Development, Simulation, and Validation Environment for Autonomous Driving Algorithms Based on a ROS Architecture

Constantin Blessing, Reiner Marchthaler

***Abstract*—Virtual environments for development, simulation, and validation of robot applications are indispensable, constituting a cheap, safe, and often faster alternative to working with real-world vehicles. Consequently, the scope of this paper covers the design and implementation of such an environment geared to the needs of smaller scaled autonomous driving applications based on a robot operating system architecture.**

***Index Terms*—Autonomous Driving, Simulation, Robot Operating System, Vehicle Dynamics, Robotics**

I. Introduction

In an era of rapid technological advancements, the quest for autonomous driving (AD) has emerged as a pivotal challenge in the realm of computer engineering. With the aim of increasing vehicle safety, optimizing traffic flow, and revolutionizing transportation systems, researchers and engineers have turned their attention toward developing intricate algorithms capable of enabling vehicles to navigate and operate autonomously. But achieving autonomy comes at the cost of increasing algorithmic complexity and corresponding validation.

While the commercial automotive industry offers extensive and powerful ecosystems for developing and validating AD algorithms — often advertised for their certification capabilities — their feature set and complexity are often excessive for smaller scaled projects where such aspects are less relevant.

Therefore, the objective of this paper is to investigate the feasibility of a practical, functional, and easy-to-use framework that supports the efficient development, simulation, and validation of AD algorithms for smaller scaled applications.

The resultant environment is presented in the context of the autonomous model vehicle depicted in Figure 1. This vehicle's software stack is built on top of the robot operating system (ROS) and its sensor suite encompasses a stereo RGB camera for lane and object detection, two reflectance sensors to measure the wheel speeds of the front left-hand and front right-hand wheels, respectively, and an inertial measurement unit (IMU) capturing rotation and acceleration [2].

Constantin Blessing, constantin.blessing@hs-esslingen.de, Reiner Marchthaler, reiner.marchthaler@hs-esslingen.de. Esslingen University of Applied Sciences, Flandernstraße 101, 73732 Esslingen, Germany.

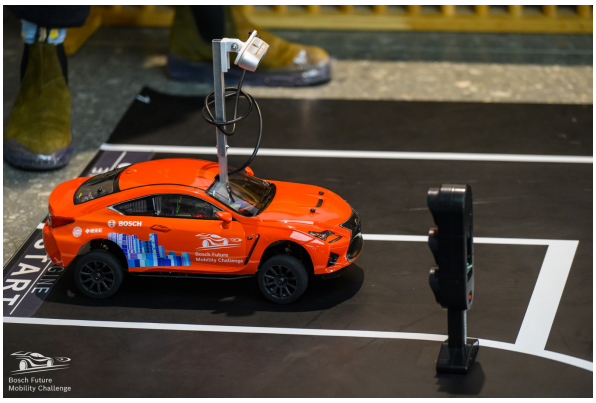

Figure 1. The real-world vehicle serving as the basis for the digital counterpart. Source: [1]

The remainder of this paper is structured as follows: Section II investigates currently available simulation and development frameworks relevant to the objective of this work, and section III outlines the driving factors behind the development, simulation, and validation environment presented in this paper. The environment's goals and objectives are subsequently highlighted in section IV. Continuing with Section V, the conceptual ideas behind the environment are discussed, followed by the actual implementation in section VI. An overview of the resultant environment can be found in section VII. Finally, section VIII and IX will conclude the paper and provide an outlook, respectively.

II. Related Work

J. Collings et al. provide an overview of numerous physics-based simulators for robotics applications [3]:

The AirSim simulator features simulated environments powered by Unreal Engine and is focused on the simulation of aerial vehicles such as drones, but also features support for wheeled vehicles [4]. Due to being built on top of Unreal Engine, AirSim can portray environments with a high degree of realism, lending itself well to generating camera-based training data for machine vision applications. In addition to cameras, sensors such as IMU, GPS, and barometer

are part of its supported sensor suite [4]. AirSim can serve as a simulation and development environment on all major operating systems, i.e. Linux, Windows, and MacOS. First published in 2017, it is scheduled to be archived in 2024 and will be replaced by Project AirSim [5].

