Passau University

Master Thesis

DriveCloud An efficient cloud for Simulation-based Testing of Autonomous Vehicles

Author:
Stefan Huber

Supervisor:
Prof. Dr.-Ing. Gordon Fraser

Advisor:
Alessio Gambi, Ph.D.

Tuesday 26th September, 2017

Abstract

Running many simulation based tests in the context of autonomous cars is challenging due to a costly process of defining test scenarios, specifying success and fail criteria, setting up a simulation environment and executing them. So, I present a cloud architecture for testing autonomous cars as service ~~as a service~~ (SaaS). This approach allows to completely automate the process of executing simulation based tests, enables execution of many concurrent test cases and requires testers to buy neither any hardware nor an expensive simulator. Additionally this system enables researchers to collect test scenarios and their simulation results to define benchmark suites, to identify very critical test cases and to compare artificial intelligences (AIs) running the same test scenarios. Therefore the cloud system also provides a predefined set of test cases.

1 Introduction and Motivation

The progress in developing autonomous cars over the past years is impressive. One of the main problems is still testing them.
To assure the safety of self driving vehicles it would in theory be necessary to drive millions of kilometers on real streets. Whenever something of the hard- or software changes this whole process would need to start over again which is definitely infeasible. Furthermore untested autonomous cars would endanger other traffic participants and lead to accidents, injuries and possibly to death. Companies and researchers circumvent this by using simulators. Simulators additionally allow to execute many tests concurrently and to put a car into predefined ~~enforced~~ situations.
However, executing many tests ~~in parallel~~ requires ~~much~~ computational power and an infrastructure to manage them. To solve this cloud computing can be used.

1.1 Problem Statement

?⟨sec:problem⟩?

When ~~trying to~~ use cloud computing for simulation based testing of autonomous vehicles many problems arise.

1.1.1 Test case definition

The first major problem is the definition of test cases that are also referred to as test scenarios. Each scenario has to define an environment, the test criteria, the behavior of participants and the communication points with AIs controlling one or more of the traffic participants within the scenario.
An environment needs to describe lanes, roads, intersections, moving participants and static obstacles. Participants as well as obstacles have to have properties including initial position, orientation, size and in case of obstacles their shape.
The test criteria must allow to express conditions about the current state of any traffic participant (state conditions) which includes at least damage, speed and position. To restrict when state conditions have to be considered validation constraints (e. g. position and time) on top of state conditions are needed. Combining state conditions and validation constraints allows to formalize a test criterion like “A car must not succeed a certain speed limit while driving on a certain road”. Furthermore it is not sufficient to define either success or failure criteria since in the context of autonomous driving a test case can not be considered as successful only because it did not fail or got interrupted.
The test case definition has to be able to control the behavior of traffic participants either manually by listing a number of positions and forcing a car to follow or automatically by fixing a destination point and telling an AI to go there. This is mandatory to describe behaviors like “Force the car to change the lane and tell it to go back safely.”. The test scenario further has to be capable of mixing these two control mechanisms.

1.1.2 AI communication

The second major problem is the communication with AIs. Since users of the system may not want to expose their AI implementations to the cloud system a mechanism for requesting AIs being outside the cloud is required. This communication protocol must provide a way to specify which data the AI needs and has to make sure multiple requests do not interfere each other. The protocol further has to be able to control the execution of the simulation and the test runs. It also needs the capability of including additional information the simulation can not or does not provide e. g. camera images or extra sensor data.

1.1.3 Cloud infrastructure

The third major problem is the cloud infrastructure. For providing computational resources to as many users as possible it has to be scalable and implement load balancing. Furthermore the cloud has to offer elasticity to dynamically allocate and free resources. Both load balancing and elasticity require to predict the expected load based on a given test case in order to maximize their benefits. To allow performance tests of an AI and to check whether it is responsive enough to handle a self driving vehicle while driving the cloud infrastructure as well as the communication protocol has to fulfill real time requirements. Further the test runs have to be isolated from each other to prohibit falsification of test results through interference. The cloud needs mechanisms for collecting test results and granting access to statistics and to user contributed test scenarios. The collect mechanism has to have a strategy of deciding whether to keep or drop gathered test scenarios so that the database is simply not getting flooded. Consequently the cloud shall be able to identify critical test scenarios and also should suggest these to all other users. It must be possible to compare AIs a user tested or even compare AIs of other users if they applied them on the same test scenarios. To collect and compare as many interesting test scenarios as possible the cloud needs much storage space and the ability to handle many concurrent requests.

