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.

Monday 13th May, 2019

Abstract

The most common technique for testing autonomous vehicles (AVs) is simulation based testing. For each test testers need to describe a scenario, specify test criteria, setup a simulator, execute the test and collect its results. This process is tedious and error prone. I present DriveBuild, a research toolkit for simulation based testing of AVs. DriveBuild automates the process of setting up simulators, checking test criteria at simulation time, summarizing test results and distributing test runs across a cluster. DriveBuild reduces the amount of time needed to invest into preparing, running and evaluating simulation based tests.

1 Introduction and Motivation

The progress in developing autonomous vehicles (AVs) over the past years is impressive. The effort taken for testing them is amazing e. g. cars of Waymo drove over 10 million miles on public roads and over 7 billion miles within simulations [17]. However, this is according to [13] not enough for assuring a high reliability on the safety of AVs. Hence much more miles have to be driven autonomously. As [17] suggests simulations are preferred over tests on public roads. Using simulations instead of tests on public streets reduces the possibility of accidents and injuries, vastly reduces the costs of testing, allows to test cars in predefined situations and enables testers to reproduce test results and faulty behaviors. However, the setup for simulations and the preparation of test cases is tedious and error prone and concerning the definition of test cases itself there is currently no well known scheme of abstractly specifying test criteria in the context of AVs.
I present DriveBuild, a research toolkit that automates the process of setting up simulations, executing tests in parallel, distributing them over a collection of computers, verifying test criteria during simulation time and collecting test results. DriveBuild reduces the amount of effort for testing AVs, avoids dealing manually with error prone tasks and comes with an abstract scheme for formalizing test cases.
The proposal is organized as follows: section 2 provides a detailed problem statement followed by section 3 discussing current and related approaches and section 4 describing the proposed method of this work.

2 Problem Statement

The number of possible test cases is huge. So a formalization of test cases can only treat a subset of the test case space.

When focussing on test cases explicitly targeting safety critical advanced driver assistance systems (ADASs) e. g. adaptive cruise control (ACC), lane centering, emergency brake assistant or collision avoidance system the number of possible test cases is still very large. Additionally an increasing variety of supported ADASs requires an increasing variety of input data in order to enable ADASs to work properly.
Some ADASs may even need data that is not provided by the underlying simulator. This poses the problem of shipping such data with test cases and of offering additional information that can be used by an AV to determine which of the shipped data has to be considered at a certain moment during the test run.
A traffic participant occurring in a test case is either a car that follows a predefined path on the lanes or is an AV that is controlled by an artificial intelligence (AI). Since AIs differ greatly in their implementation they can not be directly included in the simulation and have to run separately. Furthermore a tester may not want to expose the implementation of an AI to DriveBuild. Hence AIs have to run externally. This yields the problem of specifying a comprehensive but efficient way for the simulator and an external AI to communicate and exchange data.

When focussing on test cases evaluating the efficiency of an AI the execution time of the verification of test criteria and the discrepancy between the hardware used for testing and the actual

hardware used within a real AV falsify the test results.
In case of an external AI the network latency further falsifies test results.

When focussing on even more test cases to be formalized the diversity of criteria required for defining success and fail criteria rises as well. This results in the problem of an increasing complexity in the validation and evaluation of test criteria.

When distributing test runs across a collection of computers a common goal is a high utilization of the provided resources. This leads to the problem of finding a strategy of distributing executions of tests based on their predicted load and therefore to determine characteristics of test cases that deposit in the load.

The goals of this work are the creation of a scheme for formalizing test cases in the context of AVs, the specification a life cycle for handling the execution of tests and the actual implementation of DriveBuild.

