. Although the agents make the simulations more real the physical behavior of objects does not allow highly accurate test executions.

CARLA [40] is a simulator that is geared towards to simulate urban environments especially traffic. It comes with many assets and models including buildings, vehicles, streets and weather conditions to create detailed urban scenarios. CARLA can only simulate vehicles and pedestrians that are abstracted by the concept of actors which represent the complete dynamic content. It uses a server multi-client architecture that allows to distribute the control of all actors across multiple nodes. Further it offers interfaces to manage all simulation related aspects e. g. traffic generation, behavior of pedestrians and vehicles, weather conditions and sensors attached to any actor. The range of available sensors includes cameras, LiDAR, depth and global positioning system (GPS) sensors. Additionally CARLA can create environments from OPENDRIVE descriptions and integrates ROS.

AIRSIM [41] is a project of Microsoft that serves as a plugin for Unreal Engine [42] environments. It can simulate vehicles but its focus is the simulation of drones. The goal of AIRSIM is to collect annotated training data for deep learning or reinforcement learning based AIs by controlling vehicles mostly manually. The available data includes only the state of vehicles, vision cameras and LiDAR sensors. The generation of maps for AIRSIM is a complex task since it requires to generate Unreal Engine environments.

The open racing car simulator (TORCS) [43] is geared towards developing, testing and comparing AIs. Many papers and research oriented competitions utilize TORCS because of its high degree of modularity of the architecture and its simple vehicle model which covers only basic properties of vehicle components and mechanical details, a simple aerodynamic model and friction. TORCS provides a set of multiple predefined racing tracks and does not support generating new tracks. The implementation of AIs for TORCS is complicated and requires AIs to control cars comprehensively e. g. to shift gears.

Siemens PLM Software developed a X-in-the-loop driving simulation platform [44] which focuses on testing and validating ADASs. The X under test may be software, hardware or the model of an AV or even a human driver. The platform is based on PRESCAN, MATLAB and SIMULINK which enables it to handle real-time models of AVs and their subsystems. Further the simulation platform provides tactile, motion, acoustic and visual feedback which enables a human to steer a car manually in real-time.

BEAMNG [45] is a simulator that uses in contrast to most other simulators a bottom up approach to model the physics of vehicles. Therefore instead of defining the physics of vehicles directly it specifies physics for all separate components of a vehicle e. g. tires, suspensions, differentials,

engines, doors and glass. A model of a vehicle is a collection of these components which are linked together with beams. So the physics of a vehicle is the result of the physics of all its components and their connections. Hence the physics of a vehicle has a very high level of detail and its behavior is very realistic. BEAMNG includes numerous models for vehicles, textures for road materials with different characteristics that influence the friction and weather conditions. There is a free but limited research version of BEAMNG which offers the full capacity of the physics engine but provides only one vehicle model and only one predefined environment. BEAMNGPY is a Python interface to dynamically create environments, to control BEAMNG simulations and to request data about vehicles and from their sensors. This work uses BEAMNG for all simulations.

4.4 Simulation Infrastructure and Platforms

To make a real car autonomous and to setup an environment to develop and test it is very time consuming, error prone and expensive.

AUTONOMOOSE [46] is a research platform at the university of Waterloo which modifies a single car to increase its level of autonomy step by step. This car is equipped with radar, sonar, LiDAR, inertial and vision sensors as well as a powerful embedded computer. This platform subsumes multiple research groups of multiple faculties. The current projects aim to improve the behavior in all-weather conditions, to optimize the fuel consumption, to reduce emissions and to provide methods to design safe and robust computer-based controls.

DEEPRACER [47] is a project on the Amazon web service (AWS) platform and focuses on reinforcement learning (RL). Therefore it provides a software environment which allows to design, train, test and simulate RL models. The project also offers a small AV which has a camera and is ready to accept and test trained RL models in the real world. Additionally researchers can compete with each other researchers in a racing league which AWS hosts.

Setting up simulations and interfaces for interacting with them is a complex task and not standardized. Existing simulators and architectural designs for simulation based testing are typically not freely available. Nevertheless, there are platforms like METAMOTO [48] that provide a simulation infrastructure to research groups in terms of a service as a service (SaaS) to circumvent these problems.

4.5 Comprehensive Approaches

PARACOSM [49] is a project which specifies a DSL to formalize test cases plus a simulation architecture. The DSL is a synchronous reactive programming language whose main concept are reactive objects which bundle geometric and graphical features with the behavior over time of physical objects in a simulation. PARACOSM uses 3D meshes to represent the geometric features of physical objects. Each reactive object defines input and output streams of data through which objects can communicate to each other and be composed to more complex objects in a flexible way. Hence the actual computations on or the analysis of data are equivalent to stream transformations over objects. PARACOSM provides only a small set of sensor data which includes camera and depth images. To extend PARACOSM with further types of sensor data it has to be extended using the underlying application program interface (API). PARACOSM can also generate random test cases automatically. However, PARACOSM does not provide any constructs for specifying test criteria. Additionally the internal representation is not compatible with any other well known

simulator than the one PARACOSM comes with. This shipped simulator lacks a precise reflection of physical behaviors in comparison to other simulators. The paper which presents PARACOSM is not clear about how AIs communicate with a simulator and there is no information about any performance measures. The paper is also not clear about whether simulator instances can run in parallel or whether the processes of the architecture can be distributed among multiple computers.

The open source project APOLLO [50] is a comprehensive platform which specifies a high performance and flexible architecture for the complete life cycle of developing, testing and deploying AVs. It ships with software components to localize traffic participants, to percept the environment and to plan routes plus it supports many types of sensors e. g. LiDAR sensors, cameras, radars and ultrasonic sensors. Additionally APOLLO comes with a cloud service and offers a web application which visualizes the current output of relevant modules, shows the status of hardware components, offers debugging tools, activates or disables modules during a simulation and allows to control AVs manually. However APOLLO does not provide test case criteria which consider complete scenarios or multiple traffic participants at once e. g. distance between participants.

AUTOWARE [51, 52, 53] is another comprehensive open source project. Similar to APOLLO it builds a complete ecosystem which implements algorithms for localization, perception, detection, prediction and planning, ships with predefined maps and has the capability to handle actual hardware including sensors and vehicles. It comes with the LG Silicon Valley Lab (LGSVL) simulator that can visualize information like perception data or status of other participants. AUTOWARE works with ROS bag (ROSBAG) files [54] which allow to record, replay and debug executed simulations. The integration of ROSBAG requires AUTOWARE to describe environments with pixel clouds. Hence in order to work with AUTOWARE a tester has to invest lots of time to create comprehensive pixel clouds. Furthermore a test has to rely on the shipped perception algorithm since this is the only source of information about the environment for every component.

4.6 Implementation of AIs

To determine requirements of AIs that control AVs in simulations this work considers multiple common approaches to implement AIs. The two major paradigms to implement AIs are mediated perception and behavior reflex. Mediated perception relies on camera images and tries to identify and classify objects as lanes, other participants, obstacles, etc. Based on the extracted information an AI computes control commands for an AV. Behavior reflex relies on camera images as well but instead of extracting information from an image it directly maps the whole image to a driving action using a regressor like a convolutional neural network (CNN).

DARPA autonomous vehicle (DAVE) [55] is a behavior reflex approach and relies on images of a single front camera. It uses a CNN to compute steering commands directly from given input images. This CNN uses camera images which are mapped to human steering angles as its training data. It learns to detect useful road features on its own and does not include any preprocessing steps for the camera images. The paper claims that this approach eventually leads to better performance and smaller systems because the internal components self-optimize.

Another approach for implementing an AI is DEEPDRIVING [56]. DEEPDRIVING is a direct perception approach which lies in between mediated perception and behavior reflex approaches. Unlike mediated perception it does not extract as much information as possible but uses only a small number of key perception indicators from an input image and estimates an affordance value of the situation at which the image was taken. The indicators include distances to lane markings and preceding participants. To extract the key perception indicators the approach uses

deep CNNs which are trained by manually driving an AV for a few hours. The paper claims that the set of key perception indicators is sufficient to completely describe scenes and to base driving decisions on it.

Another approach is deep RL [57]. This approach aims to train a neural network model that focuses solely on the lane keeping problem. Therefore it translates the problem of training a neural network into a problem of solving Markov decision problems (MDPs). The input of the neural network are single monocular images and the distance the AV traveled safely defines the reward for the neural network. The approach determines the traveled distance by feature extraction of the input image. The paper claims that it uses a continuous and model-free algorithm which can perform all exploration and optimizations on the AV while it drives. Further it claims that an AV learns to follow a lane with only a few steps of training.

SCALE AI [58] is a company which offers training data for AVs to companies including Samsung, Toyota and Lyft. It offers data as images and videos. The types of available data are LiDAR sensors, radar, point clouds, traffic lights plus annotated and labeled data. The data can be requested via a representational state transfer (REST) API. This API also offers ML based methods to label and annotate custom images and videos.

5 Methodology

This section describes all strategies and concepts DRIVEBUILD follows to formalize test cases and to implement a test life cycle. It also explains the cycle of the runtime verification process and the underlying strategies concerning the communication between the components of DRIVEBUILD and the communication between DRIVEBUILD and clients.

5.1 Test Case Formalization

The test case formalization specifies a test environment, the behavior of participants, the test criteria and the data which either AVs under test require or which has to be collected e. g. for training data. Since there are many different kinds of ADASs which simulation based testing can validate the formalization concentrates on a subset of ADASs. To determine a subset of ADASs which implement essential functionality of AVs and are safety critical I read many recent papers that discuss ADASs. Table 1 lists the targeted ADASs and groups them by their function. Hence ADASs in same the group presumably share the same minimum set of metrics which they require to operate. For each group Table 2 identifies which basic metrics they require. It reveals that some groups of ADASs require more metrics like Group 2 and others require only a few metrics like Group 3. All considered metrics result from the basic data types position of AV, position of lane markings and speed. To test an AV which integrates ADASs of these groups the AV requires the appropriate metrics either directly or in the form of other data types e. g. camera or LiDAR images which the underlying AI uses to extract the required metrics itself. Hence it is not sufficient for the formalization to provide only metrics which ADASs require directly but also to provide other common data types.

The formalization in this work describes test cases declaratively which avoids any need to compile or interpret the formalized test case. Further the formalization is based on XML which allows to define a XML schema definition (XSD) which can validate a test case to make sure that it is specified properly before it is processed. Additionally XML has great support in many languages

Table 1: Target ADASs — Lists all ADASs that the formalization aims to support. The ADASs are grouped based on their functionality and thus by the metrics they require to work.

Group	Supported ADASs	Description
1	Collision avoidance system Forward Collision Warning Emergency Brake Assist Intersection assistant Turning assistant	Systems that avoid or reduce the severity of collisions or at least warn a driver about them
2	Intelligent speed adaptation Cruise control Adaptive cruise control Active Brake Assist	Systems that control the speed of an AV or keep a safe distance to other participants
3	Lane centering Lane departure warning system Lane change assistance Wrong-way driving warning	Systems that observe the relative position of an AV on a lane or determine the direction of a lane.

Table 2: Minimum required metrics — Lists the minimal set of metrics which ADASs in the same group (see Table 1) presumably require.

Type of data	Group 1	Group 2	Group 3
Distance to other traffic participants	✓	✓	✗
Distance to lane markings and road edges	✗	✗	✓
Angle of AV to the road	✗	✓	✗
Speed	✓	✓	✗
Relative speed to other traffic participants	✓	✓	✓

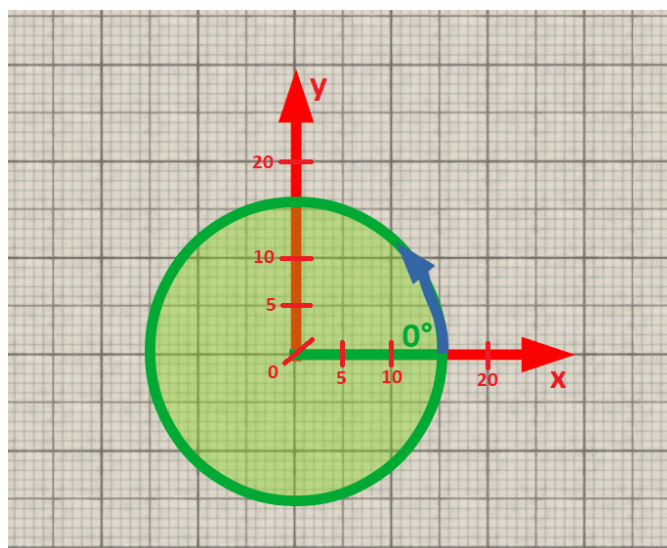


Figure 1: Environment coordinate system — Visualizes the theoretical view on an environment. For purpose of illustration the underlying grid shows squares which group 5 by 5 cells where each cell has an edge length of 1 m. This grid is not visible during a simulation.

and is well-known for decades. The formalization follows a modular approach in that separates the DRIVEBUILD environments (DBEs) which describe the static environments from DRIVEBUILD criteria (DBC) which describe participants and the test criteria. This separation allows to reuse environments throughout multiple scenarios and avoids duplication of possible very complex environments.

5.1.1 Formalization of Environments

An environment is a two dimensional world. The coordinate system defines positions with a x- and y-axis which have the unit meters and specifies rotation in anti clockwise manner starting with 0° pointing in positive x direction as shown in Figure 1. This representation sticks to commonly used mathematical representations and thus avoids additional translations between models and the formalization. A DBE specifies environment elements i. e. roads and obstacles. A road has a course, a number of left and right lanes and optionally has road markings and a name. The course is a sequence of tuples where each tuple contains a road center point and the current width of the road at that point. This representation allows to easily calculate the distance of an AV to the road center and to determine the direction of lanes. The number of left and right lanes influences the types of road markings in case a road has markings. Right lanes go in the same direction as the sequence of the road center points. Left lanes go in the opposite direction. In order to define basic static surroundings like buildings the formalization allows obstacles of type cube, cylinder, cone and bump. Obstacles can not be moved by participants, do not deform and are unbreakable. Figure 2 depicts simple examples of generated environment elements.

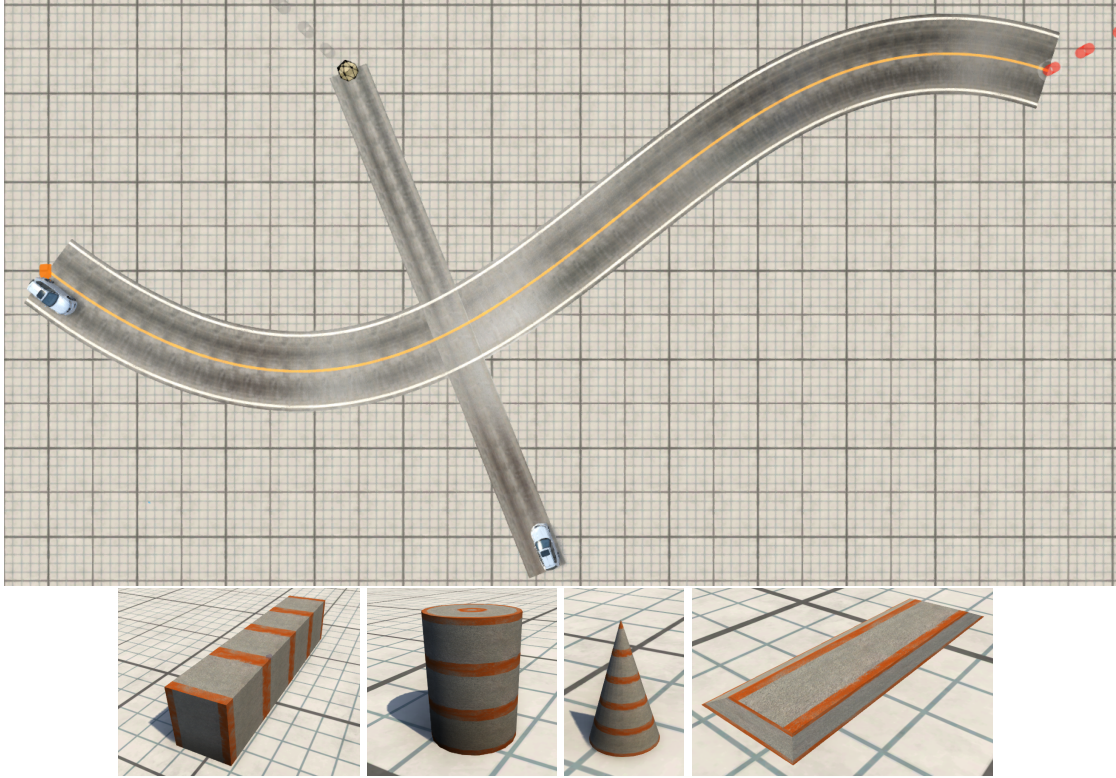


Figure 2: Visualization of example environment elements — Shows generated static environment elements including two roads and multiple obstacles. For purpose of illustration the underlying grid shows squares which group 5 by 5 cells where each cell has an edge length of 1 m. This grid is not visible during a simulation. Listings 4 and 6 in the appendix show the declarations to generate these environment elements.

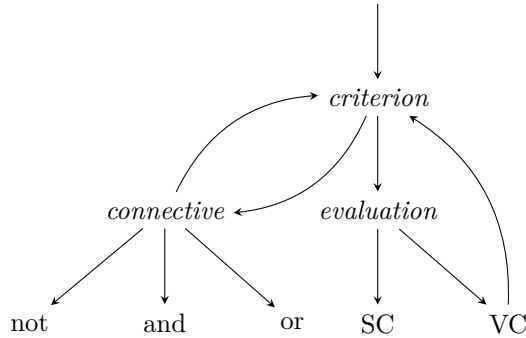


Figure 3: Nesting of a criterion — Shows the allowed nesting structure for SCs, VCs and connectives to define a criterion. Italic types are abstract.

5.1.2 Formalization of Criteria

The test criteria divide in precondition, success and fail criteria. Typically a test is considered as successful if it ends without triggering the fail criterion. In the context of testing AVs this may not be always true since the tests are very complex and there are cases where an AV does not succeed but a tester does not want the test to be marked as failed. A very common example is a test like “The AV is successful if it reaches a certain position and fails if it takes any damage or goes off-road”. If the AV under test does not move at all it does not pass the test. That the AV does currently not move does not imply that is not going to move at some point in the future. Hence it can not be concluded that the test failed. To describe such results DRIVEBUILD uses a three-valued logics. The most basic one is the Kleene and Priest logics [59] which declares besides **true** and **false** also the value **unknown**. Since the third value **unknown** is neutral to all connectives it can express that a criterion could not be determined or is currently not considered without influencing the outcome of the overall criterion. The definition of test criteria follows the concept of temporal logic and thus defines state conditions (SCs), validation constraints (VCs) and connectives (**and**, **or** and **not**) which can be nested as Figure 3 visualizes. SCs as well as VCs evaluate the current state of the simulation or an AV and determine whether it fulfills a certain condition. SCs yield in case the criterion can be evaluated either **true** or **false**. Otherwise it returns **unknown**. VCs restrict whether the inner criterion has to be considered in the evaluation of the parent criterion. If the condition of the VC is **true** the inner criterion is evaluated and the VC returns its result. Otherwise the VC returns **unknown**. The introduction of VCs allows to evaluate different criteria under different circumstances. This allows to enable or disable criteria under certain circumstances. E. g. define a fail criterion like “While AV A drives on road R it must not exceed a speed limit of S ”. In this case the speed of A should only be evaluated as long as A drives on R and return ideally either **true** or **false**. If A is not on R **unknown** shall be returned to ignore the criterion. Another use case for VCs may be to define that an AV is allowed to leave the road as long as it is in a certain area e. g. to avoid an obstacle or another participant that blocks the road in this area. Table 3 lists all supported kinds of criteria and whether they can be used as VC or SC. Listing 8 in the appendix shows example definitions for all of them. Using the mechanism which Section 5.3 explains a tester can introduce additional client side criteria.

Table 3: Test criteria — Lists all supported test criteria, describes their purpose and characterizes whether these can be used for VCs or SCs.

Type	Description	VC	SC
position	Checks whether an AV is at a certain position or within a certain radius of it	✓	✓
area	Checks whether an AV is within a certain area	✓	✓
lane	Checks whether an AV drives on a certain lane or off-road	✓	✓
speed	Checks whether the speed of an AV is below a given velocity	✓	✓
damage	Checks whether an AVs is damaged	✓	✓
time	Checks whether the simulation is currently within a certain interval of ticks	✓	✗
distance	Checks whether the distance between two AVs or between an AV and the center of the lane driving on is smaller than a given distance	✓	✓
TTC	Checks whether the time to collision (TTC) of an AV and another participant or obstacle is smaller than a given value	✓	✗

5.1.3 Formalization of Participants

The declaration of a participant specifies always a car model and an initial state. The chosen model fixes the shape and the physics of the participant. The set of available models is a predefined set which comes with BEAMNG. The initial state specifies the initial position and the initial orientation of a participant. If a participant is an AV the underlying AI and its ADASs require data to operate. The formalization allows to declaratively specify this data which a simulation has to collect and provide.

Resulting from the metrics which ADASs require in Table 2 and the available test criteria in Table 3 I determined multiple types of request data (see Table 4) which a formalization can declare and which a simulation can provide. It also shows which of the request data can be considered as sensor data and thus marks on which request data an AI should rely in order to be realistic. The other types of request data can be used to collect training data, to compare the computed results of an AI to a ground truth or to implement additional client side criteria before or after the computations which an AI does.