1.1.4 Simulator

The fourth major problem is the simulator. For having precise test results the capability of as realistic physics as possible is mandatory. It is also required that the simulation is deterministic since executing a test multiple times in a row with neither changing the AI nor the test scenario shall yield the same results. However, the simulation must not need to much computational power since this drastically decreases the number of concurrent test runs the cloud can handle properly. Further the simulator needs an interface to on the one hand control the simulation and on the other hand provide data either the evaluation of state conditions and validation constraints require or connected AIs need.

1.1.5 Interface for tester

The fifth major problem is the interface for the tester. It has to provide the whole set of user control including roles, registration, login and user specific storage for saving and accessing test scenarios. It also has to offer controls to add, delete, start and stop test executions and to

monitor all currently running tests. Furthermore to bring a user into the position of being able to evaluate test results and AIs the monitoring must show information like current status of test suites as well as successful, failed, skipped and not yet executed tests and a history of AIs listing test results of previous executions.
The interface shall also provide information about the current utilization of the cloud.

1.1.6 Predefined test scenario templates

The sixth major problem is to provide users a predefined set of test case templates that enable them to run tests without any need to invest time beforehand creating them. There are already many well-tried test scenarios available in the format of OpenDRIVE [13] and CommonRoad [2] scenarios. Since users may want to use these with the cloud it is necessary to develop algorithms for unifying them into a representation which can be used with the system.

1.1.7 Authentication

The seventh major is authentication which affects two parts of the system. The one case is authentication of a tester or researcher. They have to be authenticated to make sure that only they can control/ monitor their test runs and view results of their past simulations. The other case is authentication of AIs. This authentication is mandatory if the system has to make sure that an AI is allowed to send messages only to a certain simulator, that an AI can not be impersonated (accidentally) and that only the right AI receives the appropriate data from a simulator.

2 State of the Art and Technical Background

2.1 State of the Art

There exist some papers and projects that already tackle some aspects of subsection 1.1.

Concerning the definition of test scenario environments OpenDRIVE [13] is one of most popular formats for defining very comprehensive environments and is used by many well known car manufacturers. The format offers declarations of signs, cross falls, parking spaces, bridges and signals which may even dynamically change. Especially the definition of streets and rail roads can be very complex and enhanced with much meta information. E. g. it is possible to define predecessor and successor lanes, neighbor lanes, complex junctions, acceleration strips, side walks, multiple different types of markings, reference lines for roads/junctions and rail road switches. As a consequence the generation of simulation environments out of this format is too expensive for being used within a cloud which focuses on running many simulations in parallel. Although OpenDRIVE has many options to define environments it has neither the capability of adding any traffic participant nor of specifying their movements nor of expressing any criterion related to the actual test.
Another very popular format is CommonRoad [2] 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 describing movements of the vehicle. Each state consists of a time step, a position, an orientation of the participant and its current speed. The speed as well as the position may be not specified exactly but with an interval enabling to formulate uncertainty of these attributes. Since states tight time, position and speed strongly together there is no way to make sure described movements are realistic. Concerning the definition of test criteria CommonRoad is restricted to the definition of goal regions where the car has to get in a successful test.
Paracosm [23] offers test case description combined with a simulation architecture. It is based on a synchronous reactive programming language and the main concept are reactive objects which contain geometric and graphical features of physical objects bundled with their behavior. These are internally represented using 3D meshes. 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. Actual computations on or analysis of data are in this context equivalent to stream transformations. Paracosm also allows sensor data to be shipped with test scenarios e. g. depth images. Furthermore Paracosm is capable of generating almost 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 simulator is not able to precisely reflect physical behaviors. The paper presenting Paracosm [23] is not clear about how AIs can communicate with the simulation and there is no information about any performance measures and whether Paracosm can be executed in parallel or whether its processes can be distributed.

