Towards a distributed real-time hybrid simulator for autonomous vehicles

Noname manuscript No.
(will be inserted by the editor)

Towards a Distributed Real-Time Hybrid Simulator for Autonomous Vehicles

Jens de Hoog · Arthur Janssens · Siegfried Mercelis · Peter Hellinckx

Received: date / Accepted: date

Abstract To thoroughly test and validate algorithms and systems of autonomous vehicles, a large number of vehicles, many tests and a multitude of datasets are needed. This way of developing and testing is difficult, expensive and sometimes even dangerous. To combine the benefits of real world testing with the scalability and lower cost of simulation based testing, we present a novel methodology for a real-time hybrid simulator that is capable of handling real and simulated vehicles simultaneously with full interaction, in real time. We validated our methodology by assessing its overall performance and real-time capabilities using an F1/10 scale vehicle. The results effectively show the viability of this approach for validation of autonomous vehicles in a cost-efficient and safe manner.

Keywords Distributed · Real-time · Hybrid · Simulator · Autonomous · Vehicles

1 Introduction

These days, self-driving cars are a hot topic. The autonomy of such cars can be divided in 6 levels, starting from level zero without any automation to level five where the car has full control in all circumstances [31][30]. Major companies, such as Uber, Daimler, Waymo and Tesla, Inc., are pushing the limits of these fully autonomous vehicles. The technology has even evolved so far that the car of Waymo with autonomy level 4 has approximately one

Jens de Hoog (✉) · Arthur Janssens · Siegfried Mercelis · Peter Hellinckx
E-mail: jens.dehoog@uantwerpen.be
Tel.: +32 3 265 89 42

University of Antwerp - imec
IDLab - Faculty of Applied Engineering
Groenenborgerlaan 171, 2020 Antwerp, Belgium

disengagement (an action where the driver has to take over control) in every 5,000 driven miles [22]. Such vehicles will also be very practical in the evolution towards *Mobility as a Service* (*MaaS*). When using such a service, the user pays for the ride he is taking instead of paying for the car he or she had bought. The combination of *MaaS* and self-driving cars provides many benefits such as reducing traffic congestions and overall transportation costs [34]. The development of smart highways is an interesting topic as well. In here, the behaviour of autonomous vehicles can be optimised by studying the interaction between vehicles, vehicles and infrastructure, and between vehicles and the environment, enabling advanced applications such as traffic management and platooning. [2].

In order to thoroughly test these systems and algorithms, multiple cars are needed, along with a multitude of data and tests. However, due to unforeseen glitches or imperfections, crashes could occur during the development phases; this should be avoided at all times. Managing a large number of those cars is also both difficult and financially expensive. Therefore, a simulation-based test bed is needed in which simulated cars can interact with each other for training and testing purposes. Still, these simulators are not ideal; Gechter et al. [11] state that these are either too simplified or specialised in one particular element. For example, the tool "*Autonomie*" [1] analyses performance and energy consumption in a vehicle, while "*Simcenter Amesim*" focuses on the modelling of mechatronic systems. Additionally, the deployment of such simulators comes at a computational cost when modelling multiple vehicles in real-time with scalability in mind. Therefore, a hybrid simulator is desired; it provides the interaction between real and simulated vehicles in the same three-dimensional environment while taking the characteristics of a real car into account.

To investigate the fundamental principles of real-time hybrid vehicle simulation, such as hybrid interaction, synchronization and sensor modification, we design a simulation environment for the *F1/10*-car [29]. This vehicle is developed by the university of Pennsylvania and is a scaled model of an F1-car with scaling factor 1/10. The purpose of this project is to start a competition with self-driving *F1/10*-cars in which many different teams can strive for the best and fastest autonomous racing algorithm. This involves designing, building and testing of these vehicles in order to develop a robust solution.

Despite the fact that simulators which take the reality into account already exist, the ultimate goal of our environment is to support different types of autonomous vehicles (i.e. cars, drones, ...). This will require interaction between real and simulated vehicles, even though they are completely independent for each other. Additional complexity lies in the heterogeneity of the design, communication and operation when considering multiple vehicles. Even the F1/10-cars are heterogeneous; they do not necessarily use the same hardware or operating system. Since no simulator has been designed for this use case yet, this paper introduces a new hybrid methodology that implements the proposed requirements.

We have organised the rest of the paper in the following way. Section 2 further discusses the current State of the Art, along with the positioning of our methodology. Section 3 elaborates on the concept of our approach, while Section 4 defines a test setup for the experiments in Section 5. Finally, a conclusion and suggested topics for further research are formed in Section 6.

2 Related work

As previously mentioned in the introduction, different simulators that provide interaction with a real environment already exist. Examples are *Hardware-In-the-Loop (HIL)* and *Software-In-the-Loop (SIL)*. Following are two State-of-the-Art simulators that both provide interaction to a real environment.

Gechter et al. raises the difficulty of simulating autonomous vehicles with different sorts of simulators [11]. Each simulator is either quite generic or optimised for one specific purpose (e.g. mechanical links between components). Therefore, the authors propose VIVUS (Virtual Intelligent Vehicle Urban Simulator), which is an architecture for hybrid simulation. In this way, different sensors, actuators and entities can be tested in the same environment. The system contains at least one autonomous vehicle; this is either a real or simulated one. The simulator itself is based on Unity3D. In order to achieve hybrid simulation, the authors define two categories: *a*) Hardware in the Simulation Loop and *b*) Simulations in the Hardware Loop. The former category combines the simulator with the hardware of a real vehicle. Data generated by the simulator is fed into the hardware, in which it is processed. Afterwards, the corresponding commands are sent back to the simulator; a simulated vehicle now moves accordingly. This methodology is different from the *HIL*-implementation of Feng et al.[9]: they use the hardware as a generator of real data. The processing takes place at the central node and the commands are sent back to the hardware, after which the real vehicle moves. The latter category makes use of a real vehicle, but its environment is mocked by the simulator. Therefore, the authors are able to safely test the behaviour of the robot without real obstacles.

Feng et al. tackles the problem in which it is difficult to test a *Cooperative Vehicle-Infrastructure System* (*CVIS*) in a real environment [9]. Therefore, the authors propose a system in which miniature cars are used in a small environment in order to simulate the behaviour of real vehicles. The system is also based on *Hardware-in-the-loop* (*HIL*), but is implemented in a different way. and contains three major components: *a*) a simulation environment, *b*) a test robot and *c*) a central control service. First, the test robot is a car at scale $1/10$ that contains a camera module for vision applications and a microcontroller as processing unit. The simulation environment contains at least one modelled version of the real car via a mathematical and electromechanical model. The control service is the central node that orchestrates the different cars in the system. The data coming from the cars is sent to the control service, in which

several navigation algorithms are running. After the data has been processed, the control centre sends the appropriate commands back to the car.