In case the participant is not autonomous the declaration may specify a movement. A movement is a sequence of waypoints which a participant has to follow. A waypoint is a position at which the behavior of a participant may change. It also has a tolerance value which avoids that a participant has to precisely reach a position and allows to pass by in a certain distance. A waypoint optionally defines a speed limit or a target speed which a participant has to obey until it reaches the next waypoint that specifies another value for them. Both the initial state and all waypoints have an attribute which specifying the current movement mode of a participant. The movement mode is one of **MANUAL**, **AUTONOMOUS** and **TRAINING**. If the current movement mode of a participant is **MANUAL** the car heads straight to the next waypoint and the target speed as well as the speed limit apply. If the movement mode is **AUTONOMOUS** the simulation requests the AI that registered for controlling the AV frequently and provides it with data. If the movement mode is set to **TRAINING** the AV acts the same way it does in **MANUAL** but the simulation requests the connected AI like in **AUTONOMOUS**. In contrast to **AUTONOMOUS** the AI can not control the AV. However, the AI can still control the simulation. This mode is geared towards collecting training data for AIs. The strategy to allow to change movement modes at each waypoint enables to mix

Table 4: Available request data — Lists all types of request data which an AI that registered at a simulation can possibly request and whether it can be considered as sensor data.

Type	Description	Sensor Data
position	Absolute position of an AV	×
speed	Absolute speed of an AV	✓
steering angle	Current steering angle of the steering wheel	✓
LiDAR	Distance data provided by a LiDAR sensor	✓
camera	Camera images either colored, annotated or with depth information	✓
damage	Detects whether an AV is damaged	✓
distance to road center	Distance to the center of the nearest road	×
heading angle	Angle between the orientation of a participant and the center of the nearest road	×
bounding box	The bounding box of a participant	×
road edges	Sequences of points for the right and left edge of a road	×

sections where a participant is forced to follow a path, where an AI has to control it or where to collect data. Listings 5 and 7 in the appendix show example definitions of participants as well as example declarations of many request data. Figure 4 shows the corresponding graphical representation of the participant and its movement.

5.2 Test Life Cycle

Figure 5 shows all phases of the test life cycle and groups them into the four main steps [input validation](#), [extraction](#), [transformation](#) and [execution](#).

The input validation checks whether the test case is broken or malformed and validates it against the XSD which enforces the structure of the formalized test case. If the test case is valid the process extracts the environment description, the test criteria and the participants. It passes the information about roads, obstacles and participants in the environment to the generator which creates representations that are compatible with the underlying simulator. The representation of the movements of the participants are passed directly to BEAMNG which applies them sequentially to the appropriate participants as soon as the simulation started. The transformation step creates Kleene and Priest logics expressions which represent the defined test criteria. The verification process uses these expressions to verify the criteria during the simulation. Then BEAMNG starts the execution of the test and sets up the runtime verification process (see Section 5.3). As soon as the runtime verification is able to determine whether the test succeeded or failed it stops the simulation and returns its test result.

5.3 Runtime Verification

The runtime verification cycle applies synchronous simulation (see Section 3) and Figure 6 depicts its five phases. The first phase checks whether the current state of the simulation yields a test

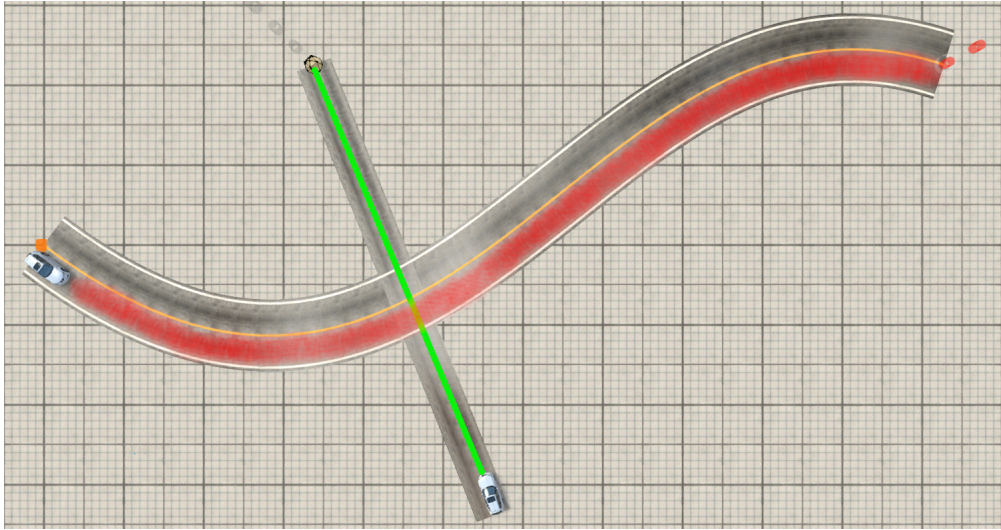


Figure 4: Visualization of example participants — Shows participants and their movements generated from the declarations in Listing 5 in the appendix. The green line marks the parts of a path where a participant is in **MANUAL** mode and has to follow the waypoints. The red stripe marks the area an AV in **AUTONOMOUS** mode is expected to follow but not forced to. In this case the participants do not change their movement mode during the simulation.

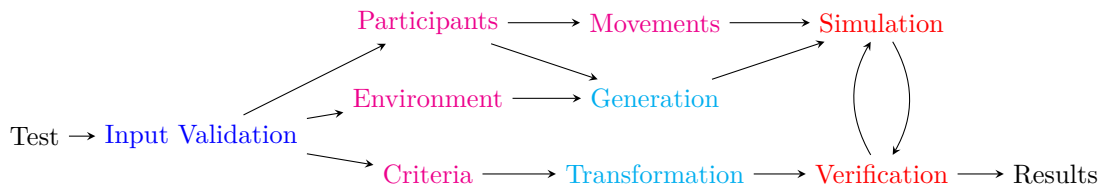


Figure 5: Test life cycle — Visualizes the four main steps of processing a formalized test. The input validation step is **blue**, the extraction step is **magenta**, the transformation step is **cyan** and the execution step is **red**.

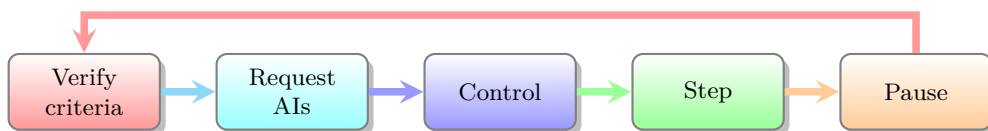


Figure 6: Runtime verification cycle — Depicts the main phases which implement the runtime verification and applies the synchronous simulation.

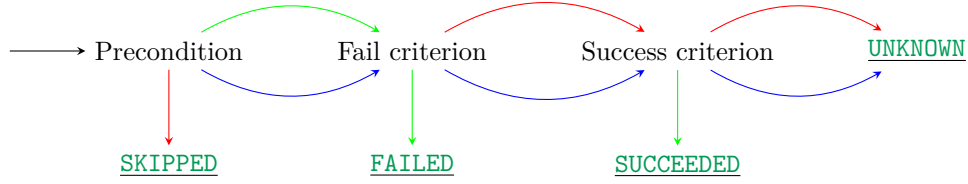


Figure 7: Test result decision tree — Visualizes the structure of the decision tree which determines the current test result. Underlined nodes are leaves and represent test results. The inner nodes represent the criteria of the test. Arrows describe the transitions from node to node which depends on whether a criterion evaluated to `true`, `false` or `unknown`.

result. Therefore it uses a decision tree where the leaves represent test results and the inner nodes represent the evaluation of the precondition, the fail and the success criterion. The inner nodes work with lazy evaluation and Figure 7 shows the complete tree. If the decision tree yields either `SKIPPED`, `FAILED` or `SUCCEEDED` the simulation stops and returns the current test result as final test result. In this case all AIs which are registered at the simulation are notified about the end of the simulation as well. If the current test result is `UNKNOWN` the runtime verification cycle continues with the next phase. The second phase implements the exchange of data with AIs. Therefore it updates all data of any sensor and all properties of any participant which were declared in the DBC or are required to evaluate the test criteria. The phase also searches for all AVs that are in movement mode `AUTONOMOUS` or `TRAINING`, notifies the AIs which registered for these AVs about the availability of new data and waits for them to send commands which control the AVs or the simulation. Figure 8 depicts the three-way protocol which the communication uses. The first message registers an AI at a certain simulation for a certain AV and blocks until the simulation notifies it about new data. Its response contains the current state of the simulation which is either `RUNNING`, `FINISHED`, `CANCELED` or `TIMEOUT`. The state of the simulation may be `UNKNOWN` which is only the case if the simulation could not be found. These states allow an AI to determine whether a simulation exists and whether it still runs or already stopped. They also allow to determine the reason why the simulation stopped. If the state of the simulation is `RUNNING` the AI requests sensor data or properties of its associated AV for which it needs to calculate control commands. A control command for an AV is a tuple which contains values for acceleration, brake intensity and steering angle. A command to control a simulation contains only the test result which has to be enforced. The opportunity to control a simulation allows an AI to eventually stop the simulation after the evaluation of client side criteria.

The control phase applies the commands which the AIs sent. In case one of them enforces a test result the simulation stops. Otherwise the phase applies all control commands of the AIs to their associated AV the runtime verification continues with the next phase. In this phase BEAMNG calculates the changes to the simulation for the next number of ticks as defined by the AI frequency, applies them and pauses the simulation again. Then the runtime verification cycle starts from the beginning.

5.4 Cluster Architecture

The architecture of DRIVEBUILD uses a master slave model. The slave components are simulation nodes (SIMNODES) that manage running simulations and runtime verification processes. The

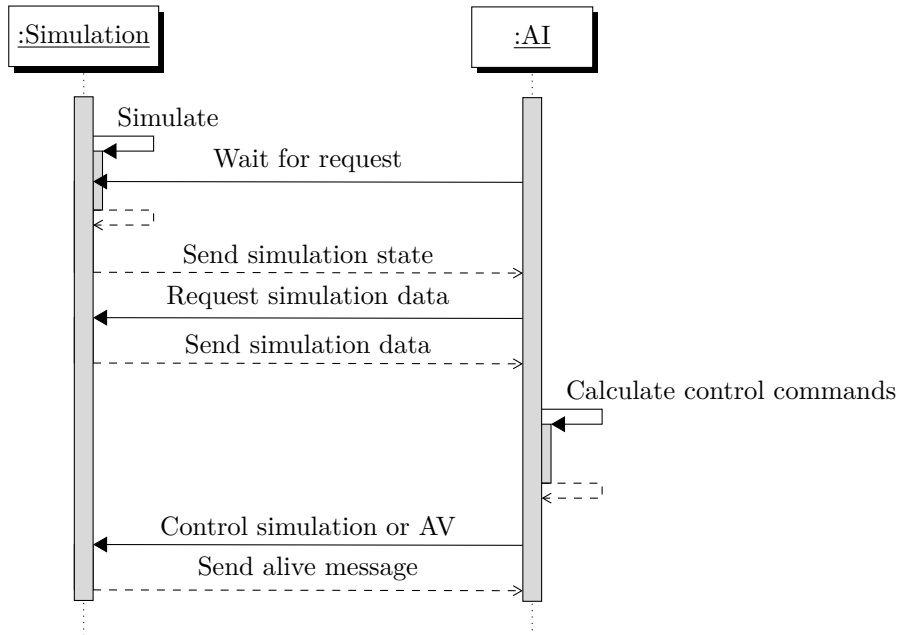


Figure 8: Communication between a simulation and an AI — Visualizes the messages of the three-way protocol sent for exchanging data between a simulation and an AI.

master component is the main application (MAINAPP). It organizes all SIMNODES and offers the entry point for all clients and AIs.

Figure 9 depicts all logical components of DRIVEBUILD, their data flow and how they are grouped to modules that can be distributed over a cluster. The architecture considers three different types of clients i. e. testers who execute tests, AIs that control AVs in simulations and researchers who access collected data. A tester submits test cases to the test case manager (TCMANAGER) which selects a SIMNODE based on the load distribution strategy. For load distribution the TCMANAGER selects the SIMNODE where currently the least number of simulations run and passes each test to a new simulation controller (SIMCONTROLLER) instance at that SIMNODE. The SIMCONTROLLER creates and manages the BEAMNG instance which executes the test and starts a runtime verification process that Section 5.3 describes in more detail. It also provides methods to request and monitor the current state of a test execution to the TCMANAGER. The SIMCONTROLLER passes the test to the TRANSFORMER which checks its validity, extracts static and dynamic information about the environment and the participants and generates a semantic representation. The TRANSFORMER also creates temporal logic expressions from the defined criteria such that these can be easily evaluated during the simulation by providing only the current state of the simulation as input. The SIMCONTROLLER uses the semantic representation to generate a BEAMNG scenario and executes it. When a simulation ends the SIMCONTROLLER stores the initial DBE and DBC along with its test result and other statistics which the evaluation needs in the database (see Section 7). The COMMUNICATOR handles the exchange of messages between a SIMCONTROLLER and AIs during a simulation and allows to collect training data. Therefore it uses the protocol which Figure 8 visualizes. The statistics manager (STATSMANAGER) grants access to the data which SIMCONTROLLERS store in the database and thus enables researchers to investigate and analyze collected data about test cases,

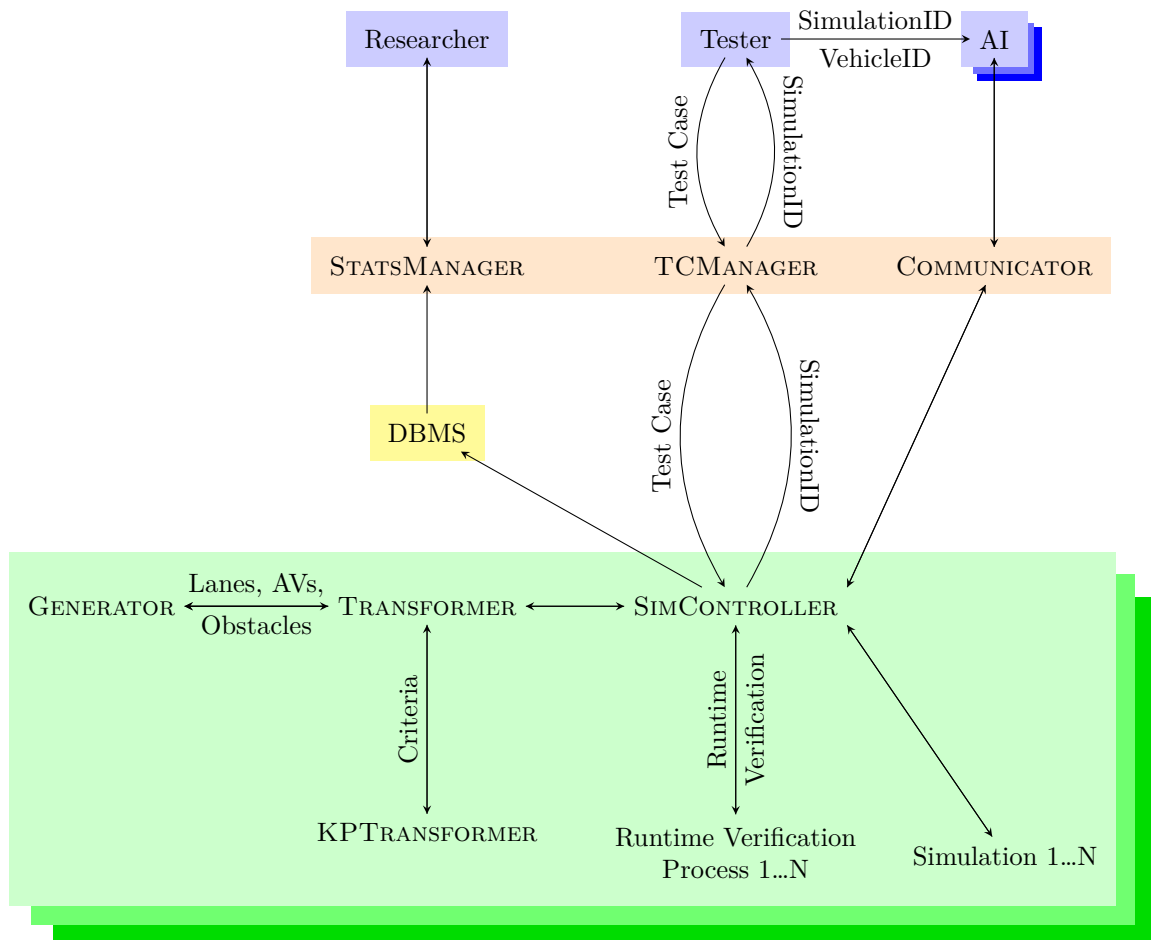


Figure 9: Cluster architecture — Visualizes all logical components of DRIVEBUILD and the data flow between them. It also groups the components into the modules `client`, `MAINAPP`, `DBMS` and `SIMNODE`.

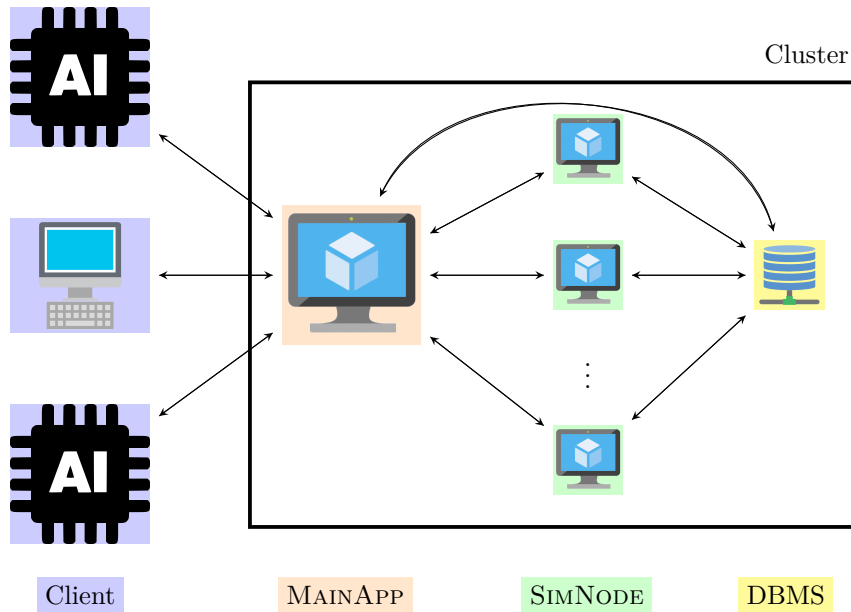


Figure 10: Distribution of modules over a cluster — Visualizes how the modules of DRIVEBUILD (see Figure 9) distribute over multiple nodes in a cluster and how they communicate with each other and the client.

their executions and their test results.

Figure 10 visualizes how the modules of DRIVEBUILD distribute over a cluster and how they communicate with each other and the client. A client uploads formalized test cases to the MAINAPP which distributes the test cases amongst a number of registered SIMNODES. Further a client starts instances of AIs which also connect to the MAINAPP in order to interact with the AVs which the uploaded test cases declare. Conceptually, the MAINAPP as well as the SIMNODES exchange data with the database and may run in virtual machines (VMs).

6 Implementation

6.1 Used Tools, Frameworks and Libraries

Table 11 lists all tools, frameworks and libraries DRIVEBUILD uses, characterizes them by whether the MAINAPP, a SIMNODE or a client implementation requires them and shows whether they are just a dependency of another element. This section describes the most essential tools, frameworks and libraries in detail and why they were chosen.

BEAMNG [45] is a racing game that is free to use for research purposes. It comes with a highly accurate physics engine and thus it is interesting for research as well. Most simulators have a top down approach where the physics of a vehicle are specified as a whole and the properties and behavior of all components of the car derive from it. In contrast BEAMNG uses a bottom up

approach where each tire, car suspension, the chassis, the engine, every cross beam, etc. has its own physics. A vehicle is a collection of these components which are connected by beams and thus the physics the vehicle results from the physics of its components.

BEAMNGPY [60] is a Python interface for BEAMNG. It allows to programmatically create scenarios, attach many kinds of sensors to vehicles, collect data and handle simulations and participants during simulations. BEAMNGPY can also access the pixel perfect annotation mode in which a camera returns images that mark on the image where buildings, roads, other participants etc. are. This can be used to gather training data for AIs. Since BEAMNGPY is not able to handle parallel requests to the same BEAMNG instance DRIVEBUILD wraps BEAMNGPY and introduces locks to implement basic thread safety.

DILL is a library which can serialize Python objects and extends the capabilities of the PICKLE module that ships with Python. SIMNODES have to serialize parameters when a process starts a new thread e.g. start a new BEAMNG instance and requires arguments.

FLASK [61] is a framework to create micro service based webservices. It is well known, reliable and uses annotations to map addresses with functions and parameters which makes it easy to use.

LXML [62] is a library for handling hypertext markup language (HTML) and XML files. It provides methods to parse and validate XML files against XSDs and to traverse them with XPath expressions. LXML is basically just a Python interface to the well known, reliable and very fast C libraries libxml2 [63] and libxslt [64].

Protocol buffers (PROTOBUF) [65] is a tool to specify and handle messages in a more type safe way than basic byte streams offer. Therefore it provides a language to define the structure of messages and the data types of their attributes. PROTOBUF is able to cross compile these definitions to appropriate representations in a set of well known programming languages including Python, Java, C++ and Haskell. The compilation process introduces additional methods to serialize and parse messages and to check the existence of attributes in messages.

WINPYTHON [66] is a Python distribution that ships with a set of Python packages for which the installation on Windows is complex or error prone. This includes especially packages like SCIPY which DRIVEBUILD requires.