3 State of the Art and Technical Background

3.1 State of the Art

Concerning the definition of test environments OpenDRIVE [10] is one of most popular formats for defining very comprehensive environments and is used by 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. 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 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. 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.
OpenSCENARIO [11] is a scheme for adding traffic participants to OpenDRIVE bundled with their physical properties and specifying their dynamic behavior. The behavior is organized in maneuvers which are sequences of actions like change lane, brake, accelerate and adapt the distance to other participants. OpenSCENARIO is capable of defining conditions that trigger maneuvers as soon as they are satisfied. The variety of conditions include time to collision (TTC), time headaway, (relative) speed, traveled distance, speed, acceleration or reaching a certain position. Since OpenSCENARIO is based on the extensible markup language (XML) the dynamic behavior is fixed during runtime. Hence maneuvers can not do any computations throughout a simulation e. g. calculate steering angles or any other information not directly provided by the simulator.
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 the 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. However, CommonRoad does not guaranty that specified movements are realistic. Hence it is needed to check feasibility of movements has to be checked beforehand. The definition of roads in CommonRoad consists of two sequences of points describing the left and the right border of a lane whereas the definition of roads in the simulator I will use in my work (See subsection 4.2) describes roads by a sequence of center points and the current width of a lane. The simulator interpolates this sequence as well as the widths to generate a smooth curvature. So the simulator can not accurately visualize arbitrary lanes of CommonRoad scenarios. Concerning the definition of test criteria CommonRoad is restricted to the definition of goal regions where the car has to get to in order to pass a test.
Paracosm [19] offers test case description combined with a simulation architecture. It defines a synchronous reactive programming language whose 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 test cases automatically but which are almost random. 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 [19] 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.
J. Bach, S. Otten and E. Sax follow in [0] a model based approach for describing scenarios. In contrast to other approaches this approach focuses on creating a description scheme that is not only comprehensive but also human-readable and abstract of 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 and are connected by events triggering a transition between acts. Each act is associated with exactly one event. Thus the sequence of acts is linear. Every car in a scenario has multiple perception layers of different sizes that can be used for triggering events. 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 for testing a wide range of diverse ADASs.

S. Liu, J. Tang, C. Wang, Q. Wang and J-L. Gaudiot present in "A Unified Cloud Platform for Autonomous Driving" [18] a cloud infrastructure which is explicitly geared towards testing AVs. This infrastructure focuses on high resource utilization, high performance and low management overhead. For implementation the authors use Spark [7] for distributed computing, Alluxio [5] for allowing distributed storage and OpenCL [12] for optimizing the performance of the graphics processing unit (GPU) intensive simulations. Furthermore the paper mentions frameworks like Hadoop [6] and reasons about them why the authors used some of them or not. Since the paper deals with some of the problems of section 2 it is interesting for my work concerning which tools and frameworks to use.
CloudI [24] 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 uses heavy weighted services and the granularity necessary for controlling simulations and exchanging information with AIs may not be good enough.

OpenDS [0] is a java based cross platform simulator specialized for simulating AVs. It utilizes the capabilities of the physics library JBullet [0] and therefore provides a very realistic behavior of participants in the simulation. It uses the JMonkeyGameEngine [0] framework for generating the graphical representation of the simulation and the lightweight java game library (LWJGL) for utilizing the GPU and is capable of generating scenarios based on OpenDRIVE descriptions. However, OpenDS lacks physical properties and calculations including detailed damage and behavior of the bodywork and therefore a detailed driving behavior.
The open source project Apollo [3] 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 allows to control an AV. However Apollo does not provide test case criteria which consider complete scenarios or multiple traffic participants at once.
Autoware [8, 14, 15] is another comprehensive open source project. It builds a complete ecosystem containing algorithms for localization, perception, detection, prediction and planning, containing predefined maps and containing the capability to handle real sensors and vehicles. It also provides the LGSVL simulator which can visualize information like perception data or status of other participants. Autoware works with ROSBAG files [4] 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.

3.2 Technical Background

Synchronous simulation is an execution strategy a simulator can follow. A simulator working with this strategy calculates the progress AVs make in a certain time interval and pauses. When paused the simulation can call AIs and request them for commands controlling AVs or the simulation. After the simulator applied these commands it resumes if the simulation is not aborted and the process starts over.

Both simulations and test cases in my work are based on logical time namely **ticks** [1]. A tick is a time interval of predefined length in which the simulator calculates the changes to the environment plus to all traffic participants and applies them to the simulation. A tick is the smallest logical time unit.