Arguably the most well known simulator is Gazebo. Being maintained by Open Robotics, Gazebo possesses good integration with ROS. The Gazebo simulator incorporates an accurate physics simulation and a wide range of sensors, such as IMU, LiDAR, and camera [6, 3], although its suitability for high-fidelity image generation is limited compared to Unity- or Unreal Engine-based simulations [3]. In addition, if the existing array of sensors is insufficient, Gazebo features a plugin system through which new sensors can be implemented. In terms of cross-platform functionality, Gazebo is best suited for a Linux-based operating system (OS). Windows operating systems are only supported by the community, hence full functionality is not officially guaranteed [7].

A popular alternative to Gazebo is CoppeliaSim, a versatile and scalable robot simulation framework [8]. Worth mentioning is CoppeliaSim's capability to embed functionality directly within its environment using Lua scripting, additionally making it a powerful development environment. Moreover, like Gazebo, it features a large array of preimplemented sensors but also lacks sophisticated realistic rendering [3]. Noteable is the inclusion of a path planning module that can handle non-holonomic vehicles like cars. Similarly to AirSim, this simulation framework supports all major operating systems, although with the disadvantage of not being free for commercial use.

Alternatively, NVIDIA Isaac Sim is a high-fidelity robotics simulator built on Omniverse, offering PhysX-powered physics and ray-traced rendering for realistic sensor simulation [9]. Unlike Gazebo and CoppeliaSim, it excels at photorealistic perception data generation, making it ideal for AI-based applications. However, it requires NVIDIA GPUs for optimal performance and is primarily optimized for Linux, limiting its versatility compared to cross-platform alternatives.

Lastly, [10] derives a framework for vehicle control and simulation based on ROS and Unity, leveraging ROSBridge for cross-communication between the two tools. Through two validation use cases, [10] details their framework's support for non-holonomic robots and sensors such as LiDAR. Also described in the paper is a quite extensive manual setup process for the proposed framework. Finally, it is not immediately clear whether the introduced framework is cross-platform.

III. Motivation

The motivation underpinning this research stems from several key challenges that pervade the landscape of embedded development, specifically the development of autonomous driving algorithms:

1) **Limited Hardware Availability:** The tangible constraints of time, resources, and safety considerations impede the expansive deployment of autonomous vehicles for rigorous real-world validation and testing.
2) **Demands of AI-based Algorithms and Training Data:** The potency of AI-based autonomous driving algorithms hinges on the acquisition and use of substantial training data. However, generating such data manually through the physical vehicle proves to be an arduous undertaking.
3) **Cumbersome Iterative Development Process:** The iterative process of coding, deployment, observation, and data retrieval with a physical vehicle can be cumbersome and time-consuming.

IV. Goals

The goals defining the environment's scope are the following:

1) **Minimal Setup:** The creation of a user-friendly standalone desktop application with as few dependencies as possible and a minimal setup.
2) **Cross-Platform:** Whether a user or developer works on Windows or Linux operating systems, cross-platform capability ensures that they can harness the environment irrespective of their OS.
3) **Extensible and Modular Architecture:** A flexible, modular design, allowing easy integration of new components and features, is an essential objective of the environment. This ensures adaptability and simplifies future enhancements.
4) **Testing and Validation of Vehicle Software:** An important goal is to provide a platform for rigorous testing and validation of the vehicle software stack. The vehicle software stack should be testable as-is to the extent feasible.
5) **Iterative Prototyping of Vehicle Algorithms:** Rapid modification, implementation, and evaluation of new algorithmic approaches foster an environment that encourages innovation and experimentation.

V. Concept

Figure 2 illustrates the high-level architecture of the development, simulation, and validation environment derived in this work. Focusing on the simulation aspect, the structure can be roughly divided into two parts.