6.2 Communication

Both the content of HTTP requests as well as of socket messages is serialized byte data which represents PROTOBUF messages (PMESSAGES). The communication uses the generated serialization methods of PROTOBUF to convert PMESSAGES from and to byte data. Figure 11 lists all simple and Figure 12 visualizes all complex PMESSAGES which the communication can handle as well as their structure. PMESSAGES are considered as complex if they have at least one attribute whose type is either another PMESSAGE, an enum or an exclusive group of attributes (`oneof`). An `oneof` group is not a PMESSAGE on its own but logically the superclass of multiple other PMESSAGES. PMESSAGES can only be properly instantiated from top level PMESSAGES. Each message on the socket level is a sequence of two sub-messages as shown in Figure 13. The content length sub-message has a fixed length and sends the number of bytes that the actual content has. The actual content sub-message is a PMESSAGE which contains the serialized content. This way the receiver knows the length of both sub-messages and thus can determine whether it already got the complete message or it has to wait for more bytes to receive. An action request is a sequence of socket messages which implements a function call over a socket. Figure 14 visualizes the sequence of socket messages. The first socket message sends an action that specifies which function has to be called. The second socket message defines the number of arguments N which

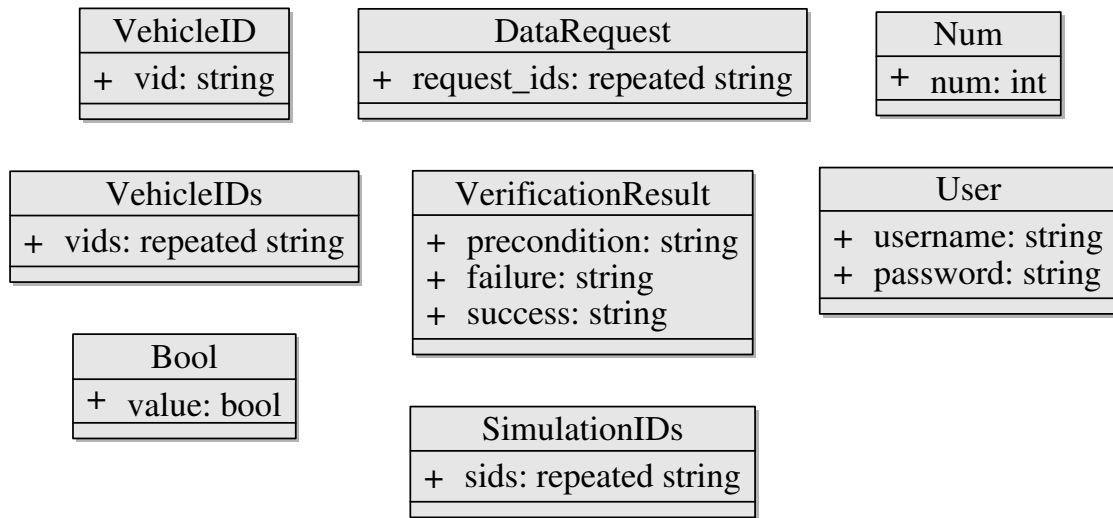


Figure 11: Simple PMESSAGES — Shows all basic PMESSAGES which do neither contain nor are nested in other PMESSAGES. It also lists their attributes with the appropriate type.

the sender passes to the function. The next N socket messages are PMESSAGES which contain the actual arguments.

6.3 MainApp

The MAINAPP is the central component of DRIVEBUILD and offers an interface to all types of clients (see Figure 9), manages SIMNODES and distributes tests across them. The interface for clients is a micro service [67] based webserver and Table 5 lists all its services. The use of micro services hides and strictly separates functionality plus it allows finer granularity than other architectures like SOA. From a testers perspective there is only the service `/runTests` which submits new tests to DRIVEBUILD and starts them. The tests have to be compressed into a zip file. This call blocks until DRIVEBUILD created all the resulting scenarios and started simulator instances which run all of them. The response of this request maps the names of the submitted tests to the generated `SimulationIDs`. A tester requires this information to map `VehicleIDs` that tests declare to `SimulationIDs` and start the corresponding AIs. The interface for AIs offers all micro services which they require to implement their interaction with the simulations (see Figure 8). A call to `/ai/waitForSimulatorRequest` registers an AI for controlling a certain AV in a certain simulation. This request blocks until either the simulation requests the registered AI or it finishes. The response contains the current state of the simulation to which this AI registered to which allows to determine whether the AI has to continue or to stop. If an AI continues it needs data for its computations. A request to `/ai/requestData` retrieves the data of sensors which are attached to a participant or current values of properties of a participant like the current position or damage. This data also allows to implement additional client side checks. `/ai/control` enables an AI to control AVs and simulations. The returned message contains a status message which may be used for logging. The interface of the MAINAPP offers also micro services for researchers

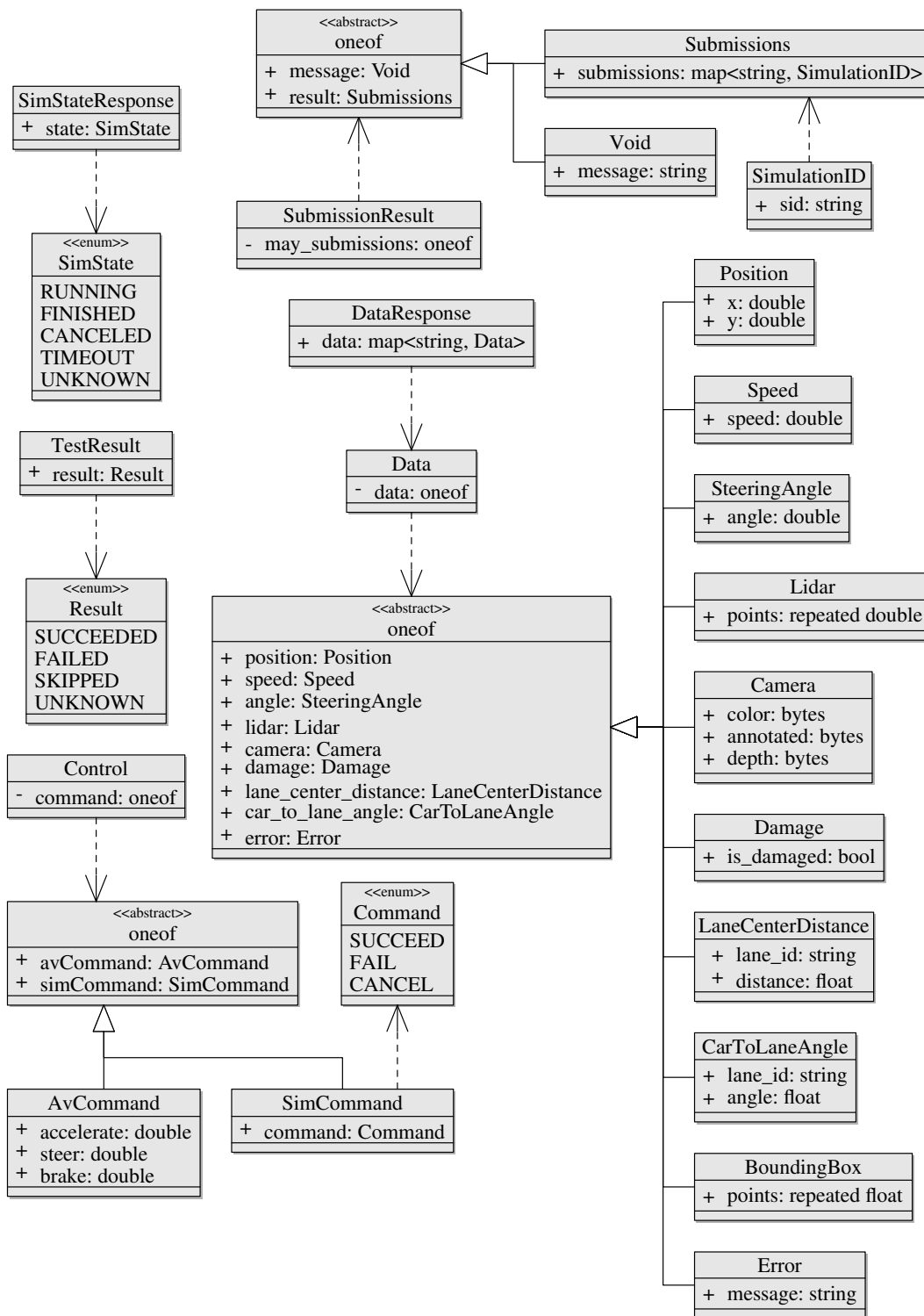


Figure 12: Complex PMESSAGES — Shows the composition structure of the complex PMESSAGES that DRIVEBUILD can handle. Private attributes denote the existence and the name of exclusive groups of attributes (**oneof**).

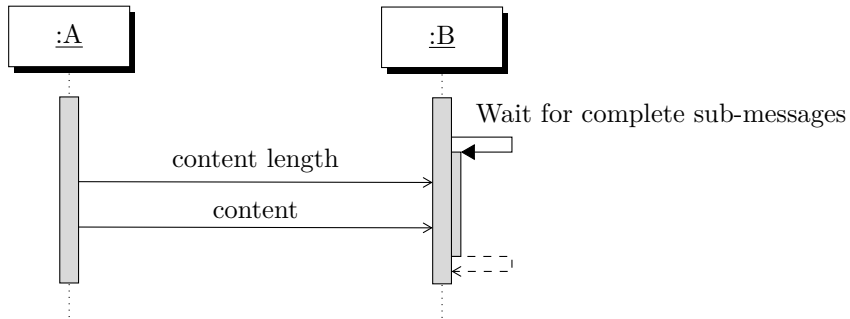


Figure 13: Socket message — Shows the sequence of sub-messages that form a complete message. This diagram assumes that A sends a message and B waits for receiving it.

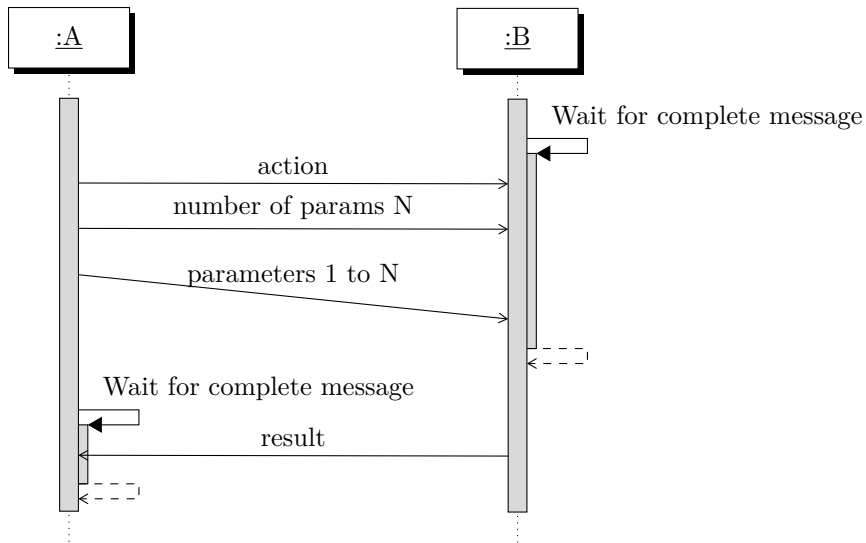


Figure 14: Action request — Shows the socket messages that a component A exchanges with component B if A requests an action that B provides. Each of these messages consists of sub-messages as described in Figure 13.

Table 5: MAINAPP micro services — Lists the micro services which the MAINAPP provides, a short description, the required parameters and the return type in case a request is successful. All parameters have to be appended to an URL as GET parameters except for underlined parameters which have to be sent as the content of a POST request. Italic entries add notes about parameters.

URL	Parameters (name: type)	Return Type
/runTests	user: User <u>zip byte data</u>	SubmissionResult
/ai/waitForSimulatorRequest	sid: SimulationID vid: VehicleID	SimStateResponse
/ai/requestData	sid: SimulationID vid: VehicleID request: DataRequest	DataResponse
/ai/control	sid: SimulationID vid: VehicleID <u>Control</u>	Void
/stats/getRunningSids	user: User	SubmissionResult
/stats/<action>	<i>At least:</i> sid: SimulationID <i>Maybe further parameters</i>	<i>Depends on action</i>
/sim/stop	sim: SimulationID result: TestResult	Void

to collect and analyze test data. So the micro service `/stats/<action>` provides an interface to query the database about currently running as well as previous tests. Calls to this service require a `SimulationID` as a parameter and possibly further parameters which are specific to the concrete action. The action `result` returns the test result of a simulation if any. A request with the action `status` returns the state of the simulation like `RUNNING`, `FINISHED` or `TIMEOUT`. The trace of all data that a simulation collected can be retrieved by calling `/stats/trace`. If a call specifies the optional parameter `VehicleID` the response contains only the collected data of the specified participant. The request `/sim/stop` forces simulations to end and can be called from a tester as well as from an AI. This call requires a `SimulationID` and a parameter which defines the test result to set. The result of this call contains a message which can be used for logging. For debugging purposes of DRIVEBUILD the MAINAPP further offers the request `/stats/getRunningSids` which returns all IDs of simulations which a given user currently runs on any SIMNODE.

The interface for SIMNODES is on the socket level and Section 6.4 describes it in detail. Besides the actual interface the MAINAPP opens a port which SIMNODES use to register themselves at the MAINAPP. When a new SIMNODE connects the MAINAPP generates a unique ID, associates the incoming socket connection with it and returns the ID to the newly registered SIMNODE.

6.4 SimNode

SIMNODES generate BEAMNG scenarios and run the actual test executions which include the simulations, the runtime verification, the interaction with AIs and the collection of data. The communication between SIMNODES and the MAINAPP is based on low level socket communication. Therefore the SIMCONTROLLERS of the SIMNODES offer the interface that Table 6 lists which enables the MAINAPP to manage simulations and to organize the interaction between on the one hand simulations and on the other hand testers and AIs. To use the interface the MAINAPP has to make action requests. The interface has many similarities to the micro services which the MAINAPP offers to clients since many of the micro services are only redirections of requests to appropriate SIMCONTROLLERS. This is the case for the actions `runTests`, `waitForSimulatorRequest`, `control`, `requestData` and `stop`. The action `runningTests` searches for all `SimulationIDs` of simulations which a given user currently runs on any SIMNODE and returns a map which associates test names with its `SimulationID`. A call to the action `requestSocket` makes the requested SIMCONTROLLER open an additional socket which registers itself to the MAINAPP. The MAINAPP uses this socket to open a socket for each AI in each currently running simulation. Since a SIMCONTROLLER runs multiple BEAMNG instances the corresponding runtime verification processes have to run simultaneously as well. Further a SIMCONTROLLER has to share references to all BEAMNG instances with the corresponding runtime verification processes. The exchange of data between processes in Python is based on serialization and results in deep copies of Python objects. Since BEAMNG and BEAMNGPY use internally lots of sockets and Python is intentionally not able to serialize sockets neither BEAMNG nor BEAMNGPY instances can be easily shared. So each SIMCONTROLLER offers a second interface (see Table 7) which runtime verification processes use to interact with BEAMNG instances and to evaluate criteria which require the current state of a simulation. A request with the action `vids` returns the IDs of all participants in a given simulation. The action request `isRunning` determines whether a specific simulation runs. The action `pollSensors` updates the cached data which AIs can request about a simulation, the properties of any participant and their sensors. The action request `verify` evaluates the test criteria based on the currently cached data and returns the evaluation result

Table 6: Interface to MAINAPP — Describes the interface to which the MAINAPP sends action requests to access the functions of the SIMCONTROLLER of a SIMNODE. Figure 14 describes the structure of action requests.

Action	Parameter Types	Return Type
runTests	User zip byte data	SubmissionResult
waitForSimulatorRequest	SimulationID VehicleID	SimStateResponse
control	SimulationID VehicleID Control	Void
requestData	SimulationID VehicleID DataRequest	DataResponse
stop	SimulationID TestResult	Void
runningTests	User	SubmissionResult
requestSocket	—	Void

Table 7: Interface to runtime verification processes — Describes the interface that a runtime verification uses to interact with simulations using action requests. Figure 14 describes the structure of action requests.

Action	Parameter Types	Return Type
vids	SimulationID	VehicleIDs
isRunning	SimulationID	Bool
pollSensors	SimulationID	Void
verify	SimulationID	VerificationResult
requestAiFor	SimulationID VehicleID	Void
steps	SimulationID Num	Void
stop	SimulationID TestResult	Void

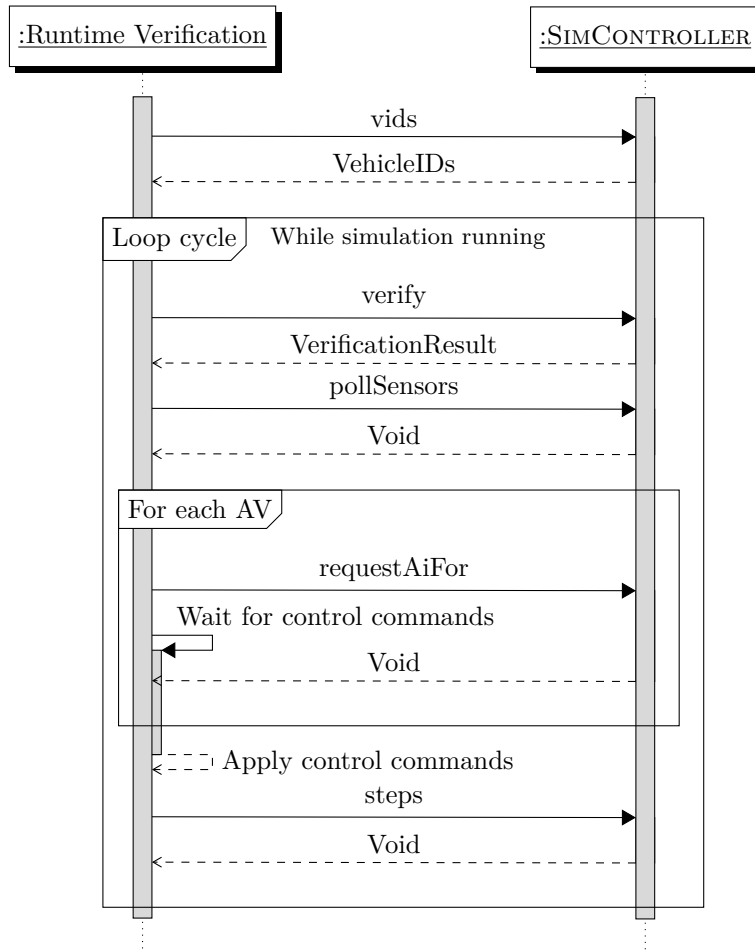


Figure 15: Runtime verification calls — Structures the calls of the runtime verification to the SIMCONTROLLER. It uses the interface which Table 7 lists.

of the precondition, the success and the failure criteria. A call to `requestAiFor` notifies all registered AIs about newly available data. This call blocks until for all AVs an AI registered with the action `waitForSimulatorRequest`. If so all the calls to `waitForSimulatorRequest` continue. The action request `steps` makes a simulation continue for the given amount of ticks. How much progress a simulation makes depends on the values `steps per second` which defines into how many ticks a second in the simulation time is divided and the `AI frequency` which specifies after how many ticks of simulation another cycle in the runtime verification starts.

Figure 15 depicts the sequence of calls that implements the cycle. The only call before the first cycle starts requests the IDs of all participants in the simulation. The first call within the cycle evaluates the test criteria and applies the decision tree in Figure 7 to determine the current test result. If the test result is something else than `UNKNOWN` it stops the simulation and notifies all registered AIs about it. Otherwise the cycle continues. In this case the runtime verification retrieves the current states and the sensor data of all AVs and updates the cached data which is available for AIs. The next calls notify all AIs which control AVs that are either in the movement

mode `AUTONOMOUS` or `TRAINING` with `requestAiFor` about new data and waits for the AIs to send control commands. Therefore the runtime verification starts a separate process that waits until the `SIMCONTROLLER` got all control commands of all AIs from the `MAINAPP`. The runtime verification first applies commands that control the simulation. If these stop the simulation the runtime verification notifies all registered AIs about it and exits. Otherwise it applies the control commands which control AVs and calls `step` which makes the simulation continue for a few ticks. Then the loop starts over again.

To implement the interaction between an AI and a simulation on the client side the implementation of the AI has to follow the scheme that Figure 16 shows. The interaction uses the same `PMESSAGES` as the low level socket communication uses (see Section 6.2). Listing 1 shows examples of how to create and read the most important `PMESSAGES` which the client requires. The first call submits formalized test cases to `DRIVEBUILD` and returns a map from the declared test names to the assigned `SimulationIDs`. The client starts for each participant in each simulation a separate process which interacts with the participant. This process repeats a certain sequence of calls as long as the simulation which simulates the participant runs. The first call in the sequence registers the process with `waitForSimulatorRequest` for the interaction with a specific participant in a certain simulation and blocks until one of the two following events occur. The one event occurs if the participant enters one of the movement modes `AUTONOMOUS` or `TRAINING` and the runtime verification calls `requestAiFor`. The other event occurs if the simulation stops. The call to `waitForSimulatorRequest` is the only required call. Any other call may be omitted. If the request returns and the simulation is not in state `RUNNING` the interaction stops, the process may clean up memory and exit. Otherwise the interaction continues and may request properties about participants or their sensor data. At this point a client can collect training data for AIs. If a participant is in movement mode `AUTONOMOUS` the process can either calculate commands which handle the AV or send commands which control the simulation. If the participant is in movement mode `TRAINING` it can only control the simulation and the `MAINAPP` ignores any command which controls an AV. Then the sequence of interaction calls starts over again.