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

Group	To be supported	Description
1	Collision avoidance system Forward Collision Warning Emergency Brake Assist Intersection assistant Turning assistant	Systems that alert a driver with a warning, avoid or reduce the severity of a collision and actively brake an AV
2	Cruise control Intelligent speed adaptation Adaptive cruise control Active Brake Assist	Systems that control the speed of an AV making sure it does not exceed a certain speed limit or maintains 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 the road and that make sure it does not leave the lane it is supposed to drive on
4	Adaptive light control	A system that controls the lights depending on the brightness and whether another participant comes towards

4 Proposed Method

4.1 Test Case Formalization

There is no standard specifying reference test cases and their expected results [20]. Since ADASs are safety critical the formalization in this work concentrates on ADASs that can be tested using simulations. Table 1 lists the target ADASs and groups them by their functionality and the data these need to work. The data ADASs require is heavily dependant on their implementation Nevertheless based on their functionality there is a common basis of information required to test them (See Table 2).
The formalization divides into the definition of environments, participants and criteria. It follows

Table 2: Minimum test data requirement — This table lists the minimum required data for the groups of ADASs described in Table 1 to test them.

Type of data	Group 1	Group 2	Group 3	Group 4
Distance to other traffic participants	✓	✓	✗	✓
Distance to lane markings and to the edges of the roads	✗	✓	✓	✗
Relative speed to other traffic participants	✓	✓	✓	✓
Damage detection	✓	✓	✗	✗
Speed	✓	✓	✗	✗

a modular approach separating descriptions of environments on the one hand from participants and criteria on the other hand to allow reuse of environments throughout multiple scenarios and to avoid duplication.

The definition of test environments follows a custom scheme since other currently well known schemes are not suitable as explained in section 2. The scheme describes lanes and obstacles. A lane is a sequence of tuples. Each tuple contains the lane center point and the current width of the lane. Since this representation is identical to the representation used by the simulator which I will use in my work and which is described in the proposed tools paragraph in subsection 4.2 the expected outcome of the generated lanes is very similar to the test case definition. Additionally this representation allows to easily calculate the distance of an AV to the lane center and to determine the direction of a lane. An obstacle may be either a cube (having a position, width, length, height and rotation) or a cylinder/cone (having a position, radius, height and rotation). Traffic participants are specified by an initial state, their movement and optionally an AI access point (AP). The shape and the physics of participants are specified by using a predefined model provided by the underlying simulator. An initial state sets the initial position, the initial orientation and the initial movement mode of an AV. A movement is a sequence of states. Each state defines a target waypoint, optionally a speed value and the movement mode of the AI starting from that point. A waypoint is a position bundled with a tolerance value which allows a participant to not precisely reach a position but to pass by in a certain distance. A speed value is either a target speed the AV should have or a speed limit the AV shall not exceed. The movement mode may be either `manual`, `autonomous` or `training`. If the movement mode of an AV is `manual` the car heads straight to the next waypoint. If the movement mode is `autonomous` the AI of the associated AP is frequently requested like explained in subsection 4.2 and provided with data it needs to control the AV. If the movement mode is set to `training` the AV acts the same way it does in `manual` but requests the AI like in `autonomous`. In contrast to `autonomous` the AI can not control the AV. This mode is meant to be used for collecting training data for AIs. This scheme for movements enables to mix sections where an AV is forced to follow a path, where an AI has to control it or where data has to be collected. If the movement of an AV has sections that have either the movement mode `autonomous` or `training` an AP has to be defined. This AP contains a network address that has to be used for requesting the AI and a frequency (in ticks) specifying how often the AI has to be requested during the simulation to control the associated AV.

The test criteria definition specifies preconditions, success and fail criteria. Commonly a test is considered as successful if it ends without triggering a fail criterion. In the context of testing AVs a possible test might be "An AV A is successful if it reaches a certain position P and fails if it takes any damage". Such test cases require to separate success from fail criteria since if A just does not move it does not fail but it is not successful either. The most basic way of describing this is the Kleene and Priest logics [16] which is a three-valued logics declaring besides `true` and `false` also `unknown`. This third value `unknown` allows to express that a criterion could not be determined or is currently not considered. The test criteria divide into connectives (`and`, `or` and `not`), state conditions (SCs) and validation constraints (VCs) and can be nested as shown in Figure 1.. SCs as well as VCs evaluate the current state of the simulation or some AV and determine whether it fulfills a certain condition. SCs always yield either `true` or `false`. VCs restrict whether the nested criterion has to be considered during the verification process described in subsection 4.2. 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. Considering a fail criterion like "While AV A drives on lane L it must not exceed a speed limit of S" the speed of A should only be evaluated as long as A drives on L and return either `true` or `false`. If A is not on L `unknown`

Figure 1: Nesting of criteria — This diagram shows the allowed nesting structure for test criteria. Italic types are abstract.