Both simulators perform well and fulfil the needs of the authors. However, these kind of simulators are not suitable for our research. We desire to be able to fully interact with a variety of heterogeneous vehicles, both real and simulated, along with scalability and flexibility in mind. Since no simulator has been designed for such a purpose yet, we propose a new hybrid methodology in the following section.

In terms of validation of our simulation framework, different methodologies exist. First, Moscato et al. introduce the MetaMORP(h)OSY framework for the verification of agreements regarding cloud computing services of different vendors [21]. These agreements are represented by models and contain several specifications, such as user requirements, features and properties. To validate this framework, the authors performed an extensive case study with two different use cases, i.e. two different kinds of agreements. In this way, the authors have concluded that the framework performs well in these use cases; thus, it has been validated successfully.

Second, Nalepa et al. propose two temporally adaptive co-operation schemes for advanced vehicle routing algorithms [23]. These schemes are designed to search for a highly acceptable routing solution, while taking co-operation between different entities into account. To validate these schemes and algorithms, the authors performed an experimental validation. They defined some Key Performance Indicators (KPIs) via which the authors have assessed the performance of their schemes. Examples of these KPIs are the average travel distances and average convergence times. By performing these assessments, the authors have concluded that their proposed schemes outperformed the existing ones.

Finally, Carstea et al. introduce an event-based simulator for testing and validating a distributed architecture of software packages for mathematical problems [5]. The designed simulator allows the authors to both validate these software packages, individually and combinations of them, and acquire useful analysis information about the real architecture by deploying these packages in a simulated grid environment. In order to validate their simulated architecture, the authors carried out two validation methods: performing a specific case study and assessing the performance and behaviour of the system. These two methods are similar to the validation methodologies of Moscato et al. and Nalepa et al.

It is clear that the validation approach of Carstea et al. is the most suitable for our application; we will not only be able to check if our system operates as intended, but also assess its performance and address improvements if necessary. Therefore, we will use this approach to validate our real-time hybrid simulator.

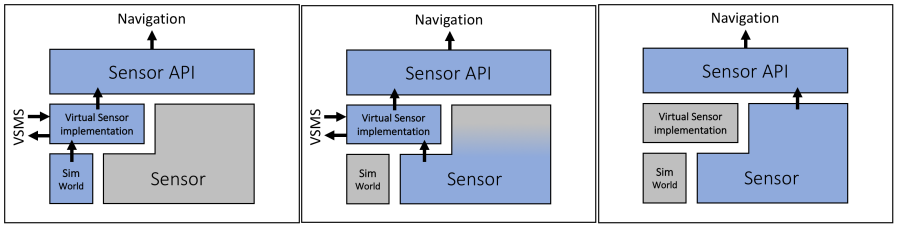
3 Concept

As discussed earlier, the goal of this research is to design a real-time simulator in which multiple real and simulated vehicles are able to fully interact with each other. Due to computational complexity and real-time requirements, we opt for a distributed approach. That is, multiple limited nodes control one vehicle each, based on the Agent Based Modelling (ABM) paradigm [4]. This means that each agent represents a vehicle with its own behaviour, while it is able to interact with the other present agents. Important to note is that an agent consists of either a real or simulated vehicle, equipped with its control logic. In case of a simulation agent, also a dedicated simulator is present. To keep track of the different vehicles, we introduce a stateful service named *Vehicle Synchronisation & Management Service* or *VSMS*. This component receives information from each agent and distributes it to the others; thus, the vehicles are aware of each other.

The *VSMS* operates as follows. First, a node registers itself at the service. After that, all information it shares with the *VSMS* is distributed to all other registered nodes. For scalability reasons, a second distributed *VSMS*-approach is under development. The bottom line is that each node knows exactly what is happening in its local context.

Based on information from the *VSMS*, a node has to make its own vehicle aware of others. To achieve this awareness, each node has a layer called *Virtual Sensor Implementation* (*VSI*), which is shown in Figure 1. This particular layer takes care of the awareness of other agents by merging data from its own sensors with information coming from the *VSMS*. In case of a simulated vehicle (Figure 1a), the *VSI*-layer generates an instance of each other vehicle and places it in its own simulated environment. Awareness is now achieved because the vehicle senses the presence of the others. In case of a real car interacting with a hybrid simulator (Figure 1b), challenges arise because of the immutable surroundings; a mock of another vehicle cannot be generated and placed into this real environment. Therefore, the sensor data of the real agent needs to be altered to achieve a modified sense of these surroundings, including simulated agents. Again, the *VSI*-layer takes care of this modification. It captures the real sensor data and modifies it based on information of the *VSMS*, after which it is made accessible through the sensor API. In this way, high level applications are now retreiving modified sensor data, thus awareness of other vehicles is achieved. In case of a real vehicle which does not interact with any hybrid simulation (Figure 1c), the original sensor data is directly available through the API without any modification.

In order to maintain a realistic simulation, an important constraint needs to be taken into account; i.e., the real-time characteristic of the overall system. Each agent needs to keep track of its own vehicle (i.e. control the physical object or perform simulation) while interacting with the others. In case of a simulated vehicle, it is important that the simulation runs in real-time for an accurate operation. However, this real-time requirement is twofold. On the one hand, one has to be aware of the computational complexity of the



(a) (b) (c)

Fig. 1: These three layer models show the sensor data flow in case of (a) a simulated vehicle, (b) a real vehicle interacting with the hybrid simulator and (c) a real vehicle without hybrid simulation.

simulator. On the other hand, the latency needs to be taken into account of communicating the sensor adaptation data to the *VSI*-layer.

4 Test setup

To test the concept of the previous section, we developed a setup that contains a simulation environment, real and simulated F1/10-cars, a navigation method and the interaction between these two kinds of cars. First, we introduce the ROS-ecosystem, which supports all of the aforementioned components in the setup. Next, we choose a simulator to model the vehicles and the virtual environment. After that, both the real and simulated cars are clarified in detail, as well as the navigation system to control these cars.

