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 18th April, 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 multiple test runs across a *cloud/cluster*. DriveBuild reduces the amount of time needed to invest into preparing and evaluating simulation based tests.

1 Introduction and Motivation

The progress in developing 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

When distributing test runs across a collection of computers a common goal is 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.

A traffic participant occurring in a test case is either a car that follows a predefined path on the roads 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.

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 ~~to test~~ 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 make ADASs work properly.
But the available data is heavily dependant on the data the underlying simulator provides. Some ADASs may even need data that is not provided by the underlying simulator. This poses the problem of shipping the data with test case definitions and of offering a characteristic based on which an AI can determine what shipped data has to be considered at a certain moment.

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 falsifies test results even more.

When the variety of test cases which can be formalized grows 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.

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

~~There exist some papers and projects that already tackle some aspects of section~~ 2.

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. 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 given in this format is too expensive for being used within a *cloud/cluster* 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.
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. Since states tight time, position, orientation 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 to in order to pass a test. Furthermore the definition of roads is absolutely incompatible with the scheme the simulator that is going to be used in my work (See subsection 4.2) uses.
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.

The authors of [18] present 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 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 [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

Both simulations and test cases in my work are based on logical time namely **ticks** [1]. A ~~video showing~~ a simulation ~~during simulation time~~ 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 calculation results in an image referred to as frame that is part of the video and defines a state at a tick during the simulation time.

Synchronous simulation is an execution strategy a simulator can follow. 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.

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.
The formalization divides into the definition of environments, participants and criteria. It follows 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. Lanes are represented using a sequence of tuples. Each tuple contains the lane center point and the current width of the lane. This representation is identical to the scheme used by the simulator I will use in this work which is described in the following proposed tools paragraph. The obstacles are defined by boxes having a width, a length, a height and a position.
Traffic participants are specified by an initial state, their movement and optionally an AI access point (AP). An initial state sets the initial position, the initial orientation of an AV and whether an AI has to control the AV. A movement is a sequence of states. Each of these states defines a target waypoint, optionally a speed value and whether the AV shall be controlled by an AI starting from that point. A waypoint is a position bundled with a tolerance value which allows a

Table 1: Target ADASs — This table lists all ADASs that the formalization aims to support

To be supported
Collision avoidance system
Cruise control
Intelligent speed adaptation
Adaptive cruise control
Intersection assistant
Lane centering
Lane departure warning system
Lane change assistance
Turning assistant
Wrong-way driving warning
Emergency Brake Assist
Active Brake Assist
Adaptive light control
Forward Collision Warning

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. If an AV is controlled by an AI the AI is frequently requested (See subsection 4.2) and provided with data it needs to control an AV. Otherwise the AV just follows the given positions. The scheme for movements enables to mix sections where an AV is forced to follow a path and where an AI has to control it. If the movement of an AV has sections that have to be controlled by an AI an AI AP has to be defined. This AP contains a network address which 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. The separation of success and fail criteria is needed since from not triggering a failure criterion it does not follow the test is successful e. g. "An AV A is successful if it reaches a certain position P and fails if it takes any damage". If A just does not move it does not fail but it is not successful either. The test criteria themselves are based on Kleene and Priest logics [16] which introduces three-valued logics having `true`, `false` and `unknown` allowing to not only express whether a condition is satisfied but also whether it could be determined and is currently 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` shall be returned. The supported types of criteria are listed in Table 2.

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.

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

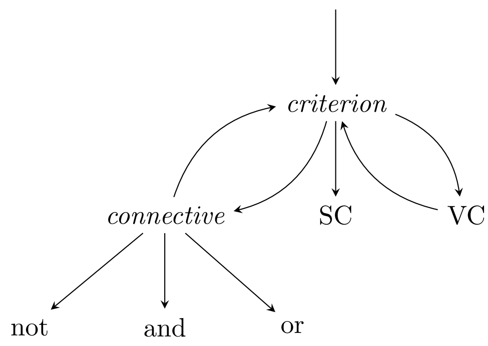
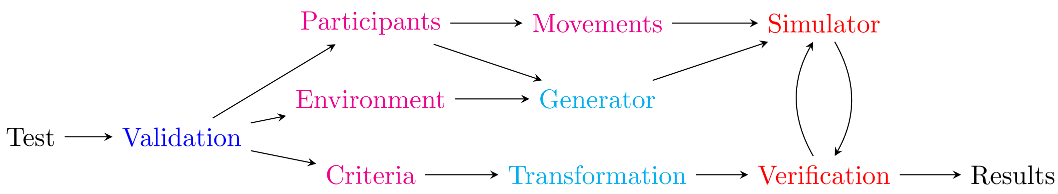


Table 2: Types of criteria — This table lists all supported types of test criteria, describes their purpose and lists whether these can be used as VCs or SCs.