The first part is realized by the Unity game engine which is responsible for simulating the vehicle and its surroundings. Additionally, to create an easily extensible and modular architecture, an operating system framework, residing inside the simulator itself, has been conceptualized. Besides controlling the simulator

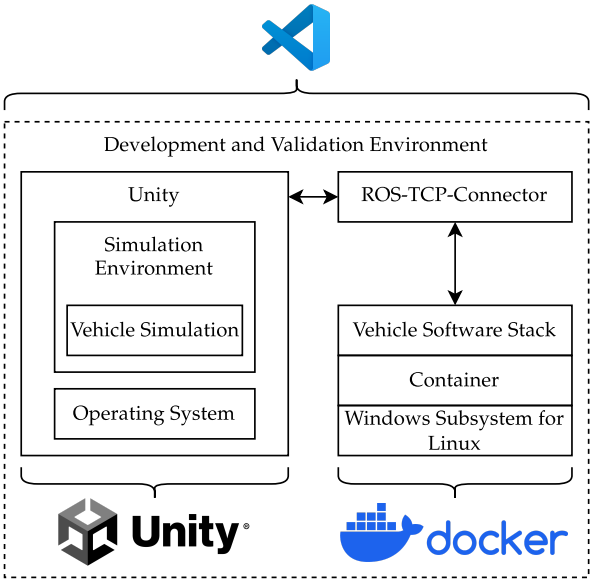

Figure 2. The architecture of the environment on a Windows OS. On a Linux OS the block "Windows Subsystem for Linux" is not present.

and managing the communication with the vehicle software stack, its purpose is to offer a common, homogeneous interface for future extensions while avoiding some common pitfalls with a potentially ever-growing set of features:

The user interface can become cluttered and difficult to navigate. Icons, buttons, and menus can start competing for screen space, making it harder for users to find the functions they need. Moreover, new features can potentially introduce performance bottlenecks, especially if they require extensive processing power or memory usage. Finally, users may also become overwhelmed by the sheer number of features, not knowing where to start or how to use the software effectively.

The second part uses Docker to containerize the vehicle software stack. This ensures cross-platform readiness and also automates the installation of the vehicle software stack and its dependencies.

Under the hood, communication between the simulated environment and the ROS-based vehicle software stack operating inside the Docker container is achieved by Unity's ROS-TCP-Connector. Through it, sensor data will be channeled into the software stack and actuator data will, in turn, be captured and applied to the vehicle simulation by leveraging ROS' topic-based communication. Notably, this connector employs a separate TCP socket through which binary data is transceived, making it substantially faster than ROS-Bridge's JSON-based approach [11]. Finally, it's worth mentioning that any vehicle software stack that is compatible with the simulator's ROS topics can be integrated.

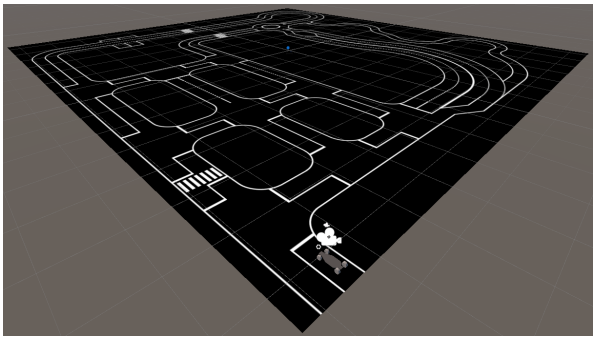

Figure 3. The simulator as visible inside Unity's scene view.

Surrounding the simulation component is the development and validation environment. The former is largely enabled by Visual Studio Code and its Docker integration, whereas the latter relies on the well-established suite of validation and analysis tools provided by ROS itself to enable testing and verification of vehicle algorithms.

VI. Implementation

The simulated environment (simulator) comprises a flat plane with a basic road network consisting of city streets, a roundabout, pedestrian crossings, as well as a highway and a country road section mapped on top, as seen in figure 3. This constitutes the static world of the simulated environment. Environmental effects such a rain, wind, or fog are not taken into account. Located within the static world is the simulated vehicle depicted in the lower center.