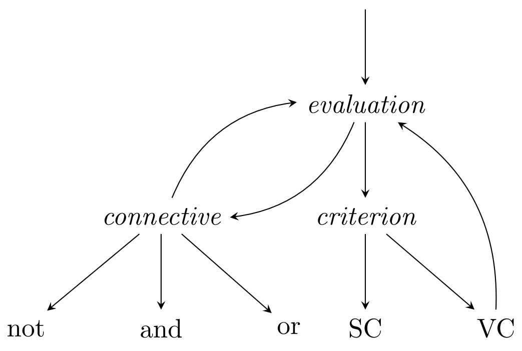
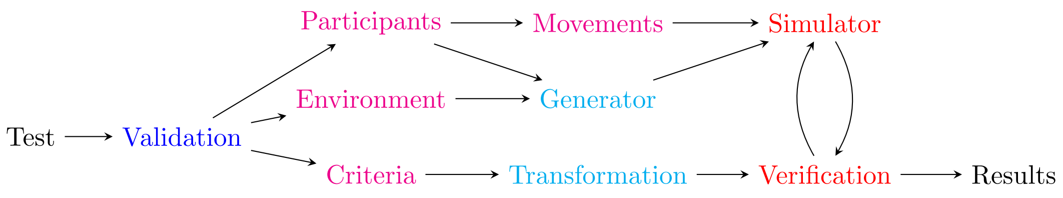


Figure 2: Test life cycle — Visualizes the four main steps the processing of a test case follows. The validation step is blue, the extraction step is magenta, the transformation step is cyan and the execution step is red.



shall be returned. The supported types of criteria are based on the formalization of tests and the data ADASs need as shown in Table 2. The list of supported criteria is in Table 3. Additional criteria can be introduced as explained in subsection 4.3.

Proposed tools: The whole formalization will be based on XML since it has great support in many languages and can be validated based on XML schema definitions (XSDs) making sure a test case is specified properly before running it.

4.2 Test Life Cycle

The test life cycle divides into validation, extraction, transformation and execution as shown in Figure 2.
The validation checks the test case whether it is broken or malformed based on the appropriate XSDs. If the test case is valid the environment, the criteria and the participants are extracted. The information about lanes and obstacles in the environment is passed to the generator creating representations compatible with the simulator. The initial states of all participants are also passed to the generator which creates for each initial state a traffic participant and adds it to the simulation. The movements of the participants are passed to the simulator that applies these sequentially after the simulation started. The defined criteria are transformed to a Kleene and Priest logics expression that can be evaluated by the verification process during the simulation.

Table 3: Types of criteria — This table lists all supported types of test criteria, describes their purpose and lists whether these can be used as 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 TTC of an AV and another participant or obstacle is smaller than a given value	✓	✗
light	Checks whether an AV activated certain lights e. g. high beam and passing light	✓	✓
waypoint	Checks whether an AV has passed a certain waypoint	✓	✓

Then the simulator starts the test and the interaction with the verification process. This interaction is described more detailed in subsection 4.3. As soon as the verification process determined whether the test succeeded or failed it stops the simulation and returns the test results.

Proposed tools: I will use BeamNG [9] as simulator since it provides very accurate physics and therefore accurate test results plus it comes with a Python interface that allows to control simulator instances, to create scenarios dynamically and to retrieve sensor data from participants. Furthermore BeamNG comes with the ability of pixel perfect annotation which some ADASs need. The validation, the extraction, the generator, the transformation and the verification will be done in Python since the interface of BeamNG is written in Python too and thus the interaction is easy and the interoperability is high.