4.1 ROS-ecosystem

The ROS-environment (Robot Operating System) serves as middleware application. This open-source framework, developed by Open Source Robotics Foundation, provides different types of libraries and packages in order to develop custom robot modules or nodes. This results in a flexible, distributed and modular system at which we can add our own nodes. These nodes communicate with each other via the Publish-Subscribe-methodology. Additionally, each node communicates directly to a master node which keeps track of the global system.

4.2 Simulator

Choosing the most suitable simulator can be quite an effort as numerous robot simulators already exist. For this research, we defined a number of KPIs to do this selection. First, a simulator with built-in support for ROS (Robotic Operating System) [27] is advisable since it is used as middleware in the F1/10-car.

			Criteria comparison		
Simulator	Supported Engines	Special traits	Comp.	Ext.	Doc.
Gazebo [25]	– ODE – Bullet – Simbody – DART	C++ API, automatically installed with a desktop installation of ROS	X	X	X
Morse [20]	– Bullet	Python API, uses Blender [3] for 3D modelling and visualisation	X		
V-Rep [6]	– ODE – Bullet – Vortex	Many API's, proprietary but offers a free educational license	X	X	
OpenHRP3 [28]	– ODE – Proprietary engine	Many API's, focus on humanoid robots	X		

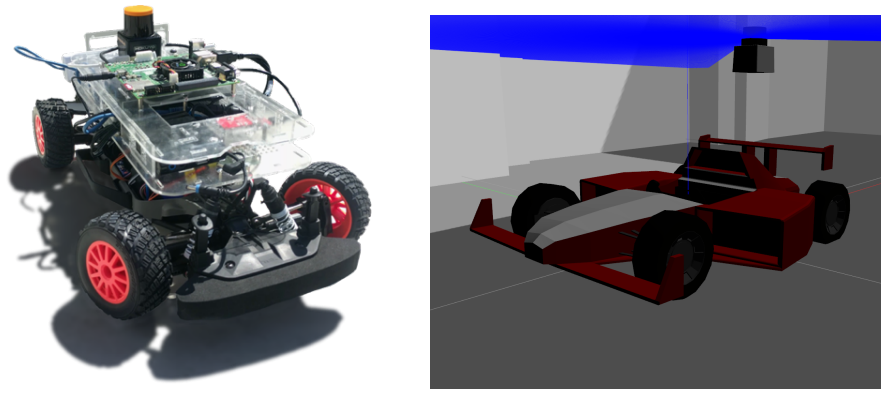
Table 1: ROS enabled non-proprietary robot simulators, along with the supported engines, special traits and a criteria comparison.

Second, we desire to simulate a variety of heterogeneous operating systems and robots, such as drones, cars and other vehicles. Therefore, approaches in which many uniform robots are simulated are not appropriate in this case. Third, the possibility to run a headless simulator (i.e. without graphical user interface) is beneficial as the final implementation may be executed in a cloud environment. Finally, the simulator should support fully dynamic robots; therefore, it needs to operate in a three dimensional environment.

Based on the aforementioned requirements and the classifications made by J. Craighead et al. [7] and J. Kramer and M. Scheutz [19], Table 1 shows four possible candidates.

The final decision was based on three criteria. The first one describes the compatibility (Comp.) with different kinds of robots. The listed simulators in table 1 all comply to this factor. The second criterion describes the extensibility (Ext.) of the simulator. Gazebo and V-Rep took the lead in this category because these simulators support the most physics engines, numerous plugins and many external libraries. The last important criterion describes the available documentation and support (Doc.). At this moment, Gazebo is the superior choice because of the giant community. Hence, Gazebo was our primary choice. N. Koenig and A. Howard wrote an overview of design and use paradigms, which can be found in [16].

Gazebo itself is controllable by using plugins written in C++. Also, by using the simulator in the ROS-environment, the simulator is able to share its data



(a) (b)

Fig. 2: These images present the F1/10-car. *(a)* is a real car, while *(b)* is the simulated model.

inside this environment. Further support for simulable ROS-components (e.g sensors, actuators, etc.) is achievable by including either plugins that already exist and are provided by Gazebo, or by designing a custom plugin that interfaces with ROS using the ROS API.
As said before, Gazebo provides support for several physics engines. The default engine is Open Dynamics Engine (ODE)[32], but this can be changed if necessary. However, we opted not to because ODE covers our requirements. Though, it is important to know that this is possible if the specifications would change in the future.

4.3 F1/10-car

As mentioned earlier, we chose to use an F1/10-car for both real and simulated vehicles. On the one hand, the real car contains a processing unit as well as multiple sensors. The simulated car, on the other hand, is a modelled version of the real one with an external processing unit and virtual sensors. The following paragraph deals with the real vehicle while the second paragraph elaborates on the simulated car.

4.3.1 Real car

The car used in this research (shown in Figure 2a) is driven by an NVIDIA Jetson TK1 development board, containing a Tegra K1 from the same manufacturer as System on a Chip (SoC) [24]. This SoC consists of a quad core CPU, which is made of four ARM Cortex-A15-cores. The CPU is assisted by 192 CUDA-cores as graphical unit. Ubuntu 14.04 is used as on-board oper-

ating system. In order to provide network capabilities, a Ubiquiti Picostation M2 is used as Wi-Fi-bridge and is connected to a local router.

Additionally, multiple sensors are present. First, a Hokuyo UST-10LX serves as LiDAR-sensor on the robot [13]. With a refresh rate of 40 Hz and an angle resolution of 0.25° over a range of 270°, this sensor is capable of both fast and accurate measurements. The sensor produces messages containing metadata about the measurement, along with two arrays of numbers. The first array contains the measured distance of the sample, while the second array contains the intensity of that sample. Each array consists of 1080 single precision floating-point numbers; hence, the size of such a message is approximately 8.7 *kilobytes*. Second, a Razor IMU-sensor (Inertial Measurement Unit) from Sparkfun is available [33][14]. This sensor consists of a magnetometer, gyroscope and accelerometer, all of which have three axes, thus having nine degrees of freedom in total. The output of both of these sensors is handled by different ROS-nodes, which in turn publish the information to other nodes.

The body of the car itself is made by Traxxas. The Velineon VXL-3s serves as motor controller or Electronic Speed Control (ESC) and is connected to a Teensy 3.2 development board, containing an ARM Cortex-M4 processor. The steering servo is also connected to this development board. The microcontroller generates two PWM-signals (Pulse Width Modulation) that are sent to the ESC and servo in order to control the behaviour of the car. This microcontroller also communicates with other nodes via ROS.