The authors of [21] present a cloud infrastructure which is explicitly geared towards testing autonomous cars. This infrastructure focuses on high resource utilization, high performance and low management overhead. For implementation the authors use Spark [10] for distributed computing, Alluxio [8] for allowing distributed storage and OpenCL [17] for optimizing the performance of the graphics processing unit (GPU) intensive simulations. Furthermore the paper mentions frameworks like Hadoop [9] and reasons about them why the authors used some of them or not. Hence the paper is really interesting for my work although it does not cover problems of communicating with AIs.
CloudI [27] is a basic cloud implementation based on Erlang and has a service oriented architecture (SOA). The main aspects are efficient messaging, fault tolerance and scalability. CloudI comes with support for many programming languages (e. g. Java, C/C++, Python, Elixir, Go and Haskel), many protocols and multiple database management systems (DBMSs). In addition, it provides implementations of routing algorithms and authentication mechanisms. Since it is based on SOA it may neither provide the scalability needed for handling many computational expensive simulations concurrently and the granularity necessary for controlling simulations and exchanging information with AIs.

The open source project Apollo [5] is a comprehensive platform offering a high performance and flexible architecture for the complete life cycle of developing, testing and deploying self driving cars. It supports many types of sensors e. g. light detection and ranging (LiDAR) sensors, cameras, radars and ultrasonic sensors. Apollo ships with software components to localize traffic participants, to percept the environment and to plan routes. Since version 3.5 it additionally comes with a cloud service based simulation platform. Further Apollo contains a web application called DreamView which visualizes the current output of relevant modules, shows the status of hardware components, offers debugging tools, activates or disables modules and control the autonomous vehicle. Apollo does not provide test case criteria considering complete scenarios or

multiple traffic participants.
Autoware [11, 18, 19] is another comprehensive open source project. It builds a complete ecosystem containing algorithms for Localization, perception, detection, prediction and planning, predefined maps and the capability to handle sensors and real vehicles. It also provides the LGSVL simulator which can visualize information like perception data or status of other participants. Autoware works with ROSBAG files [7] which allow to record, replay and debug executed simulations. The environment description used with Autoware are pixel clouds. So to work with Autoware requires to create time consuming pixel clouds and to rely on the perception algorithm since this is the only source of information about the environment.

2.2 Technical Background

Both simulations and test cases in my work are based on logical time namely **ticks** [1]. A video showing a simulation in a simulator has a certain rate of frames per second (FPS) specifying how often the simulator calculates new positions and properties of any object and shows it on the screen each second. Each calculation results in an image referred to as frame that is part of the video and defines a state at a tick in the simulation time.

Synchronous simulation is an execution strategy of a simulator. A simulator working with this strategy calculates a few ticks and pauses. When stopped it calls other programs doing calculations or analysis at the current tick. After these calculations finished the simulator resumes and the process starts over.

Elasticity [16] in context of clouds characterizes the ability of an infrastructure to dynamically allocate or free computational resources depending on the current load of the system. A cloud implementing this property is able to deal with a suddenly increasing number of tests to run.

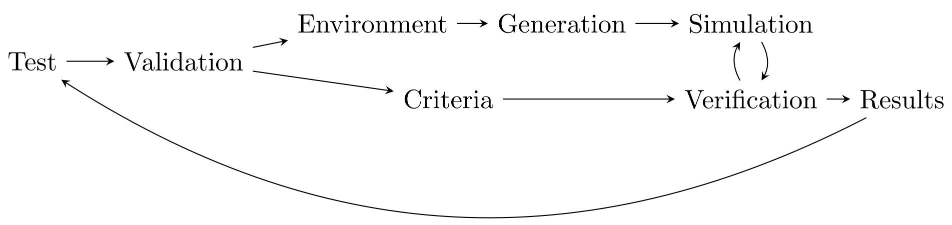
A **micro service** [25] is an architectural design pattern for an application interface that allows to organize an application as a collection of services. A system using this pattern has properties of high maintainability and loose coupling.

3 Proposed Method

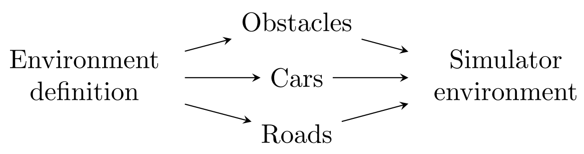
3.1 Test Life Cycle

The underlying test life cycle for the whole system is shown in Figure 1. The first step is the validation of a test. It makes sure that a test scenario definition is not broken or malformed. Each test follows a modular approach separating the test environment and the test criteria to allow reuse of test environments over multiple tests.
The definition of test environments follows a custom format named Drive Cloud Environment (DCE) since currently well known formats like CommonRoad and OpenDRIVE are not precisely suitable as explained in subsection 1.1. Additionally having a perfectly adapted format allows to define minimalistic test scenarios restricted to the information really needed and processed.
The process of generating a simulator compatible representation is shown in Figure 2 and has

?⟨fig:testCycle⟩? Figure 1: Test life cycle — Visualizes the procedure of processing a test definition, creation of a simulator compatible representation, applying runtime verification and evaluation of the simulation.