Internally a `SIMNODE` implements all phases of the test life cycle (see Figure 5) and therefore the appropriate components of `DRIVEBUILD` as Figure 9 shows. The `TRANSFORMER` implements the input validation step. Therefore it validates tests against the `XSD` which `DRIVEBUILD` provides. The `GENERATOR` and the Kleene-Priest-Transformer (`KPTRANSFORMER`) implement the extraction step as well as the transformation step. In the extraction step they use `XPath` to extract all information that the semantic model requires. The actual semantic model results from the transformation step. The `GENERATOR` uses structs to represent the environment of a simulation. The `KPTRANSFORMER` parses precondition, success and failure criteria from top to bottom and recursively creates λ expressions that abstractly represent the test criteria as Kleene and Priest logics expressions. These λ expressions take a `BEAMNG` scenario as their input and can evaluate the criteria based on the current state of the scenario. Further the `KPTRANSFORMER` attaches default data requests which collect data to partially reproduce a simulation without having to actually execute it again or which future work might require. It also may attach additional sensors to participants if the evaluation of criteria requires further data. On the one hand the `SIMCONTROLLER` creates and manages the simulations and the runtime verification processes and on the other organizes the interaction with the `MAINAPP`.

Before starting a simulation the `SIMCONTROLLER` has to generate a `BEAMNG` scenario. The level for the scenario is an infinitely big flat plain which is covered with grass. A `BEAMNG` scenario consists of a `JSON`, a `prefab` and a `LUA` file. The `JSON` file contains meta data about the scenario including its name, the author, a description and a reference to the `prefab` file. The `prefab` file describes the static information about initial states of vehicles, roads, waypoints and `LUA` triggers. The `SIMCONTROLLER` uses `BEAMNGPY` to create an initial `BEAMNG` scenario

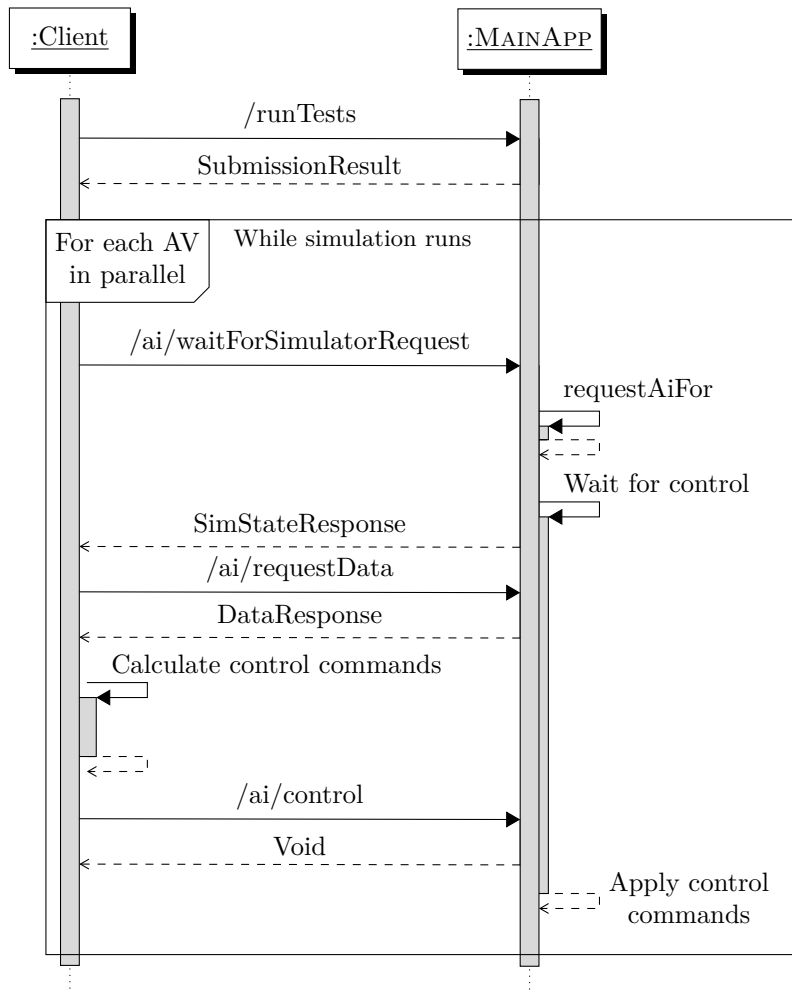


Figure 16: Client scheme — Depicts the basic scheme of the sequence of calls that a client has to make to implement interaction with DRIVEBUILD. Therefore it has to use the interface which Table 5 lists.

```

# Given a declared participant with ID "<someParticipantID>"
vid = VehicleID()
vid.vid = "<someParticipantID>"

# Given declared request data with IDs "<requestA>" and "<requestB>"
request = DataRequest()
request.request_ids.extend(["<requestA>", "<requestB>"])

```

(a) Creation of simple messages

```

# Given an object submission_result of type SubmissionResult
# Given a declared test with name "<testA>"
print(submission_result.message.message)
print(submission_result.result.submissions["<testA>"].sid)

# Given an object data of type DataResponse
# Given declared request data with IDs "<requestA>" (RoadCenterDistance) and
↳ "<requestB>" (Speed) and an undeclared ID "<requestC>" (Error)
print(data.data["<requestA>"].road_center_distance.road_id)
print(data.data["<requestA>"].road_center_distance.distance)
print(data.data["<requestB>"].speed.speed)
print(data.data["<requestC>"].error.message)

```

(b) Access to attributes of complex messages

Listing 1: Example PMESSAGES — Demonstrates the creation and usage of the fundamental PMESSAGES that a client implementation requires.

but further modifies it since BEAMNGPY does not provide all required features.

The definition of positions of participants in the formalization corresponds to BEAMNG but the definition of orientations of participants defers. The formalization defines orientation as Figure 1 shows. In contrast BEAMNG defines orientation in clockwise manner and an orientation of 0° makes the car head in negative y direction. So the generator has to translate the orientation accordingly before placing participants in a BEAMNG scenario. When adding a road it is smoothed by interpolating the points which the semantic model stores for its course with a cubic B-spline. The interpolation sets the smoothness to 0 and adds additional points to ensure that the resulting course fits all points perfectly and the its shape is like expected. The interpolation is done with SCIPY [68] The road markings in the scenario are implemented as narrow lanes which are parallel to the B-spline and just have a different texture than the actual road. SCIPY provides methods to calculate these parallel lines, i. e. offset lines. A road has a declared number of left and right lanes. The generator creates two offset lines for the left and right side road marking (continuous white line), one offset line which separates left from right lanes (continuous double yellow line) and offset lines that separate left and right lanes from each other (dashed white line). The dynamic behavior of participants is described by sequences of waypoints in the semantic model. The generator adds for each of these waypoints corresponding BEAMNG waypoints which have the position and size as defined in the semantic model. Further it adds LUA triggers for each BEAMNG waypoint. LUA triggers define areas where custom LUA functions are executed as soon as the bounding box of a participant intersects with it or

does not intersect with it anymore. When a participant reaches a LUA trigger the simulation calls a custom LUA function which makes the participant move to the next BEAMNG waypoint. Since these functions make a participant approach to a BEAMNG waypoint such a way that it almost touches the waypoint without intersecting it LUA triggers have to be slightly bigger than the underlying BEAMNG waypoint to make sure the participant intersects the LUA trigger. When a participant reaches a LUA trigger the custom LUA function also applies the additional properties of the waypoints in the semantic model which includes target speeds, speed limits and changes of the movement mode. Due to the internal implementation of BEAMNG a speed limit overrides a target speed. If the generation of the BEAMNG scenario finished the SIMCONTROLLER starts a new BEAMNG instance, loads the scenario and starts the simulation.

6.5 DBMS

The database uses POSTGRESQL since POSTGRESQL is well known, well supported in many languages and provides the data type `SERIAL` which allows to easily and safely generate unique `SimulationIDs` for tests. Figure 17 depicts the complete scheme which DRIVEBUILD uses to store data about finished and currently running tests as well as data about properties and sensor data of all participants in any runtime verification cycle. The table “Test” stores for each test execution the `SimulationID`, the current test result, the current state of the simulation and timestamps for when the execution started and finished. Additionally it contains serialized byte strings of the DBE and DBC which specify the test. Each entry in “Test” references the user in “User” who submitted the test to DRIVEBUILD. Each user has an username and a password. The table “VerificationCycles” stores all data which the runtime verification collects in each cycle. This includes especially all properties and sensor data of any participants. Since the properties as well as the sensor data is highly diverse and may change when more sensors are added or when the provided detail of sensor data increases it is very hard to specify a fixed static database management system (DBMS) scheme. So the collected data is stored in a single field where a serialized `DataResponse` object contains all the data. Each entry additionally stores two timestamps that save the time from the start of a cycle until the call to BEAMNG for continuing the simulation.

7 Evaluation

The evaluation aims to show that the test formalization of DRIVEBUILD is general enough to be used with different approaches for implementing test generators and AIs. It also aims to show whether test executions on a single SIMNODE on a real machine scale linearly and whether the performance degradation when using VMs to host SIMNODEs is unbearable. The evaluation addresses the following main research questions:

RQ1 How comprehensive is the test formalization concerning the support of various kinds of test generators and AIs? Different approaches for test generators and AIs have different requirements when testing them. The creation of benchmarks and ratings for comparing test generators and AIs to each other introduces even further requirements. Thus in order to interact with test generators and AIs, to analyze and to offer feedback data the formalization has to fulfill these requirements. The evaluation focuses on approaches that Section 4 presents to investigate the research question.

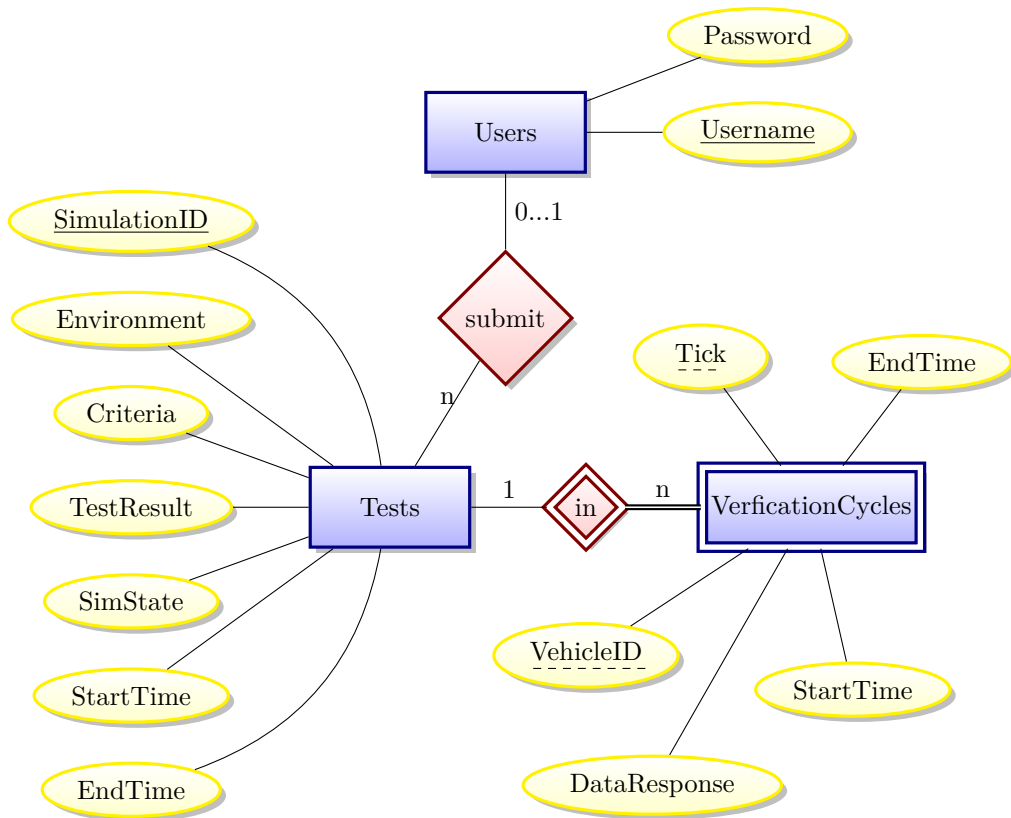


Figure 17: Database scheme — Depicts all the data that DRIVEBUILD stores about executed and running tests and how the data relates.

RQ2 How does DRIVEBUILD scale over real machines and VMs? DRIVEBUILD is a distributed system and aims to run as many simulations as possible in parallel. This research question investigates how many tests a single SIMNODE can run without having an influence on the test results, the benefit of utilizing parallelism and whether hosting SIMNODES in VMs is a feasible solution to distribute simulations over a cluster.

RQ3 How supportive is DRIVEBUILD for developers in setting up simulations, running tests and collecting data? A goal of DRIVEBUILD is to lift out testers of tedious and error prone tasks like setting up and managing simulations as well as implementing an interaction between simulations and AIs. This research question qualitatively evaluates the benefits of using DRIVEBUILD compared to manually setting up and executing tests. Therefore it compares the effort to understand and use DRIVEBUILD to the benefits DRIVEBUILD offers.

The evaluation has four subsections. Section 7.1 elaborates in which context the evaluation took place and lists all technical specifications which are relevant. Section 7.2 describes the test for checking the comprehensiveness of the test formalization and introduces a number of metrics which DRIVEBUILD can yield about test generators and AIs. Section 7.3 quantitatively evaluates the scalability of DRIVEBUILD concerning the number of simultaneously executed simulations and the number of SIMNODES connected to the MAINAPP. It also compares the scalability between a SIMNODE running on a real machine and SIMNODES hosted on VMs. Section 7.4 examines qualitatively the benefits of DRIVEBUILD, how supportive it is for testers and the presumably most difficult tasks when using it.

7.1 Experimental Settings

7.1.1 Seminar

The evaluation took place in the context of the advanced seminar “Search-based Software Engineering for Testing Autonomous Cars (5846HS)” in summer term 2019 at the University of Passau and had 10 participants which were all either master students or bachelor students in a higher semester. The students were assigned papers that propose and discuss test generators and approaches for training AIs. As part of the seminar, the students had the task to implement small prototypes of test generators or AIs which I will refer to as “submissions”. So they have to face problems which are similar to problems real tester have. The problems include the setup of BEAMNG, the management of simulations and the collection of data which are problems that DRIVEBUILD claims to solve. Hence I targeted the students as users like testers would use DRIVEBUILD.

7.1.2 Submissions

Section 3 explains all approaches the students used for their submissions. Three students implemented test generators. Test generator G1 focuses on generating tests that verify whether AVs can avoid crashes. Therefore it uses a predefined list of 528 IDs of crash reports referencing the NHTSA crash report database which provides these reports as semi-structured XML. G1 utilizes AC3R [28] which generates BEAMNG scenarios. It parses these BEAMNG scenarios and

formalizes them such a way that they can be used with DRIVEBUILD. Test generator G2 aims to generate test scenarios that focus on verifying the lane keeping capabilities of AVs. It claims to extend ASFAULT [30] by randomly placing static obstacles all over the area where the road is settled. Test generator G3 focuses on testing lane keeping capabilities of AVs as well by applying evolutionary computing to generate a population of interesting roads. Each call to G3 returns only the first element of the resulting population. It does not apply any metric to determine a specific element in the population. Besides the test generators students submitted I created a reference generator G0 which generates random tests using ASFAULT. The generated tests have one road which is restricted to be placed in an area of $100\text{ m} \times 100\text{ m}$. This road has two lanes and a fixed width of 8 m. The initial seed is a random number between 0 and 10 000.

Two of the students implemented AIs. The AI A1 is based on a version of deep reinforcement learning which uses deep deterministic policy gradient (DDPG) and a variational encoder (VAE). The student pretrained the AI on roads generated by ASFAULT. AI A2 uses DEEPDRIVING. To pretrain the AI the student drove a car manually with BEAMNG on the level which DRIVEBUILD uses for its simulations. Additionally to the AIs students implemented I use the BEAMNG AI as reference AI A0. This AI has perfect knowledge about all roads but does neither recognize any obstacles nor any other participants. It has an option “stay on lane” which allows to define a single target waypoint instead of a sequence of consecutive waypoints the AV has to follow. The AI is able to detect which lanes it has to follow to reach the desired target waypoint. As a restriction the target waypoint has to be right in the middle of the road otherwise the BEAMNG AI does not find a path to it.

7.1.3 Technical Specifications

DRIVEBUILD is a distributed system and the evaluation utilizes this capability by running the components of DRIVEBUILD, test generators, AIs and the actual evaluation process across multiple nodes. Table 8 lists the specifications of all available nodes. During the evaluation node 0 hosts one instance of the MAINAPP which is accessible from all the other nodes and node 1 runs all the submitted test generators, AIs and the SUBMISSIONTESTER which encapsulates the execution of the submissions, their interaction with DRIVEBUILD, the preprocessing of tests and the collection of data for the evaluation. Node 2 and one or multiple instances of node 3 may run instances of SIMNODES depending on the current test setup. Each active instance of these nodes hosts exactly one SIMNODE.

7.2 Evaluation of Generality

The evaluation of the generality of the test formalization considers two main aspects. The first aspect is the feature coverage that determines which features of DRIVEBUILD were used during the evaluation and which submissions used them. The second aspect examines on the one hand which metrics the submissions require to operate and whether DRIVEBUILD provides them and on the other hand which further metrics DRIVEBUILD offers to create detailed analysis about the submissions. All time measurements in any evaluation use real time over simulation time since measuring real time is at many points easier than measuring simulation time. Further real time considers in contrast to the simulation time the time test generators require to generate new tests as well as the time AIs need to calculate control commands.

Table 8: Node specifications — Lists the specifications for all (virtual) nodes that were used for the evaluation.

Property	Node 0	Node 1	Node 2	Node 3
Hardware/VM?	VM	Hardware	Hardware	VM
OS	Linux Debian	Windows 10 Enterprise	Windows 10 Pro	Windows 10 Pro
Version	10	1903	1903	1903
Build	4.19.0-6	18362.418	18362.418	18362.418
Architecture	64 bit	64 bit	64 bit	64 bit
CPU	Intel Xeon E5-2630	Intel Core i7-7700K	Intel Core i7-7700HQ	Intel Xeon E3-12xx v2
Clock Rate	2.4 GHz	4.2 GHz	2.8 GHz	3 GHz
Logical Cores	8	8	8	2
RAM	8 GB	16 GB	16 GB	16 GB
GPU	<i>Not used</i>	GeForce GTX 1080	GeForce GTX 1070	GeForce GTX Titan Black
Driver Version	<i>Not used</i>	436.48	436.48	436.30

7.2.1 Challenge Test

In order to investigate these aspects the challenge test runs the submitted AIs against the submitted test generators. In terms of the challenge test a combination of a test generator and an AI is called a “match” and has a fixed time budget. One execution of the challenge test lets each AI compete against each test generator. Each match repeats the following steps again and again until the given time budget is used. First, the selected test generator generates a test. In case the generator succeeded the SUBMISSIONTESTER checks multiple validity constraints which ensure that the test can be executed and that it can be compared with generated tests of other test generators. A test is only valid if it defines at least one participant with the name “ego”, defines at least one road and the generated criteria (DBC) references the generated environment (DBE). If the SUBMISSIONTESTER considers the test valid it further modifies the test to ensure comparability with other tests and compatibility with the AI assigned to the match: It enforces a failure criterion which makes the test fail if any participants is off-road or damaged. It also forces the AV “ego” to be in movement mode **AUTONOMOUS** during the whole simulation and any other participants to be in movement mode **TRAINING** to make sure it can collect data about all participants. To enable the assigned AI to operate properly the SUBMISSIONTESTER adds data requests to the test for all monitoring data the AI requires. For the analysis of the submissions it adds even more data requests to every participant which store the heading angle as well as the position of the participant. In case of A0 the SUBMISSIONTESTER has to define a target position for the AI. Appendix A.2 describes the process of finding an appropriate target position in detail. The SUBMISSIONTESTER submits the preprocessed test to DRIVEBUILD. If DRIVEBUILD accepts the test the SUBMISSIONTESTER creates one instance of the assigned AI and connects it to the AV with ID “ego”. I execute the challenge test in different configurations. An execution defines the time budget for matches either with 10 min or 30 min and enforces road markings or no road markings. Additionally I repeat each execution once which results in a total of 8 executions. At the end this yielded a total number of 1208 generated tests.

```

from drivebuildclient.aiExchangeMessage_pb2 import SimulationID, VehicleID,
↳ DataResponse
from pathlib import Path
from typing import Optional, Tuple

class TestGenerator:
    def getTest() -> Optional[Tuple[Path, Path]]:
        # NOTE The first path points to the DBE, the second to the DBC
        raise NotImplementedError("Not implemented, yet.")

```

Listing 2: Test generator stub — Describes the stub for the implementation of a test generator.

7.2.2 Interfaces for Challenge Test

To allow the SUBMISSIONTESTER to work with all submissions homogeneously they have to implement predefined interfaces. A test generator has to provide a class with a method having the signature given in Listing 2. The method `getTest()` does not accept any input and returns a tuple with two paths where the first points to a DBE file and the second to a DBC file which references the DBE file. The method may return `None` if the generator is not able to generate more tests.