4.3.2 Simulated car

In order to increase the accuracy of the simulation, a precise model of the F1/10-car is needed. Therefore, a conversion needs to be made from the physical and behavioral properties of the car to a description format. Two model descriptors exist: Unified Robot Description Format (URDF) [35] and Simulation Description Format (SDF) [26]. The former only describes the kinematic and dynamic aspects of a single robot, while the latter also takes environmental properties (e.g. the position in the scene) into account. Hence, for this research, we opted for the SDF-format to describe both the environment and the F1/10-car.

The SDF-model comprises several components: a chassis, four wheels that have individual suspension and spin, a LiDAR-scanner and an IMU. The model also incorporates an implementation of Ackermann Steering and four-wheel differential drive. An overview of the implementation is illustrated in Figure 3 and Table 2. The model itself is shown in Figure 2b.

Similar to the real F1/10-car, the simulated one publishes its sensor data from both the IMU and the LiDAR scanner, along with its speed. The model is controllable by publishing proper ROS messages to the topic at which the Gazebo plugin of the model is subscribed to.

Weight	Suspension for every wheel	Collision model
Chassis = 10kg	Spring stiffness front = 10000, back = 15000	Box
Wheel = 0.2kg	Damping front = 150 , back = 200	Cylinder
LiDAR = 0.1kg	Friction = 0.7	Cylinder

Table 2: Dynamic settings used by a simulated F1/10

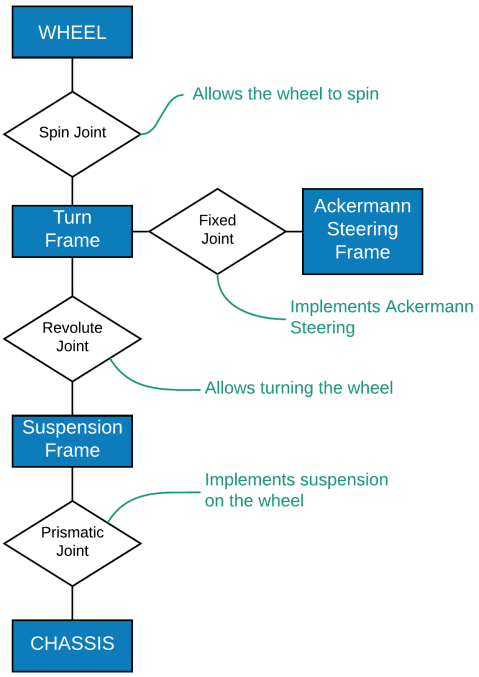


Fig. 3: Block representation of the Gazebo links and frames implementing the front wheels in Gazebo

4.4 Navigation algorithm

For this research, the ROS Navigation Stack has been implemented. This stack consists of multiple nodes and packages. One of these nodes is the *move_base*-node, which has a major role in the system. It provides the linkage between the global and local planning algorithms, along with several other input and output nodes. The required inputs are the odometry data, transformations, localisation information, the map on which the navigation takes place and the data from a rangefinder. In this case, the LiDAR-sensor serves as rangefinder. The commands by which the car will drive are the generated output of the central node.

The transformations needed by the *move_base*-node are used to provide a linkage between the location of the LiDAR-sensor and the frame of the car. In this way, the sensor data will be transformed to data relative to the center of the car. This results in an abstract location of the sensor on the car, as long as its transformation to the frame changes accordingly.

To perform localisation on the F1/10-car, we use *Adaptive Monte Carlo Localization* (*AMCL*)[10]. This is an algorithm designed for mobile robot localisation, based on probabilistic methods and particle filters. Based on the given LiDAR-data, the current map and transformations, the algorithm calculates a multitude of particles; each particle is a guess of the current location. Based on these particles, the node outputs a single location estimation. In this experimental setting, the amount of particles ranges between 500 and 5000. The algorithm itself will choose how many particles it calculates based on the given input.

The global planning algorithm calculates a route between a given startpoint and endpoint on a given map. This is done by a costmap generated from an environmental map, which in turn is made by a procedure called *hector_mapping*. This is an open source SLAM (Simultaneously Localisation And Mapping) algorithm by Team Hector, designed for the RoboCup Rescue Competition [17, 18]. The shortest-path algorithm from Dijkstra was used for path finding [8].

The local planner generates a costmap based on actual LiDAR-data and the current position estimation of *AMCL*, whereafter a route is calculated around obstructions that are not located on the global map. This calculation is carried out by a node called *base_local_planner*, based on the research of Gerkey et al. [12].

Both the real and simulated cars are equipped with the ROS Navigation Stack and make use of the same environmental map. This map can be seen in Figure 5a.

4.5 Interaction between real and simulated cars

As mentioned in the previous section, one has to make sure that a real vehicle is aware of the others by modifiying its sensor data. In order to modify the data from a LiDAR-sensor, we opted to use a central ray trace mechanism that is located inside the *VSMS*. Based on the computational complexity and the available resources of the host, we chose to implement this at a powerful host instead of the processing unit of the real car.

The original sensor messages are sent from the real car to the *VSMS*, after which the ray trace sequence takes place. In here, the mechanism determines for each laser beam whether it hits with a simulated vehicle. It does so based on positional information of the other vehicles, coming from the *VSMS*. If a laser beam hits with an object, the length of this beam is shortened to the first hit point it encounters. In order to ray trace simulated objects in an efficient