A. Operating System

Figure 4 shows a simplified class diagram of the operating system. Its user interface (UI) is implemented using Unity's UI Toolkit. The entry point is the singleton `OperatingSystem` which possesses the capability to instantiate new applications via its application programming interface (API):

Depending on the generic type parameter, the singleton dynamically constructs an appropriate application instance, injecting the `arguments` parameter passed into `OpenWindow<TApplication>(...)`, using C# reflection. Following the setup of the application, it is embedded into a new `Window` instance, and a `TaskbarItem` object is created. The latter two components serve distinct roles:

- **The `TaskbarItem` Class:** When clicked via the mouse, a taskbar item toggles the visibility of the associated window and causes it to fire an `OnWindowEvent`, transmitting the appropriate `WindowEvent` value.
- **The `Window` Class:** Windows are the containers for started applications. They can be minimized, maximized, closed, resized, and moved freely within the limits of the desktop environment.

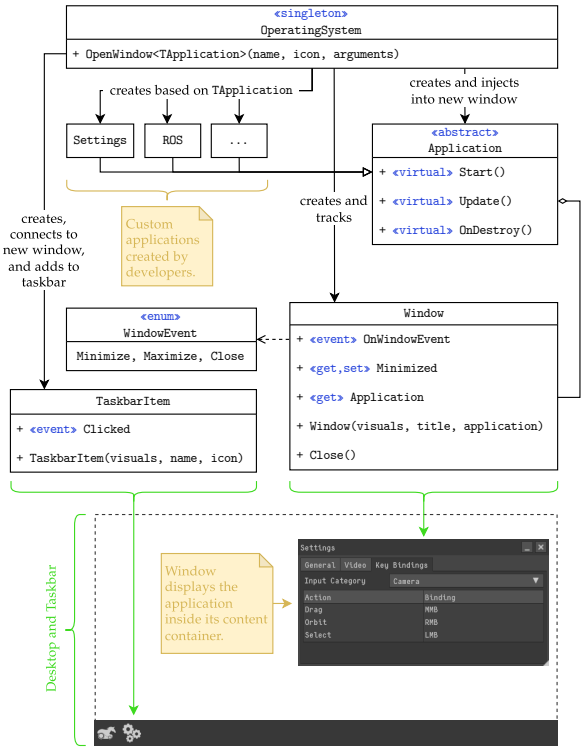

Figure 4. Simplified class diagram of the operating system.

B. Vehicle Simulation

The simulated vehicle is integrated into Unity's physics system. Its sensor sensor and actuator suite encompasses the sensors (camera, IMU, and wheel speed sensors) and the actuators (steering, motor, and wheels).

The camera was implemented using a Unity camera component with asynchronous GPU read-back to mitigate frame drops. To read back the depth texture, a custom shader pass, separating the depth element from the camera render into a separate texture, is executed prior. The color and depth frames are consequently published via the ROS-TCP-Connector at a rate of 30 Hz.

The IMU's acceleration is measured via Unity's `Rigidbody.GetPointVelocity(...)` API and the orientation is retrieved through the GameObject's transform component. Subsequently, the information is published on the ROS network at a rate of 70 Hz.

Similarly, the wheel speed sensors also use the same rigidbody API to retrieve the velocity at the respective wheels and publish it with a frequency of 100 Hz.

The simulated vehicle uses front-wheel steering with an Ackermann geometry. There is no dedicated motor simulation. Instead, the torque is applied directly to a custom wheel component that applies the forces shown in Figure 5. The implementation of the wheel component is largely based on [12]. The suspension force F_S (seen on the left) emulates the suspension of

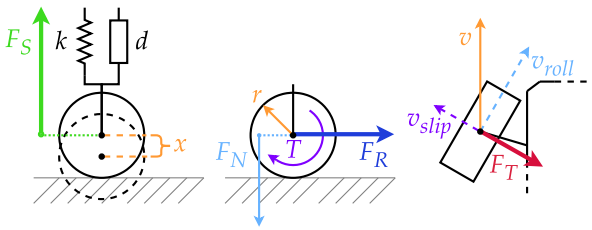

Figure 5. The wheel forces from left to right: Suspension force F_S, Rolling force F_R, Turning Force F_T

the wheel and is modeled as a spring-damper system by

$$F_S = k \cdot x - \dot{x} \cdot d \tag{1}$$

whereas x refers to the spring's displacement from rest, and k and d the spring and damper stiffnesses, respectively. Depicted in the center, the rolling force F_R converts the applied torque into a force through the radius of the wheel. Additionally, friction is taken into account by applying a force proportional to the normal force and directed against the rolling direction. c_{rr} in the equation 2 refers to the coefficient of rolling resistance, and the normal force F_N can be computed using equation 1.