An implementation of an AI has to implement the interface Listing 3 shows. The constructor of the AI accepts an instance representing the connection to DRIVEBUILD. This connection provides methods for the AI to request data and control AVs. The method `add_data_requests(...)` adds data requests (see Listing 7) which the AI requires for operation to the declarations of AVs. Therefore it accepts the XML tag to which the data requests have to be added to and the ID of the AV for which the data requests are declared. This ID should be used to create unique IDs for the declared data requests. The method `start(...)` follows the client scheme (see Figure 16) which implements the interaction between an AI and DRIVEBUILD. It accepts the ID of the simulation which simulates the AV to control, the ID of AV itself and a callback method which gathers runtime data about the simulation for the evaluation afterwards.

7.2.3 Feature Coverage

The feature coverage indicates which features of DRIVEBUILD were used during the evaluation and thus it suggests which features are somewhat tested and presumably work as they are expected. Table 9 summarises the most basic features DRIVEBUILD provides and which submissions use them. The submissions use only a very restricted subset of the features that DRIVEBUILD offers. There is only one test generator which declares obstacles. There is also only one test generator which defines more than a single participant on a single road and which intentionally creates intersections. That the test generators use the road markings feature results from the fact that the configuration of the challenge test enforces them. The AIs rely on camera images and one AI additionally relies on the current speed of an AV. The evaluation part of the table includes features to evaluate the challenge test as well as the test criteria. The evaluation uses only features which are related to check and analyse the lane keeping capabilities of AIs. The evaluation does not declare any criteria that use validation constraints (VCs).

```

from drivebuildclient.aiExchangeMessage_pb2 import SimulationID, VehicleID,
↳ SimStateResponse
from drivebuildclient.AIExchangeService import AIExchangeService
from lxml.etree import _Element

class AI:
    def __init__(self, service: AIExchangeService):
        self.service = service

    @staticmethod
    def add_data_requests(ai_tag: _Element, participant_id: str) -> None:
        raise NotImplementedError("Not implemented, yet.")

    def start(self, sid: SimulationID, vid: VehicleID, add_dynamic_stats:
↳ Callable[[], None]) -> None:
        while True:
            sim_state = self.service.wait_for_simulator_request(sid, vid)
            if sim_state == SimStateResponse.SimState.RUNNING:
                add_dynamic_stats() # Allows the evaluation to introduce
↳ further calls to DriveBuild
                # Request data
                # Control AV or simulation
            else:
                break

```

Listing 3: AI stub — Shows the scheme of the implementation of AIs required for the evaluation and uses the client scheme of Figure 16.

Table 9: Feature coverage — Summarizes which features are used by the submissions and the evaluation. If a cell contains check mark a submission or the evaluation uses the feature intentionally. If a cell contains a tilde the feature is used only sometimes and the submission is not explicitly designed to use it.

Feature	G0	G1	G2	G3
Scenario Elements				
Multiple participants		✓		
Obstacles			✓	
Multiple roads		✓		
Intersections	~	✓	~	~
Test Configuration				
Road markings	✓	✓	✓	✓
Speed limits				
Target speeds				
	A0	A1	A2	Evaluation
Data Requests				
Camera images		✓	✓	
Speed			✓	
Position				✓
Bounding box				✓
Criteria				
Damage detection				✓
Off-road detection				✓
Goal area detection				✓
VC				

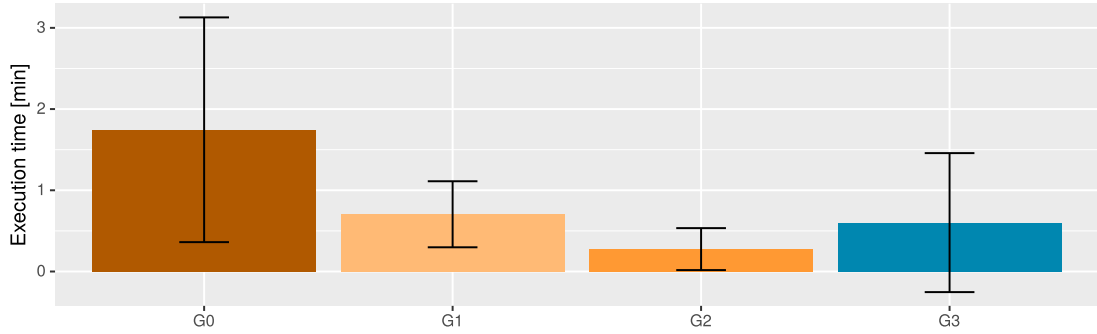


Figure 18: Execution times — Visualizes the durations of executed simulations and their standard deviation. The execution time includes generation of a test, upload to DRIVEBUILD, start of a simulator instance, running the test until it finishes. This diagram shows only execution times of tests which did not encounter a timeout.

As a conclusion the small set of used features is sufficient to support all submissions and to create metrics to analyze them. DRIVEBUILD offers many more features which are not listed in Table 9. Since neither the submissions nor the evaluation use them their investigation is subject of future.

7.2.4 Efficiency of Test Generators

The efficiency of test generators is a metric to identify how many valid tests a generator can produce in a given time interval. Figure 18 visualizes the distribution of the duration of the executed tests. The efficiency e of the test generators is 1 divided by the average time per test execution t_{avg} (see Equation (1)).

$$e = 1/t_{avg} \quad (1)$$

The average execution time t_{avg} for G0 is 1.75 min, for G1 0.70 min, for G2 0.28 min and for G3 0.60 min. G2 is by far the most efficient test generator. G1 and G3 have a similar efficiency despite the fact that G1 uses AC3R which focuses on generating very compact test scenarios and should result in short execution times whereas G3 uses a GA to evolve tracks. G0 is clearly the least efficient test generator although G2 claims to use ASFAULT the same way. Figure 19 groups the generated tests by test generator and compares the number of generated tests based on the available time budget of the challenge test execution. When comparing the number of generated tests in executions with different time budgets the number of generated tests within 30 min is about three times as high as within 10 min. This suggests that the efficiency of the test generators does not decrease the longer they run although some of them should rely on feedback data. A validation of this statement requires many more executions of the challenge which is not done during this thesis.

7.2.5 Complexity of Test Cases

Metrics about the complexity of generated tests allow to estimate how difficult it is for an AV to succeed a test. Depending on which properties of an AV have to be tested different aspects

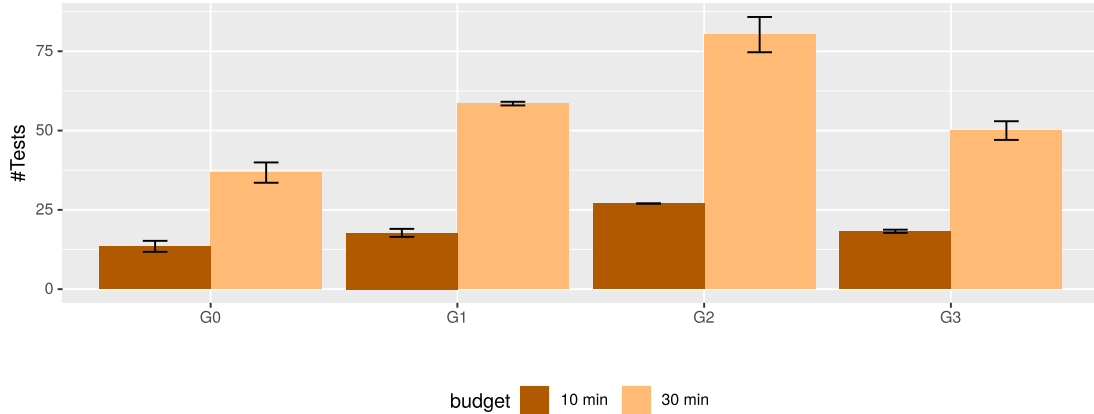


Figure 19: Number of generated tests — Shows how many valid and invalid tests each of the test generators generated in total during the challenge test. The diagram groups them by the time budget available in a round. It also shows their standard deviation.

of complexity are considered. Since most of the submitted test generators test lane keeping capabilities the evaluation investigates the complexity of the generated environments and how far AVs traveled. For the metrics I use amongst others the number of scenario elements i. e. participants, obstacles and roads plus the complexity of the curvature and the length of the generated roads.

Figure 20 visualizes the number of scenario elements. Every test generated by G0, G2 or G3 declares exactly one road and one participant. In contrast to any other generator G1 often defines two roads and one participant on each road. This is due to the fact that G1 uses crash reports to generate test cases and the crash reports that the underlying database provides most commonly involve two participants and often intersections. G2 is the only generator that adds static obstacles to its tests since the student who implemented it had the explicitly the task to generate tests with obstacles.

Figure 21 shows the distribution of the lengths of the generated roads. The shorter generated roads are the less potential they have to have a complex and diverse curvature. The longer the generated roads are the longer the execution of a test takes for an AV to succeed. G0, G2 and G3 generate roads of very similar length but G3 has a little less variety in their length. G1 generates much shorter roads which is a result of AC3R which aims to generate very compact test scenarios. This indicates that G1 does not define complex roads.

Figure 22 depicts the distribution of the number of segments that define each generated road. The lower the number of road segments the lower is the potential of the road to have a complex and diverse curvature. The fewer road segments define a road and the longer the road is the more likely it is that the generated curvature differs from what the test generator intended since it relies more on the interpolation done by DRIVEBUILD. Every road generated by G1 has a low number of segments which fits well with the goal of AC3R to create very compact tests. G3 uses a constant number of road segments. The initial population is a set of randomly placed positions which define the segments of a road. Based on this population the GA which G3 applies evolves tracks by repositioning the segments. G2 uses a constant number of road segments as well but the number of segments is much lower. Since the roads are much longer compared to roads of

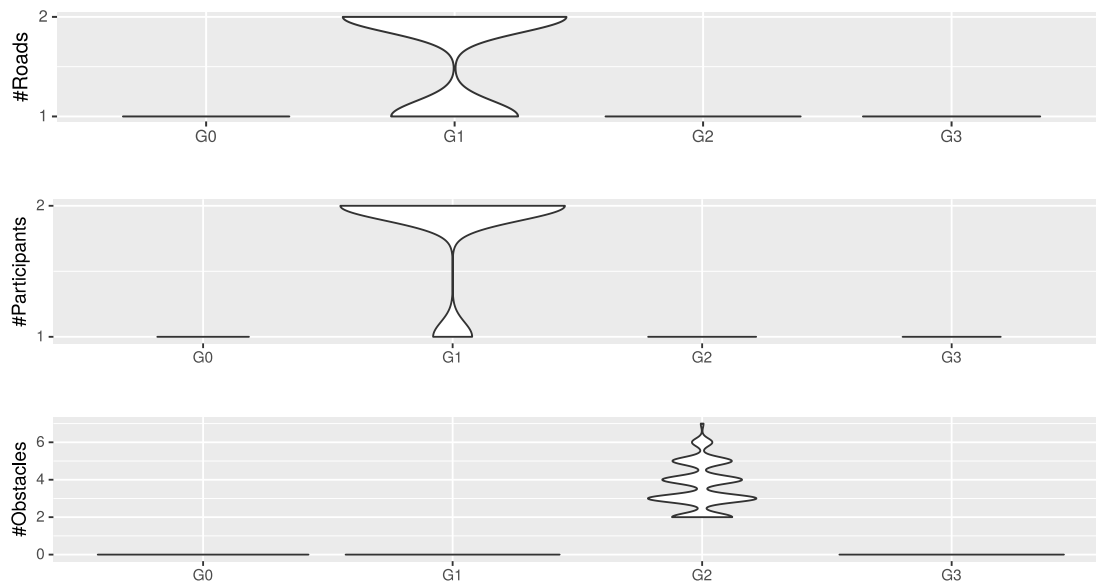


Figure 20: Number of scenario elements — Shows which types of scenario elements like participants, roads and obstacles generated tests included and how many.

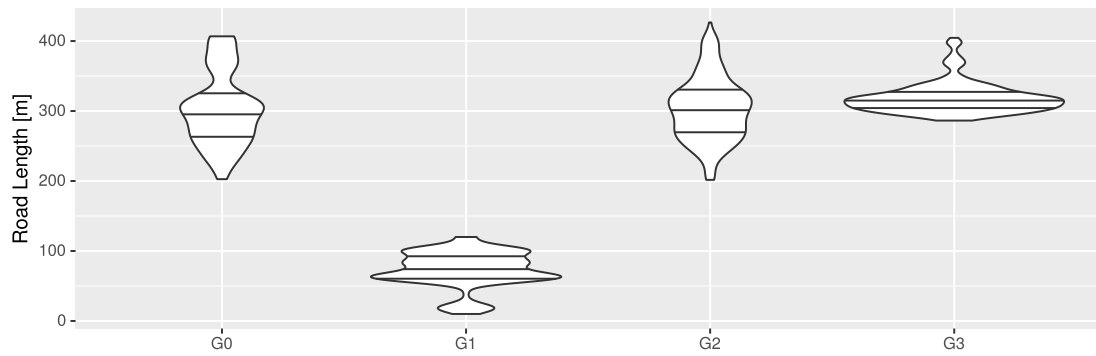


Figure 21: Road lengths — Visualizes the lengths of the generated roads grouped by test generator. The longer a road is the potentially more complex its curvature might be.

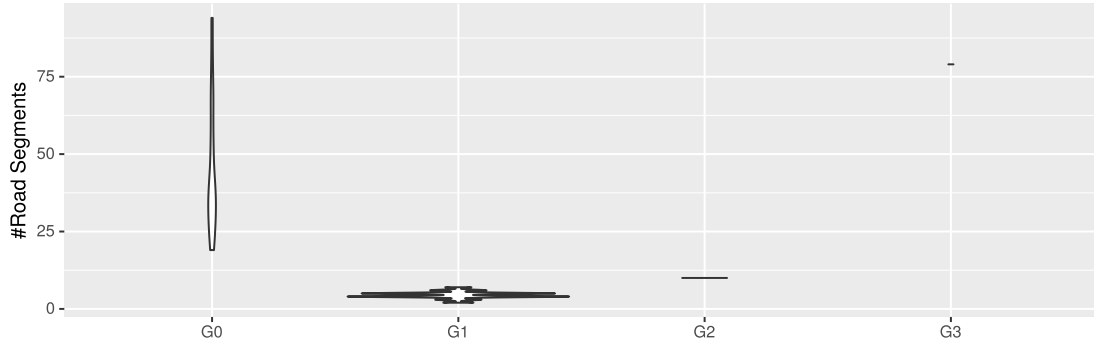


Figure 22: Number of road segments — Depicts the number of road segments that were used for any road generated during the challenge test. It groups them by test generator.

other generators it is likely that they have many long and only slightly winding sections and may not have a very diverse curvature. The constant number of road segments further indicates that the implementation of G2 does not use ASFAULT to generate random roads as it was supposed to do. This becomes especially clear when compared to G0. The number of road segments generated by G0 has a much higher variety. Combining this observation with the fact that the length of the generated roads has a variety similar to G2 and G3 indicates that the generated roads may have in some cases more linear and in others more winding sections.

Figure 23 supports these observations. Further it shows that G1 and G2 restrict the segments of the generated roads to be placed in an area of a size $200\text{ m} \times 200\text{ m}$ whereas G3 only uses an area of size $100\text{ m} \times 100\text{ m}$. In comparison G0 uses a larger area and distributes its lanes over the available area.

To get an impression about the variety of curvatures of the generated roads Figure 24 shows the roads rotated and translated such a way that the first road center point is in the origin and the second is right below of it. The roads generated by G0 consist mainly of one or two big curves and often face in a similar direction. Only occasionally a road has multiple curves that head in different directions and it rarely has sharp turns or long straight sections. G1 generates solely almost short and straight roads as expected from AC3R. The center points of the roads generated by G2 seem to be uniformly and randomly distributed over a predefined area which contradicts with its claim to use ASFAULT for the generation. G3 generates highly diverse and complex roads with many curves having different directions and sizes. The generated roads may even revolve the initial position which makes the roads interesting on an intuitive level. To estimate how interesting the generated roads are future work may use quantitative metrics [32] to further investigate the generated roads.

Figure 25 visualizes how far AVs traveled on generated roads. Comparing Figure 25 with Figure 24 shows that the ratio of the distance AVs under test traveled to the length of the roads is low. This confirms that either the AIs have no good lane keeping capabilities except for A0 which has perfect knowledge about the lanes or the test generators are very effective in finding bugs in the lane keeping capabilities of AIs.

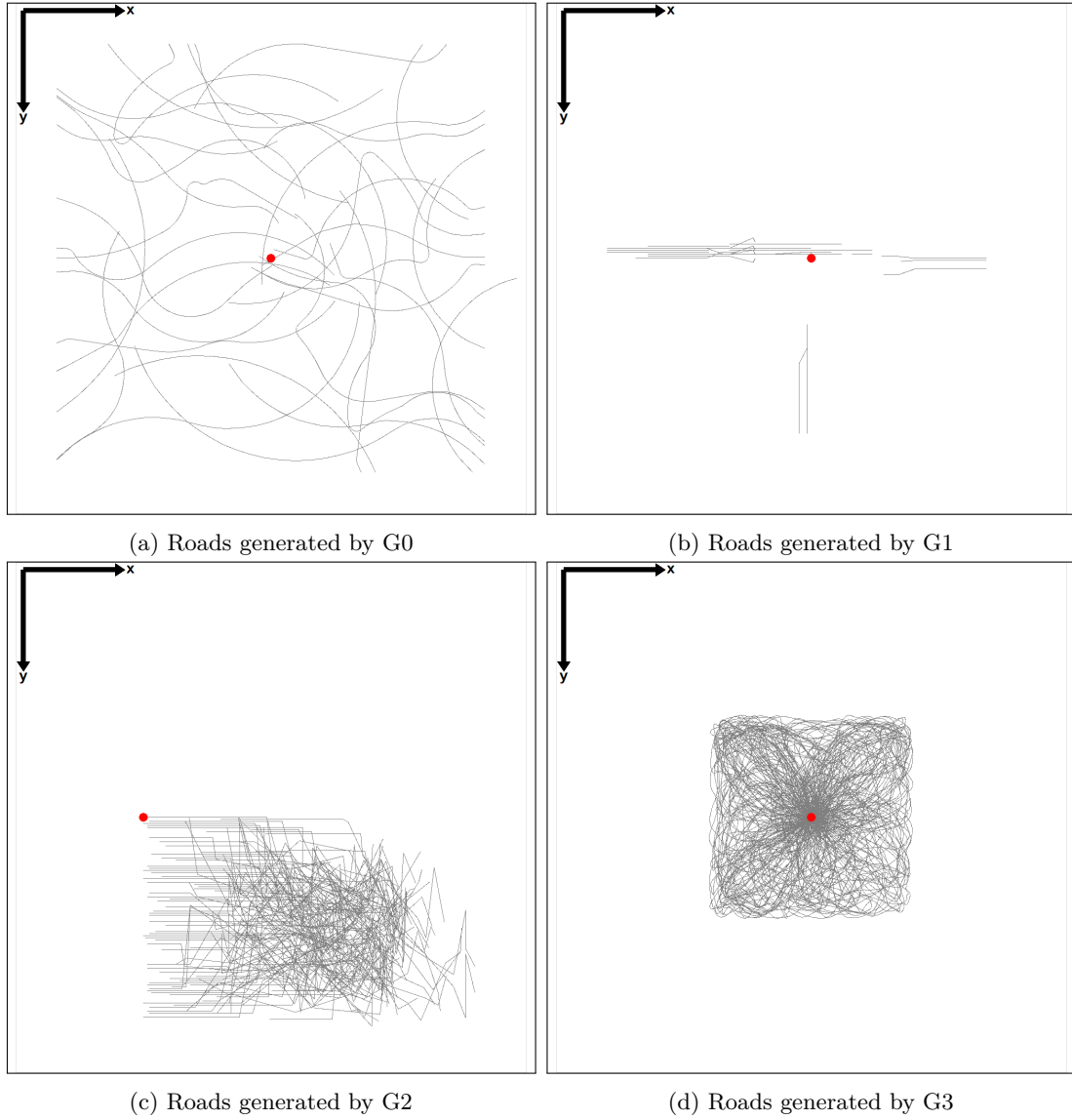


Figure 23: Visualization of roads — Visualizes the segments of all generated roads. Each picture covers all points where $-250 < x < 250$ and $-250 < y < 250$ except for Figure 23c which is shifted to cover all points with $-125 < x < 375$. The red point marks the origin. The points are connected with straight lines instead of interpolating them as they are when starting a simulation.