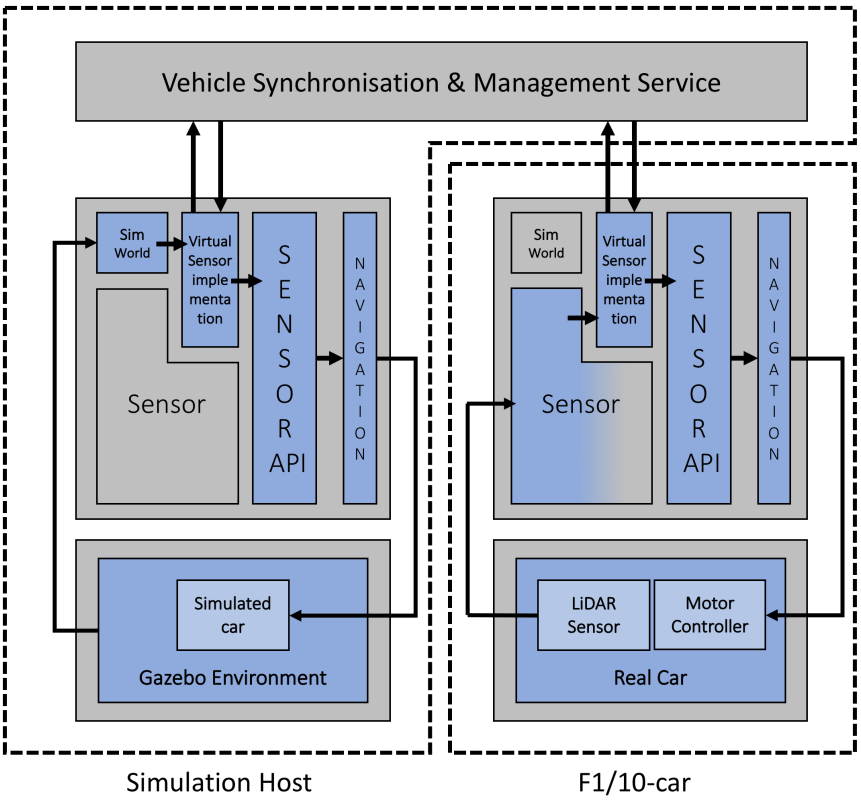


Fig. 4: Software diagram with the different data flows

way, we created bounding boxes around those simulated object, which are then used in the ray trace sequence. In this way, this sequence does not have to take the complex shape of the modelled car into account.

As discussed earlier, in case of a simulated car, a representation of other vehicles is placed in the simulated environment in order to achieve awareness. The modelled car contains a LiDAR-sensor as well, which has a built-in mechanism to sense its environment. In this way, the simulator automatically provides the awareness of other vehicles.

4.6 Hybrid simulator setup

Figure 4 shows the different flows of sensor and actuator data streams between the software blocks. In this setup, both the *VSMS* and the simulator are located on one host, while the software of the real car is located on the car itself. The *VSMS* also contains the ray trace mechanism, which is elaborated in the previous section.

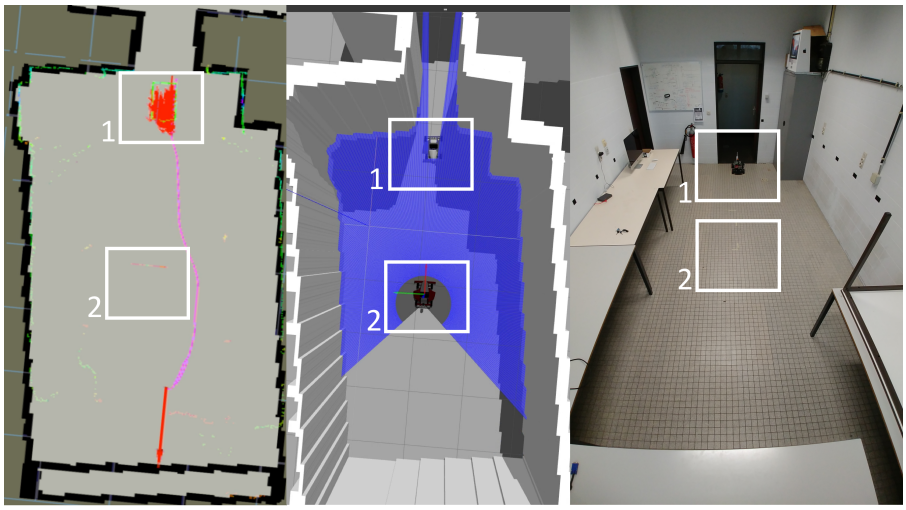


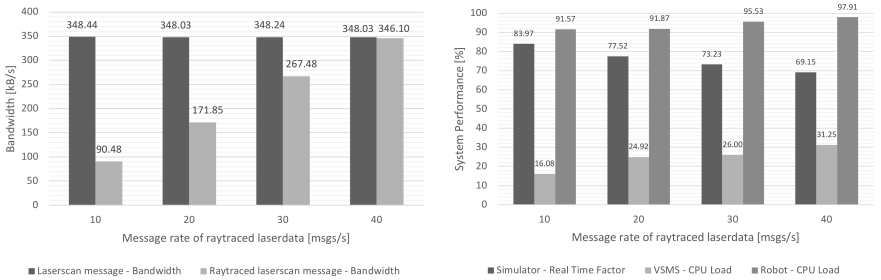
Fig. 5: Environmental setup containing one real car (Rectangle 1) and one simulated car (Rectangle 2).

For verification and testing purposes, we provided visualisation of the hybrid environment from the perspective of both the real and the simulated car, as shown in Figure 5. Boxes 1 and 2 in the three subfigures are respectively the real and simulated vehicle. Figure 5c shows the room in which we tested our hybrid simulator. Figure 5a is the view of the real car and its sensor data, whereas 5b displays the simulated environment and its modelled car in Gazebo. In Figure 5a, the trajectory of the real vehicle around the simulated object is also shown. The collision avoidance was achieved by injecting the pose estimation of the simulated vehicle into the sensor data of the real vehicle as discussed in the previous section.

5 Results

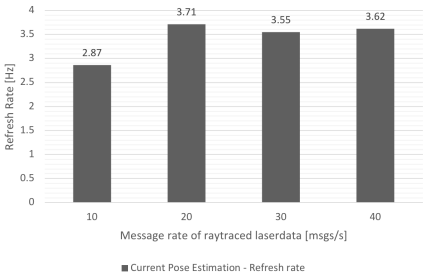
For this research, we performed multiple experiments to assess the performance of the system via four KPIs: *a*) the used bandwidth of the modified sensor data, *b*) the Real Time Factor (RTF), *c*) the CPU load of the *VSMS* and *d*) the rate at which a new pose estimate is calculated. The tests are executed by performing multiple measurements for each KPI, after which these have been averaged to a single result.

First, we evaluated the impact of the rate of updated sensor data on the performance of both the car and simulator. In the second experiment we changed the settings of the physics engine of Gazebo in order to evaluate the performance impact.



(a)

(b)



(c)

Fig. 6: These plots show the impact of an increasing rate of modified sensor messages on the network bandwidth (a), the performance of the simulator and Vehicle Synchronisation & Management Service (b) and the rate at which the car is able to calculate a new position (c) and . The network bandwidth of the original laser messages is added for reference purposes.

5.1 Impact of modified sensor message rate on performance

In this experiment, we examined the four KPIs and the impact on them when changing the update rate of the modified sensor messages:

The resulting impact on these indicators is shown in Figure 6. If we take a look at the network bandwidth of the modified data in Figure 6a, it is clear that it increases in proportion to the message rate. When this rate is equal to 40 messages/s (msgs/s), the bandwidths of the original and modified sensor data are approximately the same; this is obvious as the original data also has a rate of 40 Hz.