$$F_R = \frac{T}{r} - \text{sign}(v_{roll}) \cdot F_N \cdot c_{rr} \tag{2}$$

While not implemented, one could adaptively adjust c_{rr} based on the contact surface to simulate various surface conditions.

Lastly, the turning force F_T originates from the fact that a wheel prefers to rotate around its mounting axle. Any movement parallel to this axle results in the wheel slipping or scraping along the surface and is therefore opposed. Equation 3 models this behaviour by applying a force opposite to the slip velocity of the wheel.

$$F_T = -m \cdot \frac{v_{slip}}{t} \tag{3}$$

One critical aspect is to ensure that the aforementioned equations are calculated with sufficient frequency. If the forces are computed too sporadically, the wheel simulation breaks down due to instability. The computational frequency is largely governed by Unity's physics time step, which has been set to 200 Hz.

C. Simulator-Aware ROS Nodes

One of the objectives of the simulator is to test the software in its unaltered state. However, achieving this objective encounters limitations, particularly regarding features that interface with real-world hardware embedded within the vehicle such as sensors and actuators. In light of this, it becomes clear that the environment must incorporate mechanisms that enable ROS nodes to detect the simulator.

Table I
MECHANISMS FOR SIMULATOR-AWARENESS.

Language	Mechanism

CMake	`if(DEFINED SIMULATOR)`
	`  # ...`
	`endif()`

C/C++	`#ifdef SIMULATOR`
	`  // ...`
	`#endif`

Python	`import rospy`
	`name = "SIMULATOR"`
	`if rospy.has_param(name):`
	`  # ...`

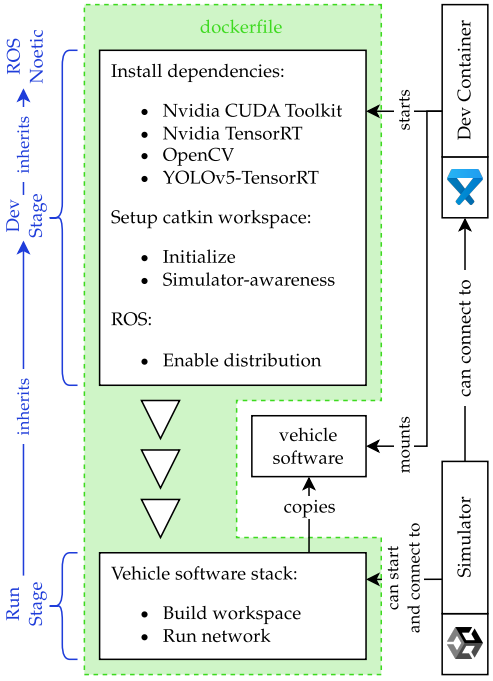

Figure 6. The multi-stage `dockerfile` and its dependents.

Table I presents the mechanism implemented for each language. The CMake and C/C++ mechanisms are enabled by passing the appropriate CMake arguments via `catkin config --cmake-args` during setup of the catkin workspace. Python utilizes the ROS parameter server to make nodes simulator-aware.

D. Docker Container

The Docker container is set up using the multi-stage `dockerfile` seen in figure 6. As visible, there are two stages:

1) **Dev Stage:** Based on ROS' official Noetic image, various dependencies are installed, most of them required for the image processing logic of the vehicle. Additionally, the catkin workspace is set up, and the mechanisms for simulator-awareness are enforced here.
2) **Run Stage:** This stage inherits its contents from the dev stage. Its main purpose is to copy the vehicle software, build it, and run the resulting ROS network.