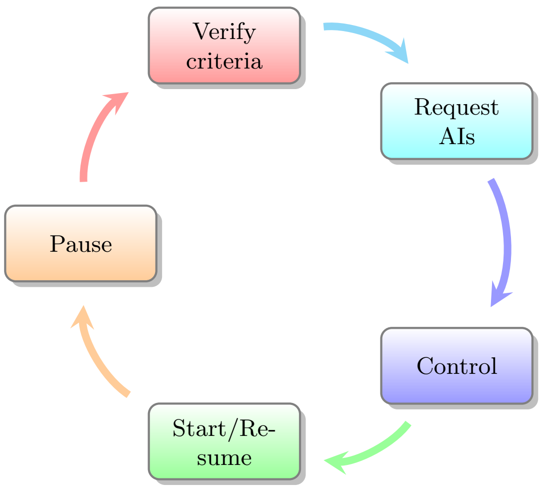
?⟨fig:generation⟩? Figure 2: Generation of environments — This scheme lists the basic types of recognized elements of the test environment.



two phases. In the first phase it extracts obstacles, traffic participants and roads including information about initial position, size and orientation. The second phase converts this data into a representation that the simulator can interpret.
The definition of test criteria also follows a custom format named Drive Cloud Test Suite (DCT) since there is currently no well known scheme available to describe them in a comprehensive and flexible enough way. Each criteria definition has a section for traffic participant paths, preconditions, success and failure conditions. Each of the sections precondition, success and failure define criteria based on state conditions and validation constraints. State conditions are criteria about the current state of a traffic participant including position, damage and speed. Validation constraints allow to restrict under which circumstances state conditions have to be evaluated and considered e. g. time and position. This separation of state conditions and validation constraints enables the definition of very precise criteria like "A test fails if the traffic participant P drives on lane L and exceeds speed of S" i. e. the car exceeds the speed limit of a road. For expressing that state conditions are ignored during evaluation due to a validation constraint a third boolean state is required. This work will use Kleene and Priest logics [20] which introduces an `unknown` state. The paths section describes the movements of participants in the environment. A movement of a vehicle is a list of states describing position/waypoint, target speed and whether a AI should take control over the car from now on. The capability of changing between manual and automated drive mode at every waypoint results in the ability to mix these modes which is required e. g. if a test wants to force a car to change a lane and tell the AI to change back like it is when overtaking another car.
After separating DCE and DCT the generated environment is put into the simulator and the criteria into the verification module After that the simulation starts. The subsection 3.2 describes the interaction of the executed simulation and the verification process in more detail. As soon as the verification module is able to determine whether the test succeeded it stops the simulation and collects the results. These results are returned to the tester who can analyze them.

Figure 3: Runtime verification — Shows the main execution loop of the interaction between a simulation controller and its runtime verification (See Figure 1).

runtimeVerification⟩?



3.2 Test criteria verification

runtimeVerification⟩?

The verification process evaluates the test criteria frequently during a simulation and decides whether a test succeeded or failed. It follows the concept of a synchronous simulation scheme and is shown in Figure 3. When a simulation is started at first the preconditions and success and fail criteria are evaluated. If one of them is satisfied the simulation stops and the result is returned. Otherwise the simulator requests all AIs for control commands for the autonomous cars defined in the test scenario. The communication follows a four way protocol (see Figure 4). The first message either tells the AI to stop because the simulation got interrupted or the verification module yielded a result or tells it to calculate control commands for the simulation. If the latter is the case the second message sends a list of data required by the AI e. g. position, speed, steering angle of the participant to control, camera, LiDAR or other sensor data the simulator offers. After the third message sends the actual data the AI starts to calculate commands. At this point an AI can use additional data not provided by the simulator like images of real streets. An AI may also check additional constraints which are not part of the test criteria if these need to concern information not available within the scope of the criteria definition. The calculated commands may be controlling the car the AI is associated with e. g. accelerate, break or steer or may be controlling the simulation e. g. stop it because the AI broke, is not able to calculate commands or some custom constraints are violated. In the control phase these commands are applied on the appropriate traffic participants or in case of simulation commands on the simulator. If the simulator shall continue it calculates the next few ticks (according to the verification frequency) and pauses the simulation. Then the verification process starts again.