Type	**Description**	**VC**	**SC**
position	Checks whether an AV is at a certain position or within a certain area	✓	✓
lane	Checks whether an AV drives on a certain lane or off-road	✓	✓
speed	Checks whether an AV exceeds or falls below a given velocity	✓	✓
damage	Checks the damage of AVs	✓	✓
time	Checks whether the simulation is currently within a certain interval of ticks	✓	×
distance	Checks the distance between two AVs or between an AV and the center of the lane driving on	✓	✓
TTC	Checks the TTC of an AV and another participant or obstacle	✓	×
output	Checks the output of an AI for patterns	✓	✓
light	Checks whether an AV activated certain lights e. g. high beam and passing light	✓	✓

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.



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 roads 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. 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

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

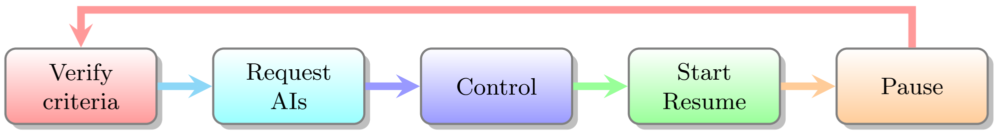
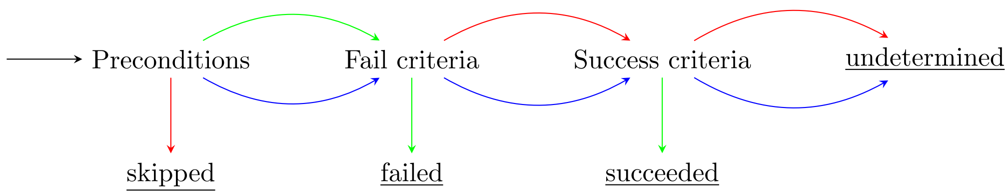


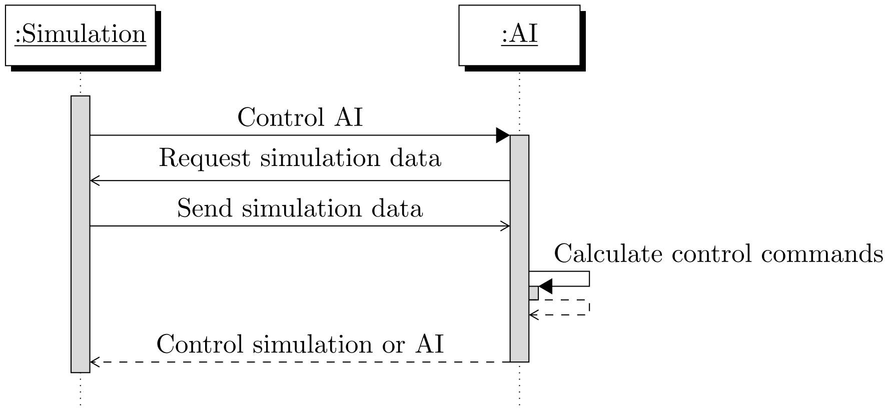
Figure 4: Verification state machine — Shows the state machine for determining the current state of the test case execution based on the evaluation of preconditions, success and fail criteria. Underlined nodes are final states and arrows describe the transition from node to node depending on whether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



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 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 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. 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.

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



4.4 *Cloud/Cluster* Architecture

The architecture uses a client server model and is depicted in Figure 6. The functionality of the *cloud/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 *cloud/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 cloud/cluster will be based on simple linux utility for resource management (*SLURM*) [25]. Why?* I will use PostgreSQL since it 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

Figure 6: *Cloud/Cluster* architecture — Visualizes all components of the *cloud/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 *cloud/cluster*. The orange area contains *cloud/cluster* components that build the micro service based accessible interface.