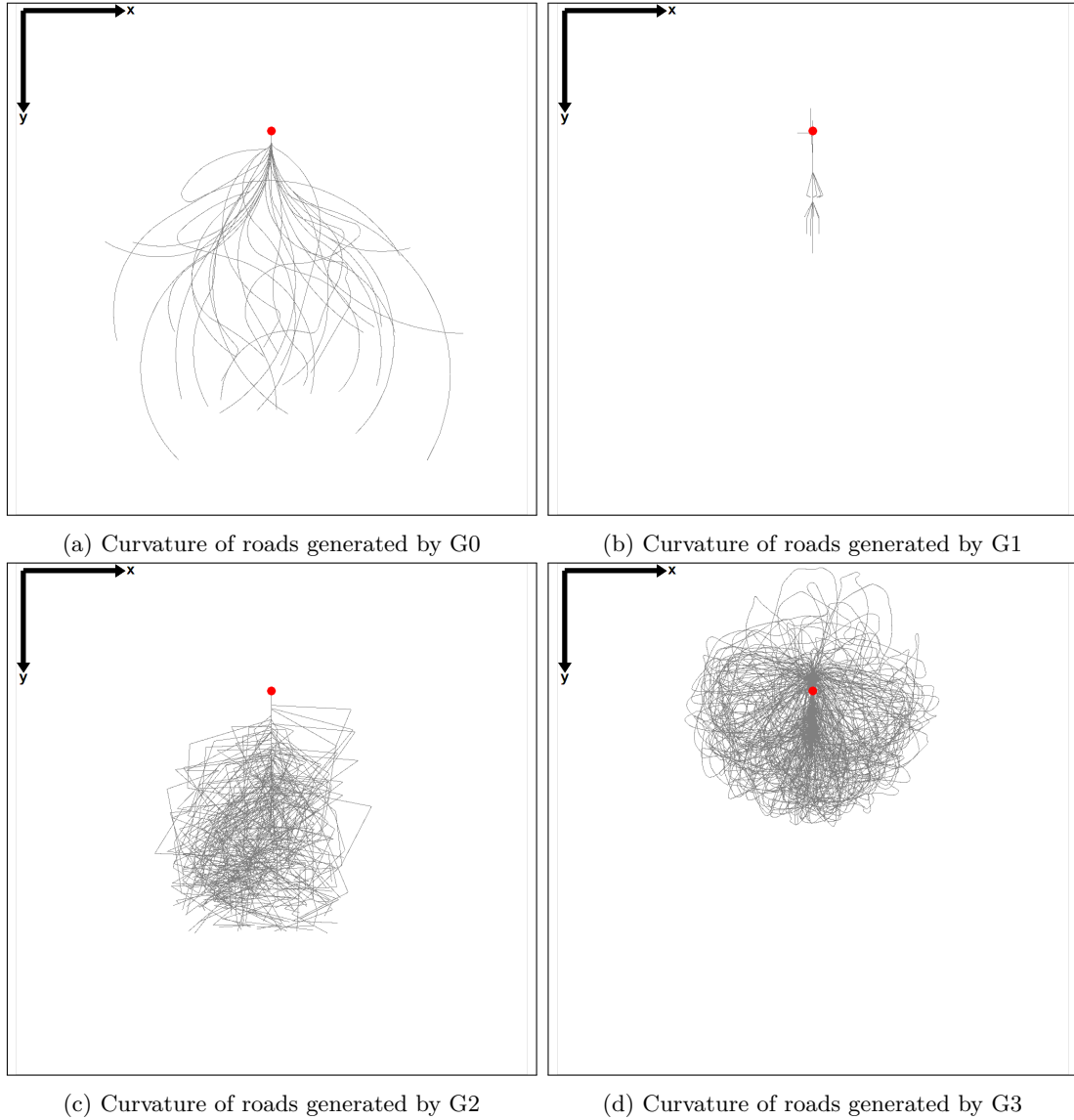
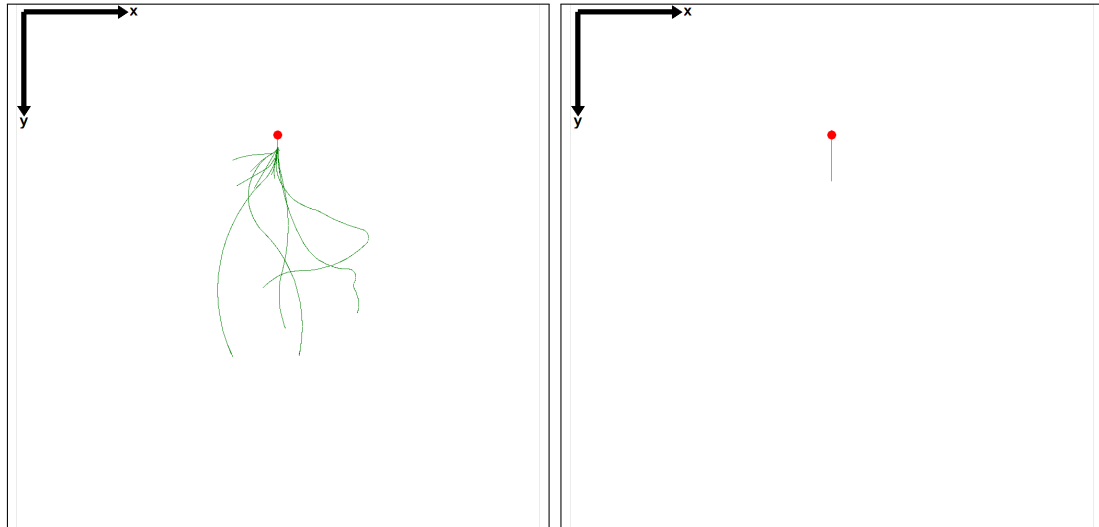
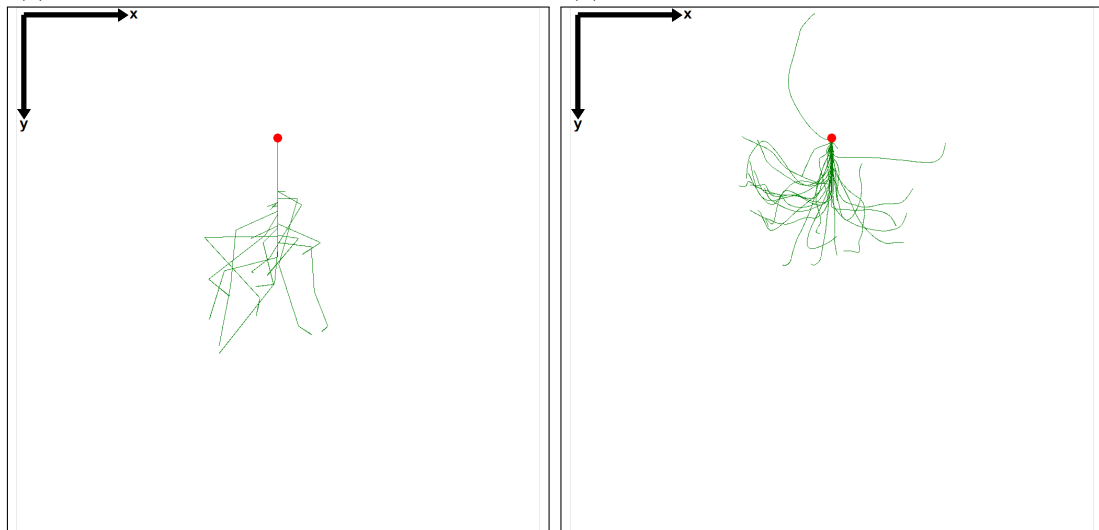


Figure 24: Visualization of road curvatures — Visualizes the segments of all generated roads but equally rotated based on the first two road segments and translated to start at the origin. Each picture covers all points where $-250 < x < 250$ and $-125 < y < 375$. The red point marks the origin.



(a) Traveled distances on roads generated by G0

(b) Traveled distances on roads generated by G1



(c) Traveled distances on roads generated by G2

(d) Traveled distances on roads generated by G3

Figure 25: Visualization of traveled distances — Visualizes the segments of all roads in Figure 24 but depicts only the actually traveled segments. Each picture covers all points where $-250 < x < 250$ and $-125 < y < 375$. The red point marks the origin.

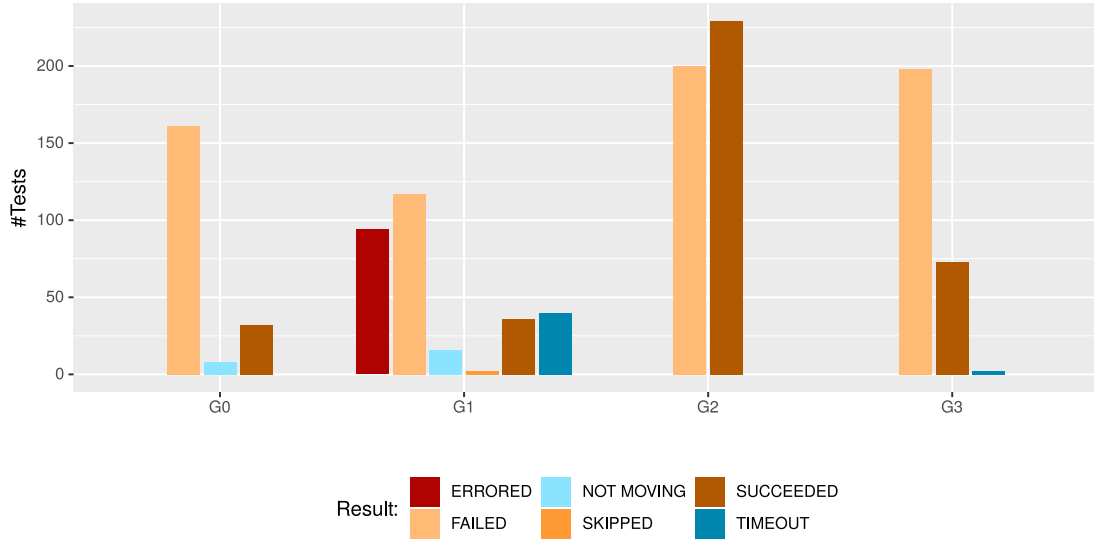


Figure 26: Test results — Depicts the number of generated tests and groups them by test generator. It further distinguishes the number of generated tests based on their test result where **ERRORED** and **NOT MOVING** are no actual test results but denote that the test generator failed to generate a test or the connected AI did not move respectively.

7.2.6 Effectiveness of Test Generators

The effectiveness estimates how effectively a test generator can find bugs in AIs. The effectiveness of test generators is computed by dividing the number of failed tests through the sum of failed and successful tests. The 8 executions of the challenge test yielded in total 1208 generated tests where 1114 were valid and could be executed. Figure 26 and ?? show all test results and group them by either the test generator or the AI. The computation of the effectiveness considers only successful and failed tests but there are more tests. According to Figure 26 G0 and G1 AV did not move in some tests. An AV is considered as non moving if it does not move more than 5 m within the first 60 s of the simulation. These two values are chosen based on a sophisticated guess considering the startup time of BEAMNG and the possible computation overhead of the connected AI. Since according to Figure 27 the only AI that did not move in some tests cases is A0 it is very likely that G0 and G1 did not always define goal positions that fulfill all restrictions inherited by the BEAMNG AI. Additionally many generations of G1 resulted in an error. According to the log of the challenge executions errors occur because the download of a police report fails, AC3R throws an error or AC3R is not able to translate the police report into a comprehensive simulation. In addition many tests result in a timeout which means that a test does neither fail nor succeed within a predefined time. This can happen if the connected AI does not respond i. e. continue with the execution of the client scheme or the success criterion is either infeasible or the AI requires more knowledge about the scenario to fulfill it e. g. a goal area which is behind but not in front of the AV. This timeout consideration applies also for G3. Table 10 lists how many of the executed tests succeeded and failed per test generator as well as the resulting effectiveness. Compared to the other generators G2 is less effective in generating

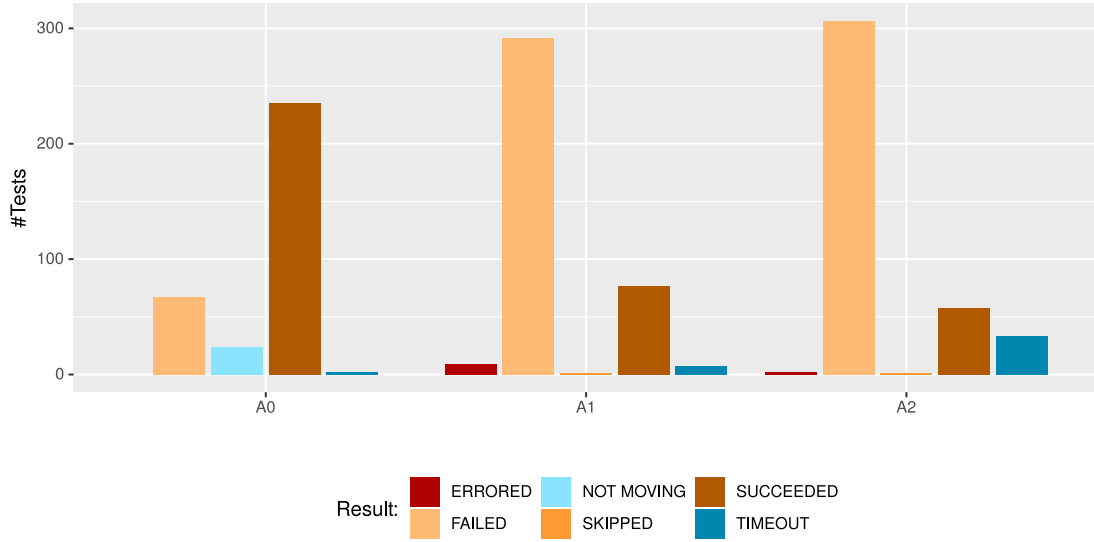


Figure 27: Test results — Divides the executed tests by their result and groups them by the connected AI. The test results **NOT MOVING** and **ERRORED** are no actual test results but denote that an AI did not move or its implementation threw an error.

Table 10: Test generator effectiveness — Displays the calculation of the effectiveness of the test generators. Higher values indicate higher effectiveness.

Property	G0	G1	G2	G3
#Failed	161	117	200	198
#Succeeded	32	36	229	73
Sum	193	153	429	271
Effectiveness	0.83	0.76	0.47	0.73

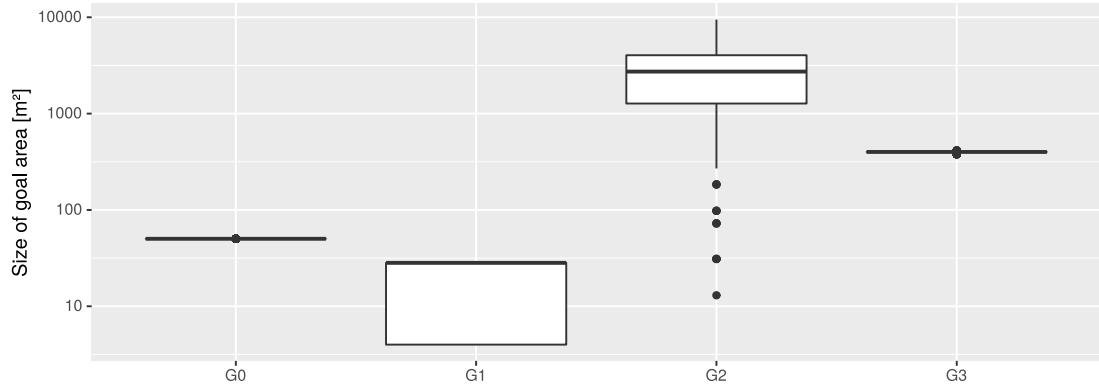


Figure 28: Goal area sizes — Shows the sizes of generated goal areas that vehicles have to reach in order to succeed a test. It only considers valid goal definitions of goal regions.

tests that trigger faulty behavior of AIs although it should be very similar to G0. Both generate always exactly one participant which drives on a single road and both claim to use ASFAULT for generating roads. G2 has in contrast to G0 very small roads and additionally adds obstacles to its test scenarios (see Figure 20). So one would expect G2 to be more effective than G0 since it should be more likely that an AV goes off-road or crash into an obstacle. This is clearly not the case. G0, G1 and G3 have a high effectiveness which mainly results from the bad lane capabilities of the AIs that the short traveled distances (see Figure 25) suggest.

According to Figure 20 G1 is the only generator that adds more than one participant and one road. Since the AIs focus on lane keeping and not on crash avoidance the test generators reveals many bugs and therefore it has a high effectiveness. Figure 28 shows the area sizes of the goal regions (in m^2) the test generators defined for the AV under test to reach. The area of the goal regions G0 generates is constant. To define the goal region in a test G0 selects the road center point of the last segment of the generated road and specifies at this point a goal position having a tolerance radius identical to the width of the road. Since the width was fixed during the challenge test the size of the resulting goal regions is constant. In a similar way G3 chooses one of the road center points and defines the goal region by a square with a fixed edge length of 20 m around the point. So the size of the resulting goal areas is constant as well. G1 generates the smallest goal areas which results from the fact that the generated tests are very compact and do not require big goal areas. G2 generates vastly bigger goal areas which are likely to contain almost the whole environment. A look into the implementation of G2 reveals that it uses the first and the last two road center points to create a huge triangle which then defines the goal region of a test. Hence often an AV almost immediately succeeds a test without driving at all which leads to the low effectiveness of G2. Figure 29 depicts for how long AIs drove in simulations of generated tests and supports this observation.

7.3 Evaluation of Scalability

This experiment is organized in two parts. The first part evaluates the scalability of a single SIMNODE which runs on real hardware. The second part examines whether SIMNODES can be

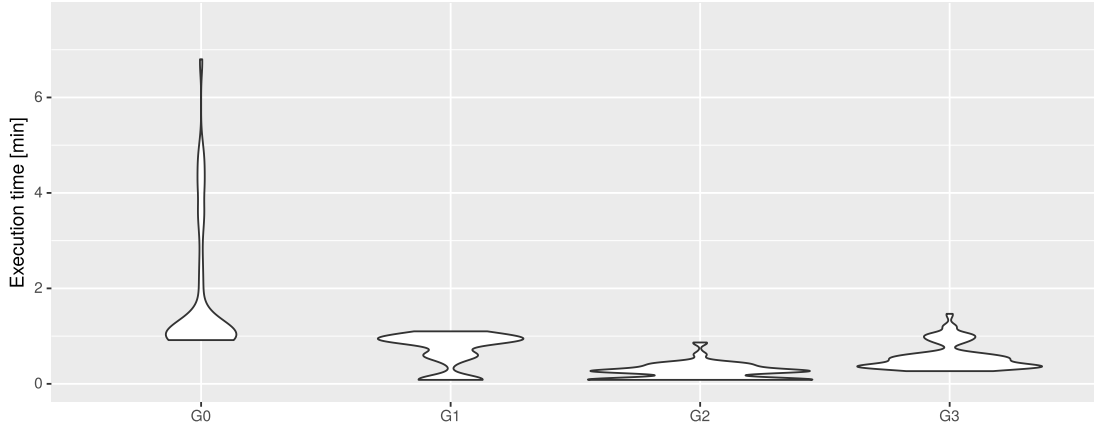


Figure 29: Execution times of AIs — Visualizes the time that AIs drove on tests scenarios until the simulation finished without being canceled and without timeout.

hosted by VMs, whether this influences the results of executed tests and the benefits of distributing simulations across multiple SIMNODES. For the experiment I used a fixed test scenario T which was generated by G0. T defines a single road and places a single AV on it that has to follow the road to succeed the test. To control the AV I used A0 (see Section 7.1.2) and T is configured to request it every 0.5s. The timeout for test executions is fixed to 10 min.

The first part of the experiment used a setup of DRIVEBUILD where node 2 (see Table 8) was the only connected SIMNODE. In each round I created between 1 and 10 instances of T and submitted them to DRIVEBUILD all at once. Figure 30 shows the execution time for each of the 10 rounds. Every execution of T in every round succeeded except for one execution in round 9 which yielded a timeout. When submitting 9 or more instances of T at once the simulations get so slow that some of them may exceed the timeout and thus not succeed as they would if they were less simultaneously running instances of T . To avoid this problem each SIMNODE defines a maximum number of simulations it can handle simultaneously. If every SIMNODE reached the limit HTTP requests for submitting more tests DRIVEBUILD rejects them stating that there were too many active requests. For sake of this experiment I disabled this limit on every SIMNODE. However, the diagram clearly visualizes that DRIVEBUILD is able to utilize parallelism to speedup test executions as this work claimed. Further the regression line suggests that the execution time for less than 11 tests is linear. But much more test executions with even more instances of T are required to support this observation.

Figure 31 shows how long the execution of each single instance of T in any round took to complete. It also visualizes when executions of T started and how they took until they ended. The relative start of the tests per round reveals that the startup time of BEAMNG is about 50s. According to Figure 30 the execution of a single instance of T takes about 210s. Hence the startup of BEAMNG takes almost a quarter of the complete execution time of a single instance. The diagram also shows that the earlier a test in a round is started the longer it takes to complete and tests in the same round often complete almost at the same although they start offset and each test should take the same time. One possible explanation might be that DRIVEBUILD has a bug which e. g. makes test executions synchronize with each other and finish at the same time. I was

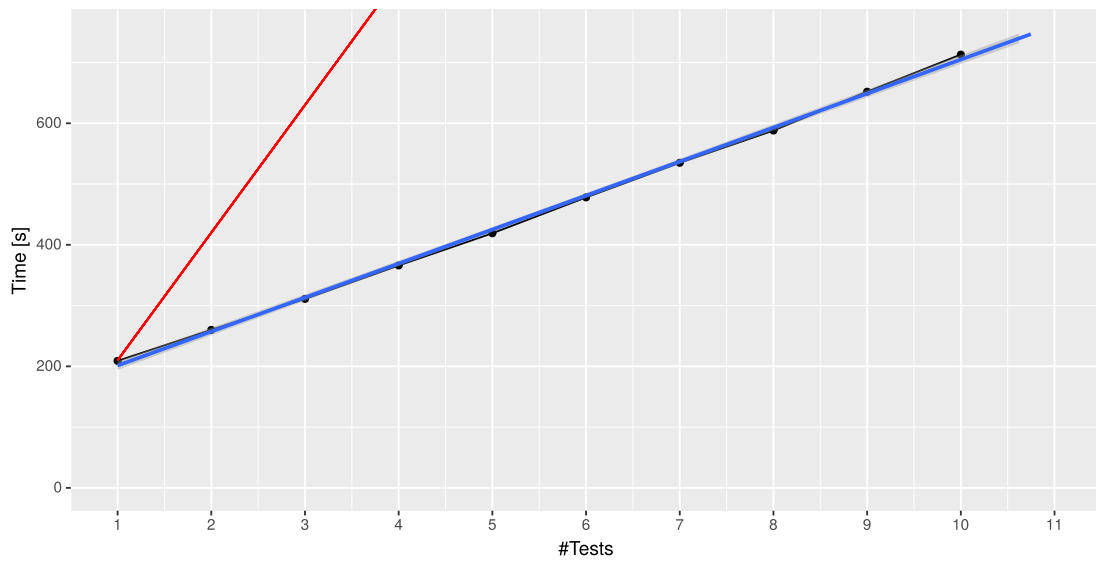


Figure 30: Execution times — Shows the scalability of DRIVEBUILD based on the number of tests submitted at once. It also shows a linear regression line (blue) and the linear projection if the instances of T were executed sequentially (red).