3.3 Cloud Infrastructure

Based on the test life cycle and the verification process the structure of the cloud system follows as visualized in Figure 5. The system has a cloud layer, a client layer and a micro service

Figure 4: Communication between simulation controller and AI — Visualizes the messages sent for exchanging data between simulator and an AI.

?⟨fig:aiSimProtocol⟩?

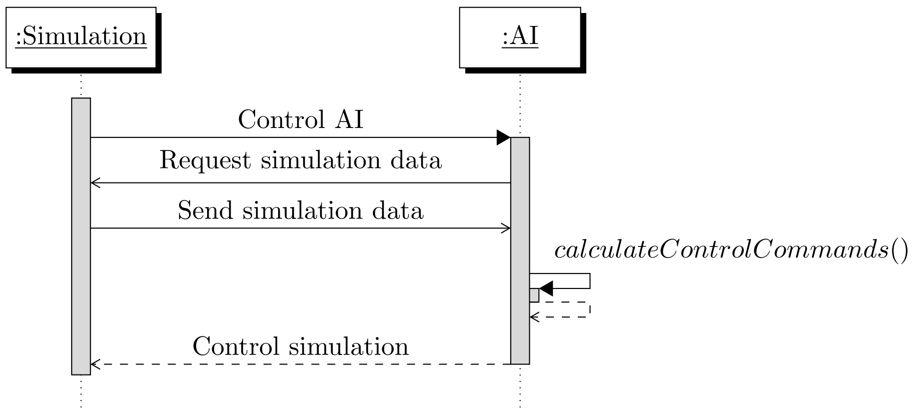
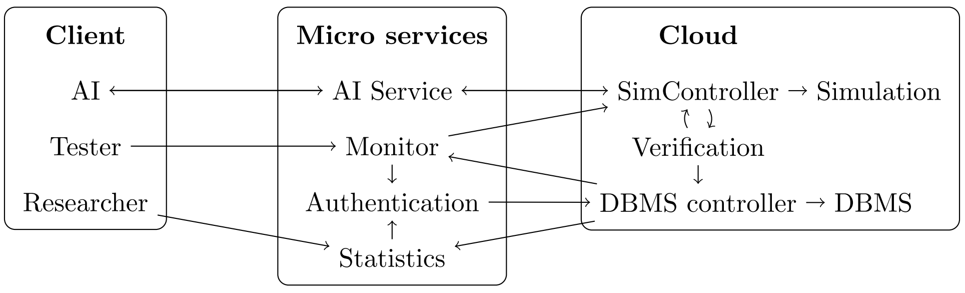


Figure 5: Data flow cloud — Shows the main modules of the cloud system and how they exchange information. The precise interaction between simulation and verification is described in subsection 3.2.

?⟨fig:dataflowCloud⟩?



layer connecting both. The cloud layer manages the simulator instances and their verification process as well as a DBMS storing all results of tests and test scenarios worth for being offered to other users. The micro service layer provides all services which have to be accessible from outside the cloud. It includes an authentication module for users to identify themselves against the system. After a tester authenticated the most elementary layer for the tester is the monitor. The monitor allows to control currently running simulations, to stop them, to start more tests and to access information about previously executed test scenarios. After a researcher authenticated the statistics service grants access to general information about the content of the database, to stored test scenarios and to results connected with these scenarios. The AI service implements the connection port for data transfer between a simulator and an AIs controlling some traffic participant within.

3.4 Proposed Tools

DCEs as well as DCTs will be based on extensible markup language (XML) because the support in many programming languages is very well and it can be easily verified using XML schema definition (XSD).
For the simulator I will use BeamNG [12] since it yields accurate and reproducible test results due to its very realistic physics and its option to run simulations in a deterministic mode. Furthermore BeamNG provides a Python application program interface (API) for generating scenarios dynamically, controlling the executions of a simulator instance and accessing much data about traffic participants e. g. their current state and lots of sensor data including damage, LiDAR and camera. Beyond that BeamNG allows to extract pixel perfect image annotation.
I will base the cloud on tools and frameworks discussed in [21] for the reason that this paper follows targets very similar to the targets of my work and already did a lot of reasoning about which tools to use. So the tools I will use are Spark [10], Alluxio [8] and OpenCL [17]. For the micro service layer I will use Flask [26] since it based on Python and therefore has interoperability with the Python API of BeamNG. Additionally Flask allows to define and use hypertext markup language (HTML) templates in a straight forward way.