4.3 Runtime Verification

The runtime verification implements the interaction between a simulator and the verification of test criteria deciding whether a test succeeded or failed. It follows the synchronous simulation strategy to make sure that the point in time where AIs have to control AVs is not influenced by network latency or the current load of the underlying hardware. The main execution loop realizing synchronous simulation is shown in Figure 3.
The runtime verification starts with evaluating the preconditions, success and fail criteria and determines the verification result based on the state machine shown in Figure 4. If the verification ends in one of the final states `skipped`, `failed` or `succeeded` the simulation stops and returns the final state. In this case all AIs defined in the test case are told to stop too. Otherwise the verification yields `undetermined` and the runtime verification continues with the main execution loop. If the runtime verification continues it searches for all AVs that are according to their movement currently controlled by an AI and that have to be requested according to their

Figure 3: Runtime verification — Depicts the main execution loop of the interaction between a simulator and a verification process (See Figure 2).

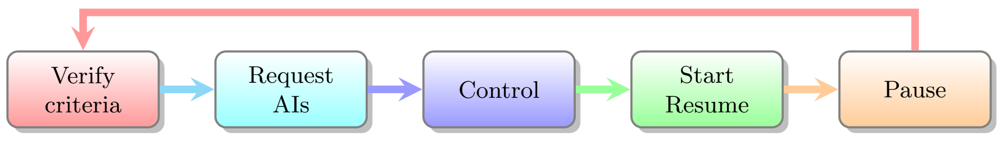
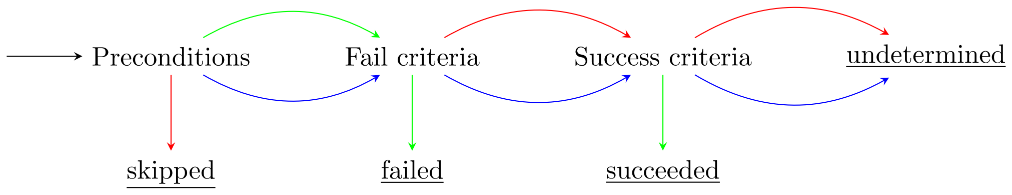


Figure 4: Verification state machine — Shows the state machine for determining the current state of the test case execution based on the evaluation of preconditions, success and fail criteria. Underlined nodes are final states and arrows describe the transition from node to node depending on whether a criterion evaluated to true, false or unknown.



frequency and the current time of the simulation (in ticks). Using the network address of the AP of these AIs the runtime verification requests them for commands controlling the simulator or the appropriate AV. Figure 5 depicts the four-way protocol which is used for the communication. The first message sends either `finished`, `interrupted` or `requested`. This allows an AI to recognize whether the simulation still runs or stopped. If the simulation still runs the AI requests properties of the current state of the associated AV and its available sensor data needed to control the AV. The third message transfers the appropriate data in a serialized form. After receiving the data the AI starts to calculate control commands for either controlling the simulator (e. g. interrupt it or skip the next waypoint) or for controlling the associated AV (e. g. accelerate, brake or steer). The opportunity to send commands controlling the simulator instead of the AV allows an AI to evaluate additional constraints that extend the specified test criteria or to skip a waypoint if it can not be reached without braking and turning the AV around. In such a case an AI can send a stop command to the simulator and abort the test case execution. The control step applies the commands the runtime verification received appropriately to the simulator or the correct AV. If the simulator does not get a command to stop it starts the test case execution respectively resumes it if it was paused previously. The simulator calculates the next tick, pauses the simulation and the runtime verification starts over again.

Proposed tools: The exchange of messages and data will be based on Thrift [23] since it allows to define messages in a programming language neutral way, it is able to cross compile the definition into many other languages including Python and it supports an exception mechanism.

4.4 Cluster Architecture

The architecture uses a client server model and is depicted in Figure 6. The functionality of

Figure 5: Communication between simulation controller and AI — Visualizes the messages sent for exchanging data between a simulator and an AI.