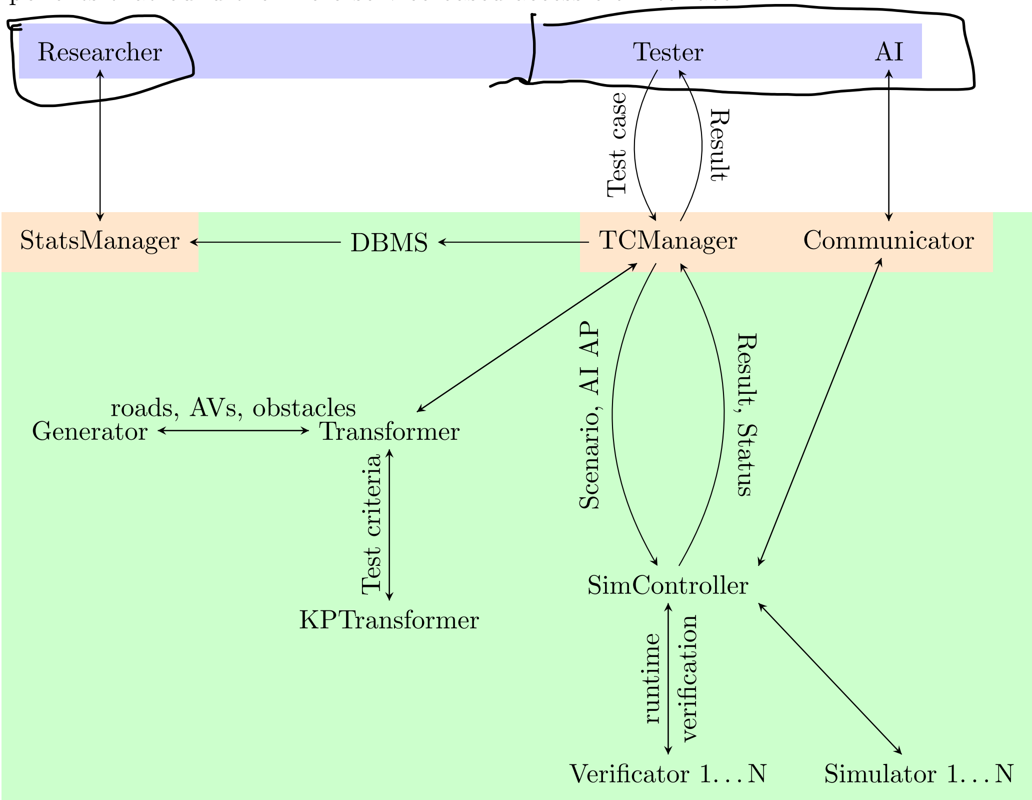


Table 3: Interpretation of the runtime complexity of the test criteria verification

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.

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 test criteria definitions. The complexity is defined by the number of test criteria and the average depth where the number of test criteria is the number of SCs plus the number of VCs. The higher the number of test criteria and the higher the average depth 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 the execution times of every call to the test criteria verification. For the evaluation I will create tuples (N, D, T) for every execution time T where N is the number of criteria and D is the average depth of the associated test case. Based on these I will create two graphs: one grouping the tuples according to N and averaging over T and one grouping the tuples according to D and averaging over T. Both graphs yield a runtime complexity.
In the end, the runtime complexities determine the maximum complexity of test criteria definitions that is feasible and whether either the number of test criteria or the average depth is more crucial. The runtime complexity is interpreted as listed in Table 3.

5.2 Qualitative Study of Formalization

The qualitative study investigates the comprehensiveness of the formalization of test cases. I will ask participants of the seminar (about 8 to 12 people) whether there are test cases they could not formalize and why they could not formalize them. Then I will take a look if they can not be formalized based on the current scheme after all. If this is not the case I will explore what would have been needed to formalize the test case and whether I add it. I will document all of these steps.
In the end, this documentation shows the capabilities of the test case formalization. I will be able to show problems of the original scheme, whether these problems could be solved and how there where solved.

6 Schedule

The thesis is limited to a maximum time period of 6 months. Starting at the 22th of April this results in 26 weeks ending on 20th of October. Since DriveBuild is used during the seminar a working implementation has to be ready by half the semester. Hence these 26 weeks are divided into tasks and milestones according to Table 4.

Table 4: Thesis Schedule — This table also contains the planned milestones and their names

Weeks	Task
1	Specify formalization schemes and AI communication protocol
2	Implement Generator, KPTransformer and Transformer
3 — 4	Implement SimController with runtime verification
5	Implement Communicator and TCManager
Milestone M1	“Ready to go”
6	Implement collection of data and StatsManager
7 — 14	Conduct and document qualitative study, collect data for quantitative analysis and provide support and bugfixes
End of semester	
15 — 18	Analyze qualitative study
19	Conduct quantitative analysis
Milestone M2	“Final countdown”
20 — 24	Refine and finalize thesis
25 — 26	Contingency time of 2 weeks
Milestone M3	“Final destiny”

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.

Requirement R1 The system shall be able to convert a 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 be able to create log files for replaying and debugging simulations e. g. ROSBAG files.

Requirement M2 The system should provide live update monitoring for currently running tests.

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.

An overview over all requirements is listed in Table 5.

Table 5: Summary of the expected thesis features

Feature	**Must-Have**	**May-Have**	**Must-Not Have**
Requirement R1	×	—	—
Requirement R2	×	—	—
Requirement R3	×	—	—
Requirement R4	×	—	—
Requirement R5	×	—	—
Requirement R6	×	—	—
Requirement M1	—	×	—
Requirement M2	—	×	—
Requirement N1	—	—	×
Requirement N2	—	—	×

8 References

[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/.

[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).

[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.