When looking at the system performance in Figure 6b, two interesting evolutions are visible. First, the Real Time Factor decreases noticeably, which causes the simulator to diverge from the real-time characteristic since a factor of 1.0 is considered as real-time. Only the simulated vehicles are affected as the real one is not aware of this factor. Second, the load of the management service increases, which causes the decrease of the RTF; the *VSMS* needs to process and raytrace more messages per second. In addition to these two

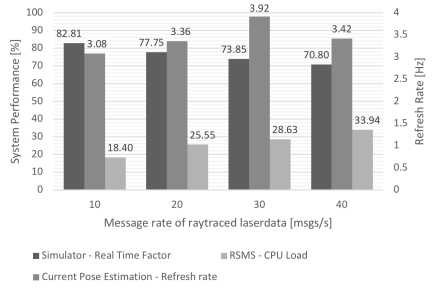
evolutions, the plot shows the CPU load of the car as well. A small increase is noticeable because the car processes more data when performing localisation and navigation.

Finally, Figure 6c shows the rate at which the car calculates a new position via *AMCL*, based on the modified sensor data. The average update frequency is approximately 3.6 Hz. However, at a rate of 10 messages per second, we notice a frequency drop which is quite remarkable. Also worth mentioning is that two out of five experiments with 10 msgs/s failed. The vehicle was unable to calculate its current position at a regular basis, thus it could not navigate properly whereafter the car got stuck.

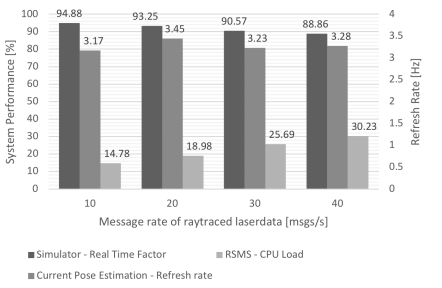
Based on these results, we are able to draw three conclusions. First, we can conclude that a rate of 20 messages per second is enough in this case. It uses less bandwidth, has a lower load on the *VSMS* and retains a relatively high Real Time Factor. However, this rate is not high enough when driving at an increased speed. Therefore, a more viable solution is to adaptively manage the message rate, depending on the speed of the car. In this way, we keep the advantages of a low message rate at a slow speed, while retaining the accuracy with a high message rate when driving faster. Second, it is not recommended to deploy the *VSMS* and simulator on the same machine. This is due to the fact that the load of the *VSMS* has an influence on the Real Time Factor of the simulator, which is not desired. Therefore, we suggest to use different machines to make sure the simulation performance is not dependent on additional loads. Third, to maintain scalability, the location of the ray trace mechanism needs to be revised. At this moment, we tested with one real car. However, when deploying two or more cars, the *VSMS* needs to execute the ray trace sequence at least twice, which has a tremendous impact on the overall load. Additionally, every sensor message is sent twice over the network; first it sent to the ray tracer, after which the message is sent back to the car. Therefore, we suggest that the ray trace mechanism needs to be present at the real car itself instead of in the management service. This location will also have an impact on the load of the car, but it is a viable solution when optimising both the ray trace mechanism and the navigation system. On top of that, the used bandwidth is reduced since the sensor messages are not sent over the network anymore and only the position updates need to be communicated.

5.2 Impact of ODE step size and update rate on performance

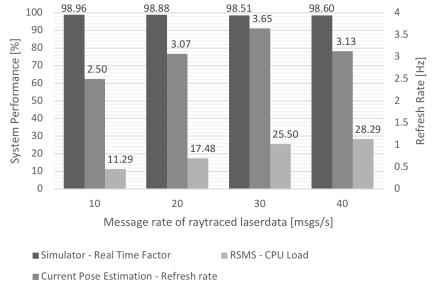
In the second experiment, we changed the settings of the physics engine in the simulator (i.e. the discrete step size and update rate) to examine the impact on the overall performance. Figure 7a shows the results when using a step size of 0.001 s with an update rate of 1000 Hz. We notice that the simulation does not achieve a real-time characteristic due to the high load of the physics engine and the increasing load of the *VSMS*. The refresh rate of the pose estimation is noteworthy; it peaks at 30 messages/s and decreases again at 40 messages/s, while the rate is low at both 10 and 20 message/s. This evolution is different



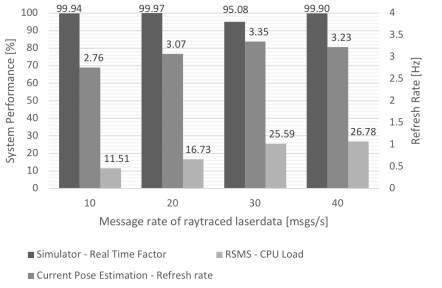
(a) Step size: 0.001 s - Update rate: 1000 Hz



(b) Step size: 0.002 - Update rate: 500 Hz



(c) Step size: 0.004 s - Update rate: 250 Hz



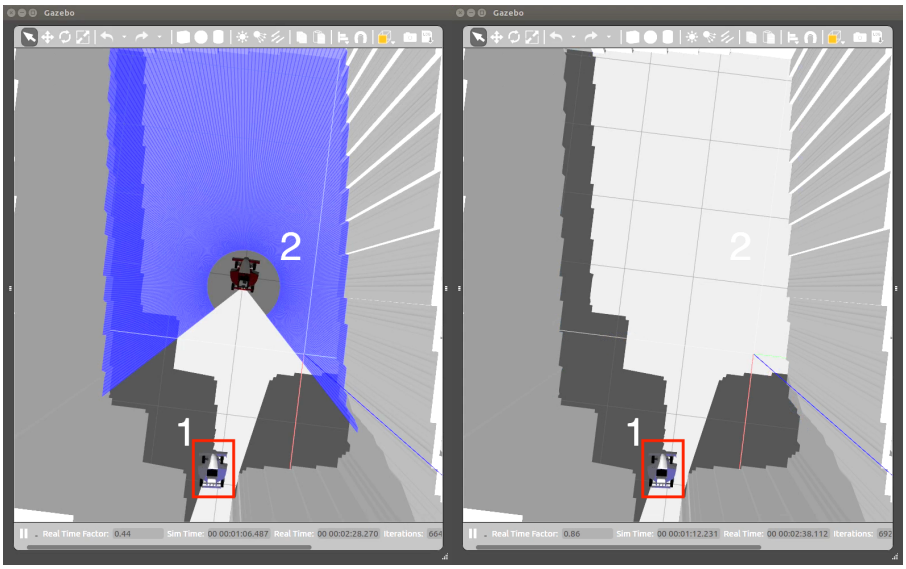
(d) Step size: 0.008 s - Update rate: 125 Hz

Fig. 7: These plots show the relation between different simulator settings of the physics engine (discrete step size and update rate) and the performance of the system (RTF 19[%], CPU Load of the *VSMS* [%] and the update rate of a new pose estimation [Hz]).

from the previous experiment, which is remarkable as both experiments are executed with the same settings. The reason lies in the dynamic character of the ROS Navigation Stack and its many uncontrollable parameters (e.g. the accuracy of the pose estimation in *AMCL*, the local planner and its dynamic plan around the simulated obstacle, etc.).