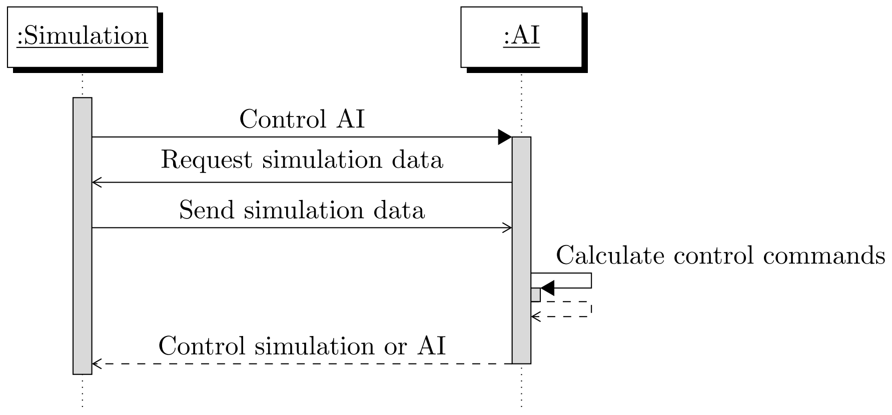
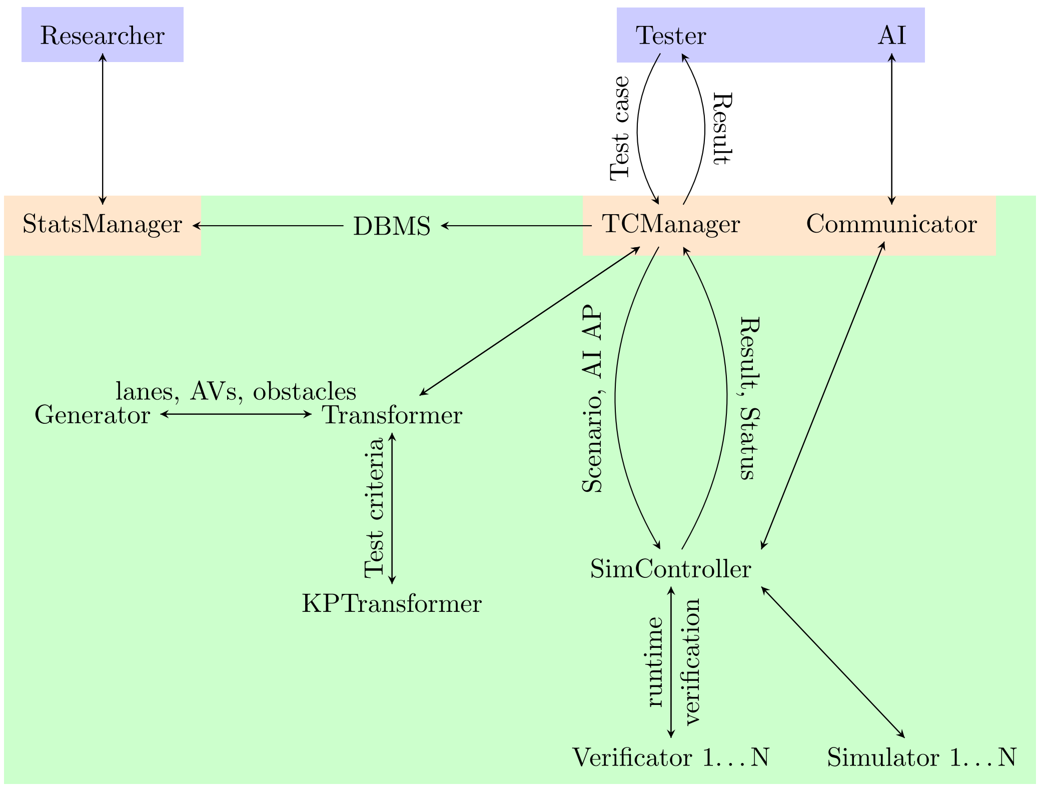


Figure 6: Cluster architecture — Visualizes all components of the cluster and the data flow between them. The components in the blue area belong to the client side and the components in the green area belong to the cluster. The orange area contains cluster components that build the micro service based accessible interface.