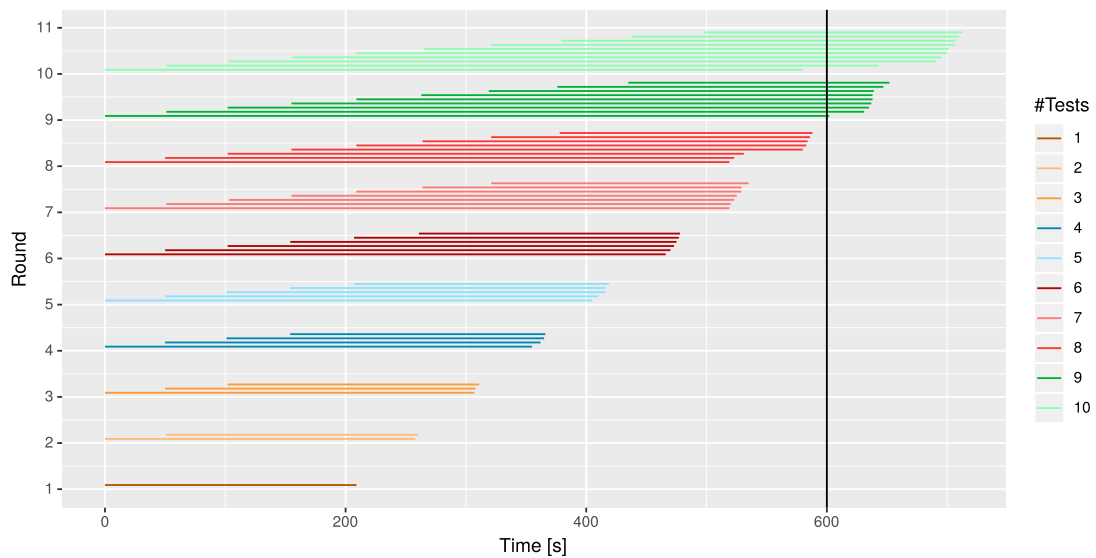


Figure 31: Test executions — Visualizes when tests started and how it took until they ended. It visualizes succeeded tests as well as tests which yielded a timeout. The diagram also includes a vertical line which marks the configured timeout.

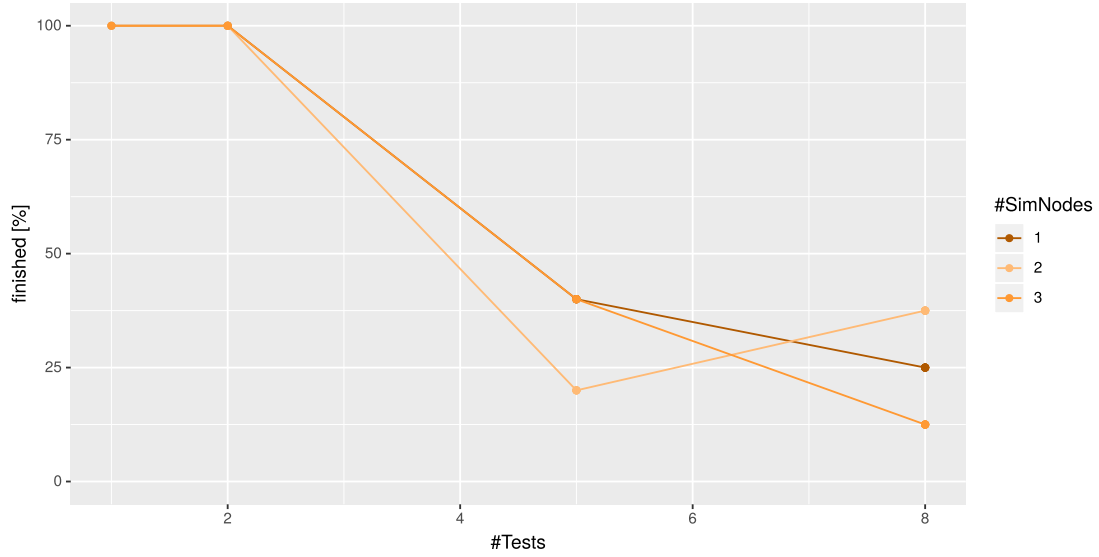


Figure 32: Scalability results for VMs — Shows how many tests finished depending on the number of available SIMNODES and the number of submitted tests. The higher the values the more reliable are test executions on VMs.

not able to verify such a bug through manual test executions. DRIVEBUILD does not implement any explicit prioritization of messages. Hence, another possible explanation might be that the message exchange of DRIVEBUILD assigns higher priorities to certain messages implicitly. A SIMNODE maintains one socket the MAINAPP uses to start an instance of T and additionally maintains two open sockets for each running instance of T . One socket accepts commands for the AV which the client sends over the MAINAPP. This socket yields `waitForSimulatorRequest` and `requestData` (see Table 6) alternately. The other socket accepts commands from the runtime verification. Having this in mind one possible scenario which might lead to the observed results is that starting new tests has implicitly a higher priority than commands from runtime verification cycles e.g. stop a simulation when it succeeded or timed out. To validate these explanations further investigation is required.

The second part of the experiment used a setup with either 1, 2 or 3 SIMNODES which were installed on VMs (see node 3 in Table 8). The VMs are managed using the simple linux utility for resource management (SLURM) [69]. Since BEAMNG is only available for Windows every VM has to run Windows. Further BEAMNG requires a good central processing unit (CPU) as well as a very good GPU. To minimize the expected performance loss SLURM provides direct and isolated access to the GPU. So the GPU is not shared with any other program running on a SLURM node. I create either 1, 2, 5 or 8 instances of T and submit them to DRIVEBUILD at once which yields a total of 12 executions of the experiment with different configurations. Since it took over a week to collect the data for the second part I did not repeat the experiment. In contrast to the results of the first part the test executions result in many timeouts as Figure 32 depicts. This results from the very poor performance of BEAMNG when executed on a VM. On the one hand, the diagram shows that an increasing number of simultaneously submitted tests rapidly decreases the percentage of tests that properly finish. On the other hand, the diagram does not

indicate that a higher number of SIMNODES increases the percentage of properly finished tests. Concerning the executions of the experiment which submitted 8 tests at once 3 tests finished with 2 SIMNODES connected, only 2 finished with a single SIMNODE connected whereas only 1 test finished with 3 SIMNODES connected. This observation may result from the fact that the SLURM cluster which hosts the SIMNODES is shared with other researchers and thus it is used intensively and under heavy but varying load. The observation may be also a result of the low reliability of the low level socket communication between the MAINAPP and the SIMNODES. If a socket is not used for a certain time interval it closes its connection which leads to a timeout of the simulation the socket has to handle messages for. To investigate these problems in more detail much more test executions with SIMNODES that are not hosted in VMs are required.

7.4 Experience Report

7.4.1 Process

The creation of a test generator or an AI is not trivial and requires a good understanding of the concepts to apply. Additionally when it comes to trying out the implementations the students had to deal with many new technologies which are involved in creating the program including libraries, frameworks, development tools as well as to set up a simulator, handle simulations and collect data. To avoid the problems concerning the actual simulations and retrieving data I encouraged students to try out their implementations using DRIVEBUILD. To enable them to use DRIVEBUILD I presented them the basic strategies and concepts DRIVEBUILD implements and showed them short code examples of possible test definitions i. e. DBC and DBE XML files as well as a basic AI implementation. I did neither explain the implementation nor the underlying communication protocol to the students. Instead I provided them a client API which hides the details of the HTTP based communication. I suggested students to use debuggers to get to know how the requested data and the abstract data objects look like. I have been available for them to request help during the implementation of their programs, to request features which address their needs and to provide feedback for DRIVEBUILD. I implemented feature requests like an additional data request for requesting the heading angle and the automatic creation of road markings. I did not implement all of their features requests e. g. weather conditions. The future work section mentions some of the feature requests which are not implemented yet.

7.4.2 Experience

Concerning the fact that the seminar targets master students and bachelor students in higher semesters the level of skill and knowledge of some students was unexpectedly low. The XSD provided with DRIVEBUILD is very long and uses advanced mechanisms including abstract elements and substitution groups and some tried to fully understand it. Some students never heard of XML, never used it and had no idea how to validate XML against XSD. Hence there was no chance for them to get an overview over the whole range of capabilities the test formalization offers. Instead I showed them only the most basic features such as the definition of roads, the placement of participants plus the usage of the two criteria for damage and off-road detection. Many students struggled to use predefined Python libraries which for example provide a tensorflow implementation that they require for their AI implementation.

For the students that implemented test generators one of the most complex topics was XML and its validation with XSD. Many of them used the example definitions I provided almost literally and only selectively modified them. When they tried to introduce the damage and off-road oracles some of them failed since they did not understand how to use logical operators to create compound conditions like “A test fails if an AV is damaged or is off-road”. No one of the students used more than a single connective and no student utilized any kind of nested criteria because this requires at least an intuitive understanding of temporal logic.

The most complex topics for the students implementing AIs were the necessary understanding of the runtime verification cycle and the client scheme plus how these interact with each other. Some students struggled to get an intuitive understanding of parallelism, why it is needed to control multiple AVs at once and to handle parallel threads with `start` and `join` calls. It was further difficult to explain them what data AIs can request, how to declare it for an AV and what the data response looks like.

Some of the students did not succeed to either create something which runs on its own and utilizes BEAMNG or to create at least something which can be integrated with DRIVEBUILD. The integration of the submissions was difficult because many students did not provide a comprehensive list of requirements. Further there are restrictions to minimum and maximum versions of packages, frameworks and Python itself. Especially setting up and integrating AI implementations was very time consuming since these implementations rely on multiple complex libraries. AIs additionally have external dependencies and some of the requirements and dependencies conflicted. Hence the list of dependencies the SUBMISSIONTESTER requires is long (see Table 12). Another problem with AIs is to properly choose values for configuration parameters like initial population sizes since students did not document them. Further the implementation of the interface of AIs is not only required to use the predefined signature but also to implement the client scheme shown in Figure 16.

Multiple students did not or at least not properly implement the interface for the SUBMISSIONTESTER as described in Listings 2 and 3. So I had to create, fix and extend most of the interfaces the students provided to me. To make sure executions of submissions do not interfere with each other the SUBMISSIONTESTER has to separate them. Most of the submissions use paths relative to the current working directory instead of relative to the called file. Thus I also had to change all hard coded paths to make submissions work. Some submissions had bugs resulting from an insufficient understanding of the concepts students should implement. To fix them I was required to have a deep understanding of the strategies the students targeted to implement them myself. The gained experience reveals that there are aspects which could be done or handled differently in future work. One problem is the potentially conflicting dependencies and requirements the submissions of the students introduce to the SUBMISSIONTESTER. To avoid these conflicts the SUBMISSIONTESTER could use separate Python environments for each submission. As a drawback this requires to create and maintain multiple Python environments and to activate and deactivate them dynamically. This also increases the complexity of interacting with submissions and most presumably requires students to implement more complex interfaces. Another approach would be to use DOCKER [70]. Additionally to the advantages like when having multiple Python environments DOCKER enables to start submissions in a clean state and to start multiple instances of the submissions. At the current point it is not clear whether these features are required. The approach also shares the same drawbacks but in a higher form since the effort of creating and maintaining docker containers which include Python environments is higher than maintaining Python environments directly. Further the activation and deactivation as well as the interaction with DOCKER containers is even more complex and most presumably requires students to implement even more complex interfaces. The interfaces for the SUBMISSIONTESTER currently pass test case definitions as paths to files containing XML. Since the test case definitions have to

be modified for the evaluation it would be easier to deal with XML trees directly instead of file paths. The evaluation shows that the reliability of the low level socket communication might be a problem especially when hosting SIMNODES in VMs and should be improved. Another thread to the reliability is BEAMNGPY since it is not designed to cope with parallel requests to the same BEAMNG instance.

8 Conclusions

In this work I developed a distributed system which automates simulation-based testing of AVs. Therefore it presents a formalization of test cases and explains all relevant strategies to generate simulations, handle simulator instances, evaluate test criteria during simulation executions, distribute simulation executions over a cluster and implement the communication between the components of DRIVEBUILD and clients. Its evaluation works with students as users of DRIVEBUILD and utilizes their implementations of test generators and AIs. It showed that DRIVEBUILD is able to cope with all these various approaches. Further DRIVEBUILD provides all metrics that either the implementations require to operate or the evaluation requires to analyze them. Since the number of available test generators and AIs is low and the quality of some of them is not as good as expected the metrics to analyze them yielded not as much interesting information as they could. So future evaluation should consider even more test generators and AIs (e.g. DAVE [55]). The evaluation also illustrates that DRIVEBUILD is able to distribute test executions among a cluster and handle simulations and AVs in parallel. Further it reveals that the performance when running SIMNODES in VMs on a shared cluster is very bad. Research on the causes that affect the performance issues requires much more test executions and dedicated hardware resources. Based on the experience which the evaluation yields possible causes may be low reliability of the low level socket communication or the fact that the hardware is virtualized. However, there are already seminar courses at the university of Passau which use DRIVEBUILD to test AIs. Further there is an ongoing discussion with researchers at the “Université du Québec à Chicoutimi, Canada” about integrating their event stream processing into DRIVEBUILD to implement runtime verification and real-time monitoring. Also at the Zurich University of Applied Science (ZHAW) there is a thesis about testing AVs which utilizes DRIVEBUILD.

9 Future Work

DRIVEBUILD does not provide any visual feedback that shows more than camera images of participants. Visual feedback is very important to get feedback about the generated environments and to get an intuitive understanding of the movement of AVs. For the visualization a format like ROSBAG [54] could be introduced. ROSBAG is a format which stores sequences of pixel clouds to represent a simulation. This allows to visually debug simulations in 3D and rewind a simulation forward and backward. Already a simple visual representation (e.g. a bird view like screenshot) of the environment as well as the initial states, positions and orientations of participants in this environment might be interesting for testers to get an idea about the setup of a test before it is executed. There is already an ongoing project which plans to extend DRIVEBUILD with a debugger which visualizes the simulation and allows to define breakpoints, to pause and resume

simulations plus to dynamically evaluate expressions during a simulation.

Concerning the initial orientation of participants DRIVEBUILD defines angles relative to the ground. The experience of this work shows that for test generators it would be easier to specify the initial orientation of participants relative to the underlying road instead of the ground.

To determine whether an AV is off-road, drives on a certain road or is within a certain region DRIVEBUILD uses either the bounding box or the center position of the AV. In order to cooperate with BEEP BEEP [25] future work should formalize these checks more precisely.

The load balancing of DRIVEBUILD considers only the number of currently running tests on any SIMNODE. A prediction of the resulting load of a simulation based on characteristics of submitted DBEs and DBCs may improve the load balancing. To further improve the load balancing certain messages like “stop” may have a higher priority whereas messages that start new simulations may have a lower priority. The priority of messages may be also dependent on the estimated remaining execution time of running simulations or their progress.

Currently DRIVEBUILD is only able to generate road markings for single roads which do not change their width and the number of lanes. Future work may improve road markings by considering junctions, varying number of lanes, variable road width and more types of road markings including broken or dirty road markings.

A future version of DRIVEBUILD may be able to set different weather conditions including rain, snow and slippery roads to check whether and compare how well AVs can cope with it.

DRIVEBUILD provides training data in terms of traces (see Figure 17). To provide even more training data future work may introduce additional approaches like SCENIC which uses the previously collected data to generate even further new data. Training data may also include the reasons why tests failed or succeeded. Therefore it is required to detect in case the failure or success criterion was triggered which elements of the representing temporal logic expression were involved to fulfill the criterion.

In future versions DRIVEBUILD may also ship with a predefined set of test generators and AIs. Therefore it may be interesting to allow a test setup for an AI to specify restrictions on the generated roads like disallowing self intersection of roads.

10 Acronyms

AC3R automatic crash constructor from crash report	MDP Markov decision problem
ACC adaptive cruise control	ML machine learning
ADAS advanced driver assistance system	NHTSA national highway traffic safety administration
AI artificial intelligence	NLP natural language processing
API application program interface	OS operating system
AV autonomous vehicle	OSM OPENSTREETMAP
AWS Amazon web service	PMessage PROTOBUF message
BMW Bavarian Motor Works	ProtoBuf protocol buffers
CEP complex event processing	RAM random access memory
CNN convolutional neural network	REST representational state transfer
CPU central processing unit	RL reinforcement learning
CRG curved regular grid	ROS robot operation system
DARPA defense advanced research projects agency	ROSBAG ROS bag
DAVE defense advanced research projects agency (DARPA) autonomous vehicle	SaaS service as a service
DBC DRIVEBUILD criterion	SC state condition
DBE DRIVEBUILD environment	SimController simulation controller
DBMS database management system	SimNode simulation node
DDPG deep deterministic policy gradient	SLURM simple linux utility for resource management
DLR deutsches Zentrum für Luft- und Raumfahrt	SML scenario markup language
DSL domain specific language	SOA service oriented architecture
GA genetic algorithm	StatsManager statistics manager
GM general motors	TCManager test case manager
GPS global positioning system	TCP transmission control protocol
GPU graphics processing unit	TORCS the open racing car simulator
HTML hypertext markup language	TTC time to collision
HTTP hypertext transfer protocol	TUM Technical University of Munich
JOSM Java OSM editor	URL universal resource identifier
KPTransformer Kleene-Priest-Transformer	VAE variational encoder
LGSVL LG Silicon Valley Lab	VC validation constraint
LiDAR light detection and ranging	VM virtual machine
LWJGL lightweight java game library	XML extensible markup language
MainApp main application	XSD XML schema definition
	ZHAW Zurich University of Applied Science

11 References

- [1] Chris Neiger. *This Is the Company With the Most Autonomous Miles Driven in 2018*. URL: <https://www.fool.com/investing/2018/12/08/this-is-the-company-with-the-most-autonomous-miles.aspx> (visited on 08/09/2019).
- [2] Daisuke Wakabayashi. *Uber's Self-Driving Cars Were Struggling Before Arizona Crash - The New York Times*. URL: <https://www.nytimes.com/2018/03/23/technology/uber-self-driving-cars-arizona.html> (visited on 08/09/2019).
- [3] John Krafcik. *Where the next 10 million miles will take us*. Oct. 2018. URL: <https://medium.com/waymo/where-the-next-10-million-miles-will-take-us-de51bebb67d3> (visited on 08/09/2019).
- [4] Nidhi Kalra and Susan Paddock. "Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability?" In: *Transportation Research Part A: Policy and Practice* 94 (Dec. 2016), pp. 182–193. DOI: 10.1016/j.tra.2016.09.010. URL: https://www.researchgate.net/publication/308538761_Driving_to_safety_How_many_miles_of_driving_would_it_take_to_demonstrate_autonomous_vehicle_reliability (visited on 08/09/2019).
- [5] David Silver. *Simulation Becomes Increasingly Important For Self-Driving Cars*. Nov. 2018. URL: <https://www.forbes.com/sites/davidsilver/2018/11/01/simulation-becomes-increasingly-important-for-self-driving-cars/#20814aa95583> (visited on 11/22/2019).
- [6] Mathew DeBord. *A Waymo engineer told us why a virtual-world simulation is crucial to the future of self-driving cars*. Aug. 2018. URL: <https://www.businessinsider.de/waymo-engineer-explains-why-testing-self-driving-cars-virtually-is-critical-2018-8?r=US&IR=T> (visited on 11/22/2019).
- [7] Charles Murray. *Autonomous Cars Will Require Years of Test*. July 2018. URL: <https://www.designnews.com/electronics-test/autonomous-cars-will-require-years-test/188554729159097> (visited on 08/09/2019).
- [8] Upamanyu Acharya. *What is Tickrate, and is it Really That Important?* July 2016. URL: <https://fynestuff.com/tickrate/> (visited on 08/09/2019).
- [9] Nancy A. Lynch. "Distributed Algorithms". In: Morgan Kaufmann, Apr. 1996. Chap. Logical Time. ISBN: 9781558603486. URL: <https://www.oreilly.com/library/view/distributed-algorithms/9781558603486/> (visited on 08/09/2019).
- [10] Andrew S. Tanenbaum and David Wetherall. "Computernetzwerke". In: Pearson, 2012. Chap. Referenzmodelle. ISBN: 9783868941371. URL: <https://books.google.de/books?id=uIyZMAEACAAJ> (visited on 01/05/2020).
- [11] Saw Newman. "Building Microservices: Designing Fine-Grained Systems". In: O'Reilly, 2015. Chap. Microservices. ISBN: 9781491950357. URL: <https://books.google.de/books?id=RD14BgAAQBAJ> (visited on 01/07/2020).
- [12] Jiri Soukup and Petr Macháček. "Serialization and Persistent Objects". In: Springer, 2014. Chap. Introduction. ISBN: 978-3-642-39323-5. URL: <https://www.techopedia.com/definition/867/serialization-net> (visited on 01/14/2020).
- [13] Harihara Subramanian, Anupama Raman, and Pethuru Raj. "Architectural Patterns". In: Packt Publishing, Dec. 2017. Chap. Client/Server Multi-Tier Architectural Patterns. ISBN: 9781787287495. URL: <https://www.oreilly.com/library/view/architectural-patterns/9781787287495/> (visited on 01/07/2020).