4 Planned Evaluation

The evaluation investigates the scalability and usability of the system. The scalability subdivides into an analysis of the generation processes and simulation executions. The data for the quantitative evaluation is collected during usage of the cloud. *As a result it is very likely to get very diverse scenarios providing a broad for the analysis. Basic problem: The simulator needs especially good GPUs... ⇒ Is random access memory (RAM) interesting?*

4.1 Quantitative Analysis of Generation

This quantitative analysis considers diagrams where the x axis considers the number of traffic participants, the number of objects or the number of road definition points (which models the complexity and size of roads) and where the y axis describes either the resulting generation time or RAM consumption. I will determine the limiting behavior of each of the six graphs. If these turn out to be linear or better the generation is considered as scalable, if they are polynomial the generation is classified as enhancement needed and as critical otherwise.

4.2 Quantitative Analysis of Simulation

The x values in this analysis are either the number of participants or the number of AIs whereas the y values are either the measured time or the needed RAM. I will determine the limiting behavior of each of the four graphs. If they turn out to be linear or better the simulation process is considered as scalable, if they are polynomial the simulation process is classified as enhancement needed and as critical otherwise.

4.3 Qualitative Study of Usability

- *Which features used?*
- *Feature yields what is expected?*
- *Feature reliable?*
- *Encountered problems of too high work loads of the system?*
- *Some feature superfluous?*
- *Some feature missing?*

5 Schedule

The thesis is limited to a maximum time period of 6 months. Starting at the 15th of April this results in 26 weeks ending on 13th of October. These 26 weeks are divided into tasks and milestones according to Table 1.

After the 26th week there are 2 days left until the available time is fully used. These are planned to print the thesis and hand it in.

6 Success criteria

The system to develop shall satisfy all Must-Have requirements to be considered as successful.

Specification R1 Definition of an ontology for DCEs.

Specification R2 Definition of an ontology for DCTs.

Specification R3a Definition of the message formats used for the communication between simulator and AI.

Table 1: Thesis Schedule — This table also contains the planned milestones and their names

Weeks	**Task**
1	Create DCE, DCT and communication protocols
2 — 3	Implement validation, generation and criteria transformation
4	Implement data extraction
Milestone M1	“Rocket launch”
5 — 6	Implement runtime verification
7	Implement AI service
8	Implement authentication
9 — 11	Implement monitor
12	Implement data collection
13	Implement statistics service
Milestone M2	“Over the sky”
14 — 19	Evaluate system
Milestone M3	“Nosedive”
20 — 26	Write thesis
Milestone M4	“Final destiny”

?⟨table:schedule⟩?

Specification R3b Definition of the message formats used for the communication between simulator and AI.

Specification R4 Generation of scenarios out of DCE files.

Specification R5 Basic contradiction check for DCT conditions.

Specification R6 Scalable cloud infrastructure.

Specification R7 Collect data of simulations, execution and tests.

Specification R8 Access to pixel perfect image annotation.

The system should implement May-Have criteria to further increase the number of use cases and capabilities provided to users of the system.

Specification M1 Support for OpenDRIVE files as DCEs.

Specification M2 Support for CommonRoad files as DCEs.

Specification M3 Perfect size match of obstacles. In the DCE ontology static obstacles may have an arbitrary complex shape. To generate static obstacles of perfect size the generation needs to use a broader range of objects available in BeamNG and to rotate and position these in a more challenging way.

Specification M4 Use ROSBAG files for logging.

Specification M5 Elasticity support for the cloud infrastructure.

~~**Specification M6** E-Mail notification of finished tests.~~

This work will not focus and implement the following aspects.

Specification N1 Real time requirements in simulation and reaction of AIs to the scenario.

Specification N2 Support for various Linux distributions and MacOS.

Specification N3 Authentication of AIs.

An overview over all specifications is listed in Table 2.