the cluster is provided through micro services [21]. The use of micro services allows hiding and strictly separating functionality plus it allows more granularity than other architectures like SOA. The cluster offers multiple services.
The TCManager service implements methods for accepting test cases of a tester, checking their validity, passing them to the transformer, triggering the execution of test cases, monitoring the execution, returning test case results to the tester and to store results in a DBMS. The Transformer (transformation step in Figure 2) accepts validated test cases, extracts information about the environment and the participants, generates representations which are compatible with the simulator, extracts the test criteria and transforms them to an expression that can be evaluated during the simulation. The SimController manages all simulator instances as well as their verification instances and implements the runtime verification described in subsection 4.3. It also provides methods for requesting the current status of test executions which the TCManager requires for monitoring. The Communicator service handles the exchange of messages between a simulator controlled by the SimController and an AI controlling an AV of a scenario and thus uses the protocol visualized in Figure 5. The StatsManager service grants access to the data stored by the TCManager in the DBMS enabling researchers to investigate and analyze collected data about test case executions.

Proposed tools: The cluster will be based on simple linux utility for resource management (SLURM) [25] since the only cluster I have access to is based on that. I will use PostgreSQL since many languages provide good support, it is well known and thus bullet proven. The micro services will be based on Flask [22] since it is written in Python providing high compatibility with other components and it allows to use hypertext markup language (HTML) templates.

5 Planned Evaluation

The evaluation divides into a quantitative and a qualitative part. The dataset will be collected during the seminar "Search-based Software Engineering for Testing Autonomous Cars" at the university of Passau which takes place in the summer term 2019.

5.1 Quantitative Analysis of Runtime Verification

The quantitative analysis evaluates the runtime complexity of the test criteria verification based on the complexity of the lanes and the complexity of test criteria definitions.
The complexity of the lane network is defined by the number of waypoints N_W and the average tolerances T_W. The higher N_W and the lower T_W the more complex is the lane network.
The complexity of the test criteria definition is defined by the number of test criteria N_C and the average depth D_C where N_C is the number of SCs plus the number of VCs. The higher N_C and the higher D_C the more complex is a test criteria definition.
The use of DriveBuild during the seminar allows to collect much data about test cases and their execution. Each executed test case is stored in the DBMS along with their results, statistics about them and the execution times of every call to the test criteria verification. For the evaluation I will create tuples (N_W, T_W, N_C, D_C, T) for every execution time T. Based on these I will create four graphs grouping the tuples by N_W, T_W, N_C or D_C and averaging over T. All graphs yield a runtime complexity.
In the end, the runtime complexities determine the maximum complexity of test criteria definitions that is feasible and which of N_W, T_W, N_C and D_C is the most crucial. The runtime complexity is interpreted as listed in Table 4.

Table 4: Interpretation of the runtime complexity of the test criteria verification

Runtime Complexity	**Interpretation**
$< O(n)$	Highly complex test criteria definitions are feasible.
$O(n)$	Reasonable complex test criteria definitions are feasible.
$O(x^n)$	Test criteria definitions are not feasible for reasonable high complexity.
$\geq O(n^x)$	The feasible complexity of test criteria definitions is clearly restricted.

5.2 Qualitative Study of Comprehensiveness

The qualitative study investigates the comprehensiveness of DriveBuild in terms of the capabilities of the formalization and the application areas for which DriveBuild can be used. I will ask participants of the seminar (about 11 people) which kinds of test cases they realize, how they use DriveBuild for automation, benchmarks and training AIs and which problems they had to face or were not able to solve. When one of the participants encounters a problem I will investigate whether the current system is able to solve it. If this is not the case I will explore how DriveBuild would have to be extended to solve the problem and decide whether I add it. I will document all of these steps.
In the end, this documentation shows the capabilities of DriveBuild. I will be able to show problems of the original system, whether these problems could be solved and how they where solved if added.

6 Schedule

The thesis is limited to a maximum time period of 6 months. Starting at the 13th of May this results in 26 weeks ending on 10th of November. Since DriveBuild is used during the seminar the schedule has to consider the progress of the seminar and requires to result in a working implementation until half of the semester. The schedule in Table 5 divides into three milestones. The first milestone contains the collection of requirements and the actual implementation of DriveBuild. In the second milestone I will collect data about the executed tests, conduct my qualitative study and evaluate my work. In the third milestone I will write the actual thesis.

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.

7 Success criteria

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

Table 5: Thesis Schedule — Contains all main tasks of my work, how long they presumably take and how these are grouped into milestones.