In Figure 7b, the RTF is closer to 1.0, thus closer to a real-time characteristic. As with the previous settings, the CPU load of the *VSMS* still increases, but is marginally lower. This is because of the lower publish rate of the simulator and its vehicle, which causes the database of the *VSMS* to be updated more slowly. The pose estimation update frequency is positioned at approximately 3.3 Hz, which is lower than with the previous settings, but its progression is almost equal to the one of the previous experiment. This, once again, has to do with the navigation module on the car.

For both figures 7c and 7d, the RTF is nearly 1.0, thus achieving the real-time characteristic. Additionally, the CPU loads in both figures are still increasing with the message rate and are almost equal to each other. The overall evolution of the pose update frequency is also equal, although Figure 7c has a lower minimum and a higher maximum. Important to note is that



(a) Before the simulation failure (b) After the simulation failure

Fig. 8: (a) and (b) show the simulator before and after its failure. Car 1 is a representation of the real one, while car 2 is the simulation model.

the simulation in Figure 7d failed at each measurement. This is because of a high inaccuracy in the physics engine, caused by the high discrete step size. Figure 8a shows the moment before the failure, while Figure 8b shows the state immediately after the failure. It is the same world as before, but without a simulated vehicle. A few moments later, the whole simulation environment crashes, after which it needs to be restarted.

Based on this experiment, we can conclude that a simulation with a low step size and high update rate is not necessarily required to ensure a well-performing system; even a step size of 0.004 seconds is feasible. However, we recommend to simulate at high settings if the machine is capable enough to do so while retaining an RTF of at least 1.0. In this way, the performed simulation is a more accurate approximation of the car and its behaviour in reality.

5.3 Summary & discussion

In the previous experiments we have concluded that some major improvements are necessary in order to maintain a scalable hybrid simulation. At first, a low message rate is suitable when driving at a slow speed. Yet, a more adaptive approach is recommended in which the message rate is based on the speed of the car. This is to ensure a high efficiency and accuracy at both low and high speeds when driving. Next, the *Vehicle Synchronisation & Management Service* needs to be separated from every other node because of the influence

it has on their performance. The sensor modification sequence, which is a ray tracing algorithm, also needs to be separated from the central service. This implies two big advantages. First, it will be more easy to maintain scalability as the computation complexity of the *VSMS* significantly decreases. Second, the used bandwidth will be tremendously reduced as these messages are solely processed in the agent itself.

Furthermore, a simulation host is able to retain a Real Time Factor of nearly 1.0, even if it does not have enough resources. This can be accomplished by lowering the resolution of the physics engine. However, if the host is powerful enough, a high resolution is recommended to ensure that the simulation model is an accurate and precise representation of a real car. Another approach is adaptively adjusting this resolution, i.e. a low resolution when driving slow and a high one when driving fast. An additional solution can be found in dividing the environment in different sectors. Each sector has its own resolution settings, thus also resulting in a dynamic approach.

With these improvements in mind, the second version of the overall system will be more scalable, adaptive and efficient.

As elaborated in section 2, we opted to validate this simulator with two different methods, i.e. by applying a use case and by performing assessments. The application of the use case has been performed by integrating a real car with the simulation environment. Therefore, the first validation method has been successfully completed. The second method was also successful as the assessments have been carried out in this section and conclusions are drawn.

6 Conclusion & Future Work

To thoroughly test and validate algorithms and systems of autonomous vehicles, a large number of vehicles, many tests and a multitude of datasets are needed. This way of developing and testing is difficult, expensive and sometimes even dangerous. Therefore, the need for a proper simulator is rising. As different kinds of simulators already exist, such as Hardware-In-the-Loop, none of them, however, is capable of handling both real and simulated vehicles simultaneously, with full interaction, in real-time. In this paper we proposed a novel methodology for a real-time hybrid simulator that is capable of handling both real and simulated vehicles simultaneously, with full interaction, in real-time. This hybrid approach combines the benefits of real world testing with the scalability and lower cost of simulation-based testing. In addition, we proposed a methodology for real-time synchronization between real and virtual vehicles, in which the presence of simulated vehicles was injected in the sensor streams of real vehicles and vice versa.

We validated this methodology with two experiments in which we evaluated the overall performance of the hybrid simulator based on four KPIs: *a*) the used bandwidth of the modified sensor data, *b*) the Real Time Factor (RTF), *c*) the CPU load of the management service and *d*) the rate at which a vehicle

calculates a new pose estimate. The first test measured the impact of the updated sensor message frequency on these four KPIs. In the second test, we carried out the impact of adjusting the step size of the simulators physics engine on the overall performance. The results showed that real-time hybrid simulation could indeed be achieved and that the methodology can be used for testing navigation algorithms of an autonomous vehicle, without the risks of testing this with an entirely physical test bed.

Future improvements to this approach include further optimization of the communication and processing overhead towards increasing the scalability of our simulator. We also aim to extend this approach with various other entities including other types of vehicles to enable research in other domains where safe and robust testing is needed. In addition, we intend to integrate our hybrid simulator with real use cases. For example, Ito et al. developed a system to monitor the condition of roads using multiple in-car sensors while communicating this data to the infrastructure and other vehicles [15]. This system can be deployed in our simulation environment by providing simulated cars, a mocked infrastructure to communicate with and altered sensor data for the real cars. Finally, we aim to integrate our simulator with autonomous vehicles to validate the interaction with the next generation of smart environments, paving the path towards novel applications in smart cities, highways and harbours.