Table 2: Summary of the Expected Thesis Features

Feature	**Must-Have**	**May-Have**	**Must-Not Have**
Specification R1	×	—	—
Specification R2	×	—	—
Specification R3a	×	—	—
Specification R3b	×	—	—
Specification R4	×	—	—
Specification R5	×	—	—
Specification R6	×	—	—
Specification R7	×	—	—
Specification R8	×	—	—
Specification M1	—	×	—
Specification M2	—	×	—
Specification M3	—	×	—
Specification M4	—	×	—
Specification M5	—	×	—
Specification M6	—	×	—
Specification N1	—	—	×
Specification N2	—	—	×
Specification N3	—	—	×

?⟨table:criteria⟩?

References

tickrate [1] Upamanyu Acharya. What is tickrate, and is it really that important?, July 2016.

commonRoad [2] M. Althoff, M. Koschi, and S. Manzinger. Commonroad: Composable benchmarks for motion planning on roads. In *2017 IEEE Intelligent Vehicles Symposium (IV)*, pages 719–726, June 2017.

?autoGen? [3] M. Althoff and S. Lutz. Automatic generation of safety-critical test scenarios for collision avoidance of road vehicles. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 1326–1333, June 2018.

opendriveToCommonRoad? [4] M. Althoff, S. Urban, and M. Koschi. Automatic conversion of road networks from opendrive to lanelets. In *2018 IEEE International Conference on Service Operations and Logistics, and Informatics (SOLI)*, pages 157–162, July 2018.

apollo [5] Baidu. Apollo.

?reconst? [6] C. Erbsmehl. Simulation of real crashes as a method for estimating the potential benefits of advanced safety technologies. *21st International Technical Conference on the Enhanced Safety of Vehicles (ESV)*, August 2015.

rosbag [7] Tim Field, Jeremy Leibs, and James Bowman. rosbag - ros wiki.

alluxio [8] Alluxio Open Foundation. Alluxio - open source memory speed virtual distributed storage.

hadoop [9] Apache Software Foundation. Apache hadoop.

spark [10] Apache Software Foundation. Apache spark - a unified analytics engine for big data.

autoware [11] The Autoware Foundation. Autoware.ai.

beamNG [12] BeamNG GmbH. Beamng.

openDrive [13] VIRES Simulationstechnologie GmbH. Opendrive.

?ontologyTom? [14] Tom Gruber. Ontology.

?ontologyStanford? [15] Tom Gruber. What is an ontology?

elasticity [16] Nikolas Herbst, Samuel Kounev, and Ralf Reussner. Elasticity in cloud computing: What it is, and what it is not. In *Elasticity in Cloud Computing: What it is, and What it is Not*, June 2013.

openCL [17] Khronos Group Inc. Opencl overview - the khronos group inc.

autowareOpen [18] S. Kato, E. Takeuchi, Y. Ishiguro, Y. Ninomiya, K. Takeda, and T. Hamada. An open approach to autonomous vehicles. *IEEE Micro*, 35(6):60–68, November 2015.

autowareOnBoard [19] Shinpei Kato, Shota Tokunaga, Yuya Maruyama, Seiya Maeda, Manato Hirabayashi, Yuki Kitsukawa, Abraham Monrroy, Tomohito Ando, Yusuke Fujii, and Takuya Azumi. Autoware on board: Enabling autonomous vehicles with embedded systems. In *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems*, ICCPS '18, pages 287–296, Piscataway, NJ, USA, 2018. IEEE Press.

kleeneLogics [20] Stephen Cole Kleene. *Introduction to Metamathematics*. Princeton : D. Van Nostrand, 1950.

unifiedCloud [21] S. Liu, J. Tang, C. Wang, Q. Wang, and J. Gaudiot. A unified cloud platform for autonomous driving. *Computer*, 50(12):42–49, December 2017.

?trackGen? [22] D. Loiacono, L. Cardamone, and P. L. Lanzi. Automatic track generation for high-end racing games using evolutionary computation. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):245–259, September 2011.

paracosm [23] Rupak Majumdar, Aman Mathur, Marcus Pirron, Laura Stegner, and Damien Zufferey. Paracosm: A language and tool for testing autonomous driving systems, February 2019.

?vmInCC? [24] R. Rauscher and R. Acharya. Virtual machine placement in predictable computing clouds. In *2014 IEEE 7th International Conference on Cloud Computing*, pages 975–976, June 2014.

microServices [25] Chris Richardson. What are microservices?

flask [26] Armin Ronacher. Flask (a python microframework).

cloudI [27] Michael Truog. Cloudi: A cloud at the lowest level.