Weeks	**Task**
1	Specify formalization schemes and AI communication protocol
1	Implement Generator, KPTransformer and Transformer
2	Implement SimController with runtime verification
3	Implement Communicator and TCManager
Milestone M1	“Ready to go”
4	Implement collection of data and StatsManager
5 — 12	Conduct and document qualitative study, collect data for quantitative analysis and provide support and bugfixes
End of semester	
13 — 16	Analyze qualitative study
17	Conduct quantitative analysis
Milestone M2	“Final countdown”
18 — 24	Refine and finalize thesis
25 — 26	Contingency time of 2 weeks
Milestone M3	“Final destiny”

Requirement R1 The system shall be able to convert a given formalized test scenario into a BeamNG scenario. The simulation shall contain roads, traffic participants and obstacles and shall simulate the movement specified in the test case.

Requirement R2 The system shall be able to simulate multiple generated BeamNG scenarios simultaneously.

Requirement R3 The system shall be able to continuously evaluate test criteria during simulation time.

Requirement R4 The system shall collect executed test cases and their execution results.

Requirement R5 The system shall measure the execution time of test criteria verifications.

Requirement R6 The system shall provide a service granting access to the collected data.

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

Requirement M1 The system should provide predefined test suites for training AIs.

This work will neither focus nor implement the following aspects.

Requirement N1 The system will not support test cases that require real time capabilities.

Requirement N2 The system will not support various Linux distributions or MacOS.

Requirement N3 There will not be the opportunity to simulate participants with a custom shape and physics that are not provided by the simulator.

Requirement N4 The system will not be able to create log files for replaying and debugging simulations e. g. ROSBAG files.

8 References

[0] .

[1] Upamanyu Acharya. *What is Tickrate, and is it Really That Important?* July 2016. URL: https://fynestuff.com/tickrate/.

[2] M. Althoff, M. Koschi, and S. Manzinger. "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/.

[0] German Research Center for Artificial Intelligence. *OpenDS*. URL: https://opends.dfki.de/.

[0] 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.

[3] Baidu. *Apollo*. URL: http://apollo.auto/.

[4] Tim Field, Jeremy Leibs, and James Bowman. *rosbag - ROS wiki*. URL: http://wiki.ros.org/rosbag.

[5] Alluxio Open Foundation. *Alluxio - Open Source Memory Speed Virtual Distributed Storage*. URL: http://www.alluxio.org/.

[6] Apache Software Foundation. *Apache Hadoop*. URL: https://hadoop.apache.org/.

[7] Apache Software Foundation. *Apache Spark - A Unified Analytics Engine for Big Data*. URL: https://spark.apache.org/.

[8] The Autoware Foundation. *Autoware.AI*. URL: https://www.autoware.ai/.

[9] BeamNG GmbH. *BeamNG*. URL: https://beamng.gmbh/research/.

[10] VIRES Simulationstechnologie GmbH. *OpenDRIVE*. URL: http://www.opendrive.org/index.html.

[11] VIRES Simulationstechnologie GmbH. *OpenSCENARIO*. URL: http://www.openscenario.org/.

[12] Khronos Group Inc. *OpenCL Overview - The Khronos Group Inc*. URL: https://www.khronos.org/opencl/.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[18] S. Liu et al. "A Unified Cloud Platform for Autonomous Driving". In: *Computer* 50.12 (Dec. 2017), pp. 42–49. ISSN: 0018-9162. DOI: 10.1109/MC.2017.4451224. URL: https://ieeexplore.ieee.org/document/8220475/.

[19] Rupak Majumdar et al. *Paracosm: A Language and Tool for Testing Autonomous Driving Systems.* Feb. 2019. URL: https://arxiv.org/pdf/1902.01084.pdf.

[20] 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.

[21] Chris Richardson. *What are microservices?* URL: https://microservices.io/.

[22] Armin Ronacher. *Flask (A Python Microframework).* URL: http://flask.pocoo.org/.

[23] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. "Thrift: Scalable cross-language services implementation". In: (Apr. 2007).

[0] jMonkeyEngine Team. *jMonkeyEngine.* URL: http://jmonkeyengine.org/.

[24] Michael Truog. *CloudI: A Cloud at the lowest level.* URL: https://cloudi.org/.

[25] 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.