- [14] VIREs Simulationstechnologie GmbH. *OpenDRIVE*. URL: <http://www.opendrive.org/index.html> (visited on 08/09/2019).
- [15] VIREs Simulationstechnologie GmbH. *OpenCRG*. URL: <http://opencrg.org/> (visited on 08/10/2019).
- [16] VIREs Simulationstechnologie GmbH. *OpenSCENARIO*. URL: <http://www.openscenario.org/> (visited on 08/09/2019).
- [17] M. Althoff, M. Koschi, and S. Manzingler. “CommonRoad: Composable benchmarks for motion planning on roads”. In: *2017 IEEE Intelligent Vehicles Symposium (IV)*. June 2017, pp. 719–726. DOI: 10.1109/IVS.2017.7995802. URL: <https://ieeexplore.ieee.org/document/7995802/> (visited on 08/09/2019).
- [18] P. Bender, J. Ziegler, and C. Stiller. “Lanelets: Efficient map representation for autonomous driving”. In: *2014 IEEE Intelligent Vehicles Symposium Proceedings*. June 2014, pp. 420–425. DOI: 10.1109/IVS.2014.6856487. URL: <https://ieeexplore.ieee.org/document/6856487> (visited on 08/11/2019).
- [19] J. Bach, S. Otten, and E. Sax. “Model based scenario specification for development and test of automated driving functions”. In: *2016 IEEE Intelligent Vehicles Symposium (IV)*. June 2016, pp. 1149–1155. DOI: 10.1109/IVS.2016.7535534. URL: <https://ieeexplore.ieee.org/document/7535534> (visited on 08/09/2019).
- [20] Rodrigo Queiroz, Thorsten Berger, and Krzysztof Czarnecki. “GeoScenario: An Open DSL for Autonomous Driving Scenario Representation”. In: *2019 IEEE Intelligent Vehicles Symposium (IV)*. 2019. URL: https://uwaterloo.ca/waterloo-intelligent-systems-engineering-lab/sites/ca.waterloo-intelligent-systems-engineering-lab/files/uploads/files/iv2019_0501_fi.pdf (visited on 08/11/2019).
- [21] OpenStreetMap Foundation. *OpenStreetMap Wiki*. URL: <https://wiki.openstreetmap.org> (visited on 08/11/2019).
- [22] OpenStreetMap community. *JOSM*. URL: <https://josm.openstreetmap.de/> (visited on 08/11/2019).
- [23] Microsoft Corporation. *Bing Maps*. URL: <https://www.bing.com/maps> (visited on 08/11/2019).
- [24] ESRI Inc. *Home | Esri Deutschland*. URL: <https://www.esri.de/de-de/home> (visited on 08/11/2019).
- [25] S. Hallé and S. Varvaressos. “A Formalization of Complex Event Stream Processing”. In: *2014 IEEE 18th International Enterprise Distributed Object Computing Conference*. Sept. 2014, pp. 2–11. DOI: 10.1109/EDOC.2014.12.
- [26] Daniel J. Fremont et al. “Scenic: A Language for Scenario Specification and Scene Generation”. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA: ACM, 2019, pp. 63–78. ISBN: 978-1-4503-6712-7. DOI: 10.1145/3314221.3314633. URL: <https://math.berkeley.edu/~dfremont/papers/PLDI19.pdf> (visited on 08/10/2019).
- [27] K. Gajananan et al. “Scenario Markup Language for authoring behavioral driver studies in 3D virtual worlds”. In: *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Sept. 2011, pp. 43–46. DOI: 10.1109/VLHCC.2011.6070376. URL: <https://ieeexplore.ieee.org/document/6070376> (visited on 08/11/2019).

- [28] T. Huynh, A. Gambi, and G. Fraser. “AC3R: Automatically Reconstructing Car Crashes from Police Reports”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. May 2019, pp. 31–34. DOI: 10.1109/ICSE-Companion.2019.00031.
- [29] A. Gambi, T. Huynh, and G. Fraser. “Automatically Reconstructing Car Crashes from Police Reports for Testing Self-Driving Cars”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. May 2019, pp. 290–291. DOI: 10.1109/ICSE-Companion.2019.00119. URL: <https://ieeexplore.ieee.org/document/8802678> (visited on 09/28/2019).
- [30] A. Gambi, M. Mueller, and G. Fraser. “AsFault: Testing Self-Driving Car Software Using Search-Based Procedural Content Generation”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. May 2019, pp. 27–30. DOI: 10.1109/ICSE-Companion.2019.00030. URL: <https://ieeexplore.ieee.org/document/8802824> (visited on 09/28/2019).
- [31] 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*. ISSTA 2019. Beijing, China: ACM, 2019, pp. 318–328. ISBN: 978-1-4503-6224-5. DOI: 10.1145/3293882.3330566. URL: <http://doi.acm.org/10.1145/3293882.3330566> (visited on 09/29/2019).
- [32] D. Loiacono, L. Cardamone, and P. L. Lanzi. “Automatic Track Generation for High-End Racing Games Using Evolutionary Computation”. In: *IEEE Transactions on Computational Intelligence and AI in Games* 3.3 (Sept. 2011), pp. 245–259. DOI: 10.1109/TCIAIG.2011.2163692. URL: <https://ieeexplore.ieee.org/document/5975206> (visited on 09/29/2019).
- [33] German Research Center for Artificial Intelligence. *OpenDS*. URL: <https://opends.dfki.de/> (visited on 08/09/2019).
- [34] Advel. *JBullet - Java Port of Bullet Physics Library*. URL: <http://jbullet.advel.cz/> (visited on 08/09/2019).
- [35] jMonkeyEngine Team. *jMonkeyEngine*. URL: <http://jmonkeyengine.org/> (visited on 08/09/2019).
- [36] LWJGL. *LWJGL - Lightweight Java Game Library*. URL: <https://www.lwjgll.org/> (visited on 08/10/2019).
- [37] A. Vernaza, A. Ledezma, and A. Sanchis. “Simul-A2: Agent-based simulator for evaluate ADA systems”. In: *17th International Conference on Information Fusion (FUSION)*. July 2014, pp. 1–7. URL: <https://ieeexplore.ieee.org/document/6916186> (visited on 08/10/2019).
- [38] Olivier Michel. “WebotsTM: Professional Mobile Robot Simulation”. In: *International Journal of Advanced Robotic Systems* 1 (Mar. 2004). DOI: 10.5772/5618. URL: https://www.researchgate.net/publication/221702459_WebotsTM_Professional_Mobile_Robot_Simulation/citation/download (visited on 08/10/2019).
- [39] Morgan Quigley et al. “ROS: an open-source Robot Operating System”. In: vol. 3. Jan. 2009. URL: https://www.researchgate.net/publication/233881999_ROS_an_open-source_Robot_Operating_System/citation/download (visited on 08/10/2019).
- [40] Alexey Dosovitskiy et al. “CARLA: An Open Urban Driving Simulator”. In: *Proceedings of the 1st Annual Conference on Robot Learning*. 2017, pp. 1–16. URL: <http://proceedings.mlr.press/v78/dosovitskiy17a/dosovitskiy17a.pdf> (visited on 10/12/2019).

- [41] Shital Shah et al. “AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles”. In: *Field and Service Robotics*. 2017. eprint: arXiv:1705.05065. URL: <https://arxiv.org/abs/1705.05065> (visited on 10/14/2019).
- [42] Epic Games. *What is Unreal Engine 4*. URL: <https://www.unrealengine.com/> (visited on 10/14/2019).
- [43] Bernhard Wymann et al. *TORCS: The open racing car simulator*. 2015. URL: <https://pdfs.semanticscholar.org/b9c4/d931665ec87c16fcd44cae8fdaec1215e81e.pdf> (visited on 10/13/2019).
- [44] S. Moten et al. “X-in-the-loop advanced driving simulation platform for the design, development, testing and validation of ADAS”. In: *2018 IEEE Intelligent Vehicles Symposium (IV)*. June 2018, pp. 1–6. DOI: 10.1109/IVS.2018.8500409. URL: <https://ieeexplore.ieee.org/document/8500409> (visited on 11/08/2019).
- [45] BeamNG GmbH. *BeamNG*. URL: <https://beamng.gmbh/research/> (visited on 08/09/2019).
- [46] Autonomoose Team. *Autonomoose*. URL: <https://www.autonomoose.net/> (visited on 11/08/2019).
- [47] Inc. Amazon Web Services. *AWS DeepRacer - the fastest way to get rolling with machine learning*. URL: <https://aws.amazon.com/deepracer/> (visited on 11/22/2019).
- [48] Inc. Metamoto. *Metamoto*. URL: <https://www.metamoto.com/> (visited on 10/17/2019).
- [49] Rupak Majumdar et al. *Paracosm: A Language and Tool for Testing Autonomous Driving Systems*. Feb. 2019. URL: <https://arxiv.org/pdf/1902.01084.pdf> (visited on 08/09/2019).
- [50] Baidu. *Apollo*. URL: <http://apollo.auto/> (visited on 08/09/2019).
- [51] The Autoware Foundation. *Autoware.AI*. URL: <https://www.autoware.ai/> (visited on 08/09/2019).
- [52] S. Kato et al. “An Open Approach to Autonomous Vehicles”. In: *IEEE Micro* 35.6 (Nov. 2015), pp. 60–68. ISSN: 0272-1732. DOI: 10.1109/MM.2015.133. URL: <https://ieeexplore.ieee.org/document/7368032> (visited on 08/09/2019).
- [53] Shinpei Kato et al. “Autoware on Board: Enabling Autonomous Vehicles with Embedded Systems”. In: *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems*. ICCPS ’18. Porto, Portugal: IEEE Press, 2018, pp. 287–296. ISBN: 978-1-5386-5301-2. DOI: 10.1109/ICCPS.2018.00035. URL: <https://doi.org/10.1109/ICCPS.2018.00035> (visited on 08/09/2019).
- [54] Tim Field, Jeremy Leibs, and James Bowman. *rosvbag - ROS wiki*. URL: <http://wiki.ros.org/rosvbag> (visited on 08/09/2019).
- [55] Mariusz Bojarski et al. “End to End Learning for Self-Driving Cars”. In: Apr. 2016. eprint: arXiv:1604.07316v1. URL: <https://arxiv.org/pdf/1604.07316v1.pdf> (visited on 11/07/2019).
- [56] C. Chen et al. “DeepDriving: Learning Affordance for Direct Perception in Autonomous Driving”. In: *2015 IEEE International Conference on Computer Vision (ICCV)*. Dec. 2015, pp. 2722–2730. DOI: 10.1109/ICCV.2015.312. URL: <https://ieeexplore.ieee.org/document/7410669> (visited on 09/29/2019).
- [57] A. Kendall et al. “Learning to Drive in a Day”. In: *2019 International Conference on Robotics and Automation (ICRA)*. May 2019, pp. 8248–8254. DOI: 10.1109/ICRA.2019.8793742. URL: <https://ieeexplore.ieee.org/document/8793742> (visited on 09/29/2019).

- [58] Inc. Scale AI. *Training and Validation Data for Self Driving Cars - Scale*. URL: <https://scale.com/self-driving-cars?ref=producthunt> (visited on 11/22/2019).
- [59] Stephen Cole Kleene. *Introduction to Metamathematics*. Wolters-Noordhoff, 1974. ISBN: 0-7204-2103-9.
- [60] Marc Müller. *Github - BeamNG/BeamNGpy*. URL: <https://github.com/BeamNG/BeamNGpy> (visited on 08/09/2019).
- [61] Armin Ronacher. *Flask (A Python Microframework)*. URL: <http://flask.pocoo.org/> (visited on 08/09/2019).
- [62] Ian Bicking Stephan Behnel Martijn Faassen. *lxml - Processing XML and HTML with Python*. URL: <https://lxml.de/index.html> (visited on 08/23/2019).
- [63] Daniel Veillard. *The XML C parser and toolkit of Gnome*. URL: <http://xmlsoft.org/> (visited on 08/24/2019).
- [64] Daniel Veillard. *libxslt*. URL: <http://www.xmlsoft.org/libxslt/> (visited on 08/24/2019).
- [65] Google. *Protocol Buffers | Google Developers*. URL: <https://developers.google.com/protocol-buffers/> (visited on 08/09/2019).
- [66] The Winpython Development Team. *WinPython*. URL: <https://winpython.github.io/> (visited on 08/28/2019).
- [67] Chris Richardson. *What are microservices?* URL: <https://microservices.io/> (visited on 08/09/2019).
- [68] SciPy developers. *SciPy library — SciPy.org*. URL: <https://www.scipy.org/scipylib/index.html> (visited on 08/24/2019).
- [69] Andy B. Yoo, Morris A. Jette, and Mark Grondona. “SLURM: Simple Linux Utility for Resource Management”. In: *Job Scheduling Strategies for Parallel Processing*. Ed. by Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60. ISBN: 978-3-540-39727-4. URL: <https://www.springer.com/de/book/9783540204053> (visited on 11/07/2019).
- [70] Docker Inc. *Enterprise Container Platform | Docker*. URL: <https://www.docker.com/> (visited on 10/29/2019).

A Appendix

A.1 Code Example Snippets

```
<?xml version="1.0" encoding="UTF-8" ?>
<environment xmlns="http://drivebuild.com"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://drivebuild.com drivebuild.xsd">
  <author>Stefan Huber</author>
  <timeOfDay>0</timeOfDay>

  <lanes>
    <lane markings="true">
      <laneSegment x="0" y="0" width="8"/>
      <laneSegment x="50" y="0" width="8"/>
      <laneSegment x="80" y="20" width="8"/>
      <laneSegment x="100" y="20" width="8"/>
    </lane>
    <lane>
      <laneSegment x="50" y="-20" width="6"/>
      <laneSegment x="30" y="50" width="6"/>
    </lane>
  </lanes>

  <obstacles>
    <!-- [...] Obstacle definitions -->
  </obstacles>
</environment>
```

Listing 4: Example environment description — This example shows a basic environment description defining two roads. Figure 2 shows the resulting generated roads.

```

<?xml version="1.0" encoding="UTF-8" ?>
<criteria xmlns="http://drivebuild.com"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://drivebuild.com drivebuild.xsd">
  <author>Stefan Huber</author>
  <version>1</version>
  <name>Test A</name>
  <environment>environmentA.dbe.xml</environment>
  <stepsPerSecond>10</stepsPerSecond>
  <aiFrequency>50</aiFrequency>

  <participants>
    <participant id="ego" model="ETK800">
      <initialState x="1" y="-3" movementMode="MANUAL" orientation="-38"
        ↪ speedLimit="30"/>
      <ai>
        <!-- [...] AI request data definition -->
      </ai>
      <movement>
        <waypoint x="25" y="-12" movementMode="MANUAL" tolerance="1"/>
        <waypoint x="53" y="0" movementMode="MANUAL" tolerance="1"/>
        <waypoint x="80" y="18" movementMode="MANUAL" tolerance="1"/>
        <waypoint x="98" y="19" movementMode="MANUAL" tolerance="1"/>
      </movement>
    </participant>
    <participant id="nonEgo" model="ETK800">
      <initialState x="50" y="-27" movementMode="MANUAL"
        ↪ orientation="110" speedLimit="10"/>
      <movement>
        <waypoint x="30" y="20" movementMode="MANUAL" tolerance="1"/>
      </movement>
    </participant>
  </participants>

  <!-- [...] Criteria definition -->
</criteria>

```

Listing 5: Example participant description — This example shows an example of positioning a participant, defining its movement and which data its AI requires.

```

<cube x="105" y="25" width="1" length="10" height="8"/>
<cylinder x="60" y="-10" radius="5" height="7"/>
<cone x="40" y="10" height="10" baseRadius="6"/>
<bump x="20" y="-8" width="1.5" length="5" height="0.1" upperLength="4.5"
↳ upperWidth="1"/>

```

Listing 6: Example obstacle definitions — This snippet demonstrates the creation of all available types of static obstacles. Figure 2 shows a visualization of generated obstacles.

```

<boundingBox id="<someRequestID>" />
<camera id="<someRequestID>" width="800" height="600" fov="60"
↳ direction="FRONT" />
<carToLaneAngle id="<someRequestID>" /> <!-- FIXME Check whether this is
↳ actually implemented -->
<damage id="<someRequestID>" />
<laneCenterDistance id="<someRequestID>" />
<lidar id="<someRequestID>" radius="100" />
<light id="<someRequestID>" /> <!-- FIXME Check whether this is actually
↳ implemented -->
<position id="<someRequestID>" />
<speed id="<someRequestID>" />
<steeringAngle id="<someRequestID>" />

```

Listing 7: Examples for AI request data — This snippet shows all supported types of data that an AV can request.

```

<!-- FIXME Check whether these are all implemented -->
<xArea participant="<someParticipantID>"
↳ points="(<x1>,<y1>);(<x2>,<y2>);[...];(<xn>,<yn>)" />
<xDamage participant="<someParticipantID>" />
<xDistance participant="<someParticipantID>" to="<someParticipantID>"
↳ max="<maxDistance>" />
<xLane participant="<someParticipantID>" onLane="<someLaneIdOrOffroad>" />
<xPosition participant="<someParticipantID>" x="<x>" y="<y>"
↳ tolerance="<tolerance>" />
<xSpeed participant="<someParticipantID>" limit="<limit>" />
<vcTime from="<fromTick>" to="<toTick>" />
<vcTTC participant="<someParticipantID>" to="<someParticipantID>"
↳ max="<maxTime>" />

```

Listing 8: Example criteria definition — This snippet shows example definitions for all types of criteria listed in Table 3. The tag names are built from either the prefix “vc” or “sc” and the type name of the criterion. To spare duplication the prefix “x” is used here to denote that the prefix might be either “vc” or “sc”.

A.2 Determine Target Position for A0

The test generators did not obey the restrictions to the target position that A0 (see Section 7.1.2) introduces. So in case of A0 the SUBMISSIONTESTER has to replace the declared target position with a target position which is valid for A0. Therefore the SUBMISSIONTESTER collects all `scPosition` elements within the tag of the success criterion which are associated with the AV. If there is exactly one element this defines the target position. If there are more than one the generated test is considered invalid since the test seems to have a branch in its result. If there are no `scPosition` elements the SUBMISSIONTESTER collects all associated `scArea` elements within the success criterion that are associated with the AV. If there is more than one element the test is again considered invalid since it seems to have a branch in its result. Only if there is exactly one such element the SUBMISSIONTESTER computes the intersection with all roads in the DBE and determines all road center points that lie within. The SUBMISSIONTESTER uses one of these points as target position. If there is no such road center point or there was no associated `scArea` the SUBMISSIONTESTER uses the last waypoint of the movement which the DBC specifies for the AV.

A.3 Used Tools

Table 11: Used tools and libraries — This table lists all tools, frameworks and libraries that the components of DRIVEBUILD use or a client implementation needs.

Name	Version	MainApp	SimNode	Client
BeamNG.research	1.6.1	×	✓	×
beamngpy	1.13	×	✓	×
dill	0.3.0	✓	✓	✓
Flask	1.1.1	✓	×	✓
lxml	4.4.1	×	✓	×
numpy	1.17.0	×	✓	×
pg8000	1.13.2	✓	✓	×
pip	19.0.3	✓	✓	✓
PostgreSQL	42.2.5	×	×	×
protobuf	3.9.1	✓	✓	✓
Python	3.7	✓	×	✓
scipy	1.3.1	×	✓	×
setuptools	40.8.0	✓	✓	✓
Shapely	1.6.4.post2	×	✓	×
WinPython	3.7.4	×	✓	×

Table 12: Used tools — This table lists all Python packages, frameworks and tools the SUBMISSIONTESTER requires to run the submitted programs and the evaluation.

Name	Version
AC3R	—
Anaconda	2018.12

asfault	0.0.post0.dev29+g41c537e
beamngpy	1.14.1
dataclasses	0.6
deap	1.3.0
decorator	4.4.0
descartes	1.1.0
flask	1.1.1
gym	0.14.0
jinjja2	2.10.1
lxml	4.4.1
matplotlib	3.1.1
mpi4py	3.0.2
networkx	2.3
numpy	1.16.0
pg8000	1.13.2
pillow	6.2.0
pip	19.2.3
protobuf	3.9.2
pydotplus	2.0.2
pyqt4	1.0.0
python	3.6.9
python—dateutil	2.8.0
pyyaml	5.1.2
requests	2.22.0
setuptools	41.2.0
shapely	1.6.4
sklearn	0.0
stable—baselines	2.7.0
tensorflow	1.14.0
wheel	0.33.6

Eigenständigkeitserklärung

Hiermit bestätige ich _____(Name), dass ich die vorliegende Arbeit selbstständig und ohne unzulässige Hilfe verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die wörtlich und sinngemäß übernommenen Passagen aus anderen Werken kenntlich gemacht habe. Die Arbeit ist weder von mir noch von einer anderen Person an der Universität Passau oder an einer anderen Hochschule zur Erlangung eines akademischen Grades bereits eingereicht worden.

Ort, Datum

Unterschrift