References

1. Argonne System Modeling and Control Group (n.d.) Autonomie - Expertise
   Vehicle System Tool. https://www.autonomie.net/expertise/Autonomie.html, Accessed February 2018
2. Baskar LD, Schutter BD, Hellendoorn H (2012) Traffic management for automated highway systems using model-based predictive control. IEEE Transactions on Intelligent Transportation Systems 13(2):838–847, DOI 10.1109/TITS.2012.2186441
3. Blender Foundation (2018) blender.org - Home of the Blender Project. https://www.blender.org/, Accessed March 2018
4. Bosmans S, Mercelis S, Hellinckx P, Denil J (2018) Towards evaluating emergent behavior of the internet of things using large scale simulation techniques (wip). In: Proceedings of the Theory of Modeling and Simulation Symposium, Society for Computer Simulation International, San Diego, CA, USA, TMS '18, pp 4:1–4:8, URL http://dl.acm.org/citation.cfm?id=3213187.3213191
5. Carstea A, Frincu M, Macariu G, Petcu D (2011) Validation of SymGrid-services framework through event-based simulation. International Journal of Grid and Utility Computing 2(1):33–44
6. Coppelia Robotics GmbH (2017) V-rep. http://www.coppeliarobotics.com/, Accessed May 2017
7. Craighead J, Murphy R, Burke J (2007) A survey of commercial & open source unmanned vehicle simulators. In: Robotics and Automation
8. Dijkstra EW (1959) A note on two problems in connexion with graphs. Numerische mathematik 1(1):269–271
9. Feng S, Guan Z, Du F (2016) Building of Hardware-in-the-loop Simulation Platform Based on Vehicle-infrastructure Cooperation Environment. In: 3rd International Conference on Artificial Intelligence and Pattern Recognition, AIPR, pp 94–98, DOI 10.1109/ICAIPR.2016.7585218
10. Fox D, Burgard W, Dellaert F, Thrun S (1999) Monte Carlo Localization: Efficient Position Estimation for Mobile Robots. Proceedings of the national conference on Artificial intelligence July 18-22(Handschin 1970):343–349, DOI 10.1.1.2.342
11. Gechter F, Dafflon B, Gruer P, Koukam A (2014) Towards a Hybrid Real / Virtual Simulation of Autonomous Vehicles for Critical Scenarios. In: Arisha A, Bobashev G (eds) SIMUL 2014: The 6th International Conference on Advances in System Simulation, Nice, France, pp 14–17
12. Gerkey BP, Konolige K (2008) Planning and control in unstructured terrain. In: ICRA Workshop on Path Planning on Costmaps
13. Hokuyo Automatic Co, Ltd (2015) UST-10LX Specification. https://www.hokuyo-aut.jp/dl/UST-10LX_Specification.pdf, Rev. 1
14. InvenSense Inc (2014) MPU-9250 Product Specification. https://www.invensense.com/wp-content/uploads/2015/02/PS-MPU-9250A-01-v1.1.pdf, Rev. 1.1

15. Ito K, Hirakawa G, Arai Y, Shibata Y (2016) A road condition monitoring system using various sensor data in vehicle-to-vehicle communication environment. International Journal of Space-Based and Situated Computing 6(1):21–30
16. Koenig N, Howard A (2004) Design and use paradigms for gazebo, an open-source multi-robot simulator. Intelligent Robots and Systems
17. Kohlbrecher S, Meyer J, von Stryk O, Klingauf U (2011) A flexible and scalable slam system with full 3d motion estimation. IEEE International Symposium on Safety, Security and Rescue Robotics
18. Kohlbrecher S, Meyer J, Graber T, Petersen K, Klingauf U, von Stryk O (2013) Hector open source modules for autonomous mapping and navigation with rescue robots. In: Robot Soccer World Cup, Springer, pp 624–631
19. Kramer J, Scheutz M (2007) Development environments for autonomous mobile robots: A survey. Autonomous Robots 22(2):101–132
20. LAAS-CNRS (2017) MORSE, the Modular OpenRobots Simulation Engine. `https://www.openrobots.org/wiki/morse/`, Accessed May 2017
21. Moscato F, Amato F, Amato A, Aversa R (2014) Model Driven Engineering of Cloud Components in MetaMORP(h)OSY. International Journal of Grid and Utility Computing 5(2):107–122
22. Mui C (2017-07-20) Waymo Is Crushing The Field In Driverless Cars. `https://www.forbes.com/sites/chunkamui/2017/02/08/waymo-is-crushing-it`, Accessed May 2017
23. Nalepa J, Blocho M (2017) Temporally adaptive co-operation schemes. In: Xhafa F, Barolli L, Amato F (eds) Proceedings of the 11th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC-2016), Springer International Publishing, Cham, Switzerland, pp 145–156
24. NVIDIA Corporation (2018) Jetson TK1 Embedded Development Kit. `http://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html`, Accessed February 2018
25. Open Source Robotics Foundation (2017) Gazebo. `http://gazebosim.org/`, Accessed May 2017
26. Open Source Robotics Foundation (2017) Simulation description format. `http://sdformat.org/`, Accessed May 2017
27. Open Source Robotics Foundation (2018) ROS.org — Powering the world's robots. `http://www.ros.org`, Accessed March 2018
28. OpenHRP Team (2017) Openhrp3. `https://fkanehiro.github.io/openhrp3-doc/en/`, Accessed May 2017
29. of Pennsylvania U (2016) The Official Home of F1/10. `http://f1tenth.org/`, Accessed May 2017
30. Reese H (2016-01-20) Autonomous driving levels 0 to 5: Understanding the differences. `https://www.techrepublic.com/article/autonomous-driving-levels-0-to-5-understanding-the-differences/`, Accessed March 2018

31. Schöner HP (2017) The Role of Simulation in Development and Testing of Autonomous Vehicles. In: Driving Simulation Conference, Stuttgart, vol DSC 2017, DOI 10.1179/msi.2008.3.1.5, [Keynote Presentation]
32. Smith R (2007) Open Dynamics Engine. http://www.ode.org/, Accessed March 2018
33. Sparkfun Electronics (2016) SparkFun 9DoF Razor IMU M0. https://www.sparkfun.com/products/14001, Accessed February 2018
34. Speculations G (2016-09-20) Self-Driving Cars: The Building Blocks of Transportation-as-a-Service. https://www.forbes.com/sites/greatspeculations/2016/09/20/self-driving-cars-the-building-blocks-of-transportation-as-a-service/, Accessed May 2017"
35. Sucan I, Kay J (2017) Urdf. http://wiki.ros.org/urdf, Accessed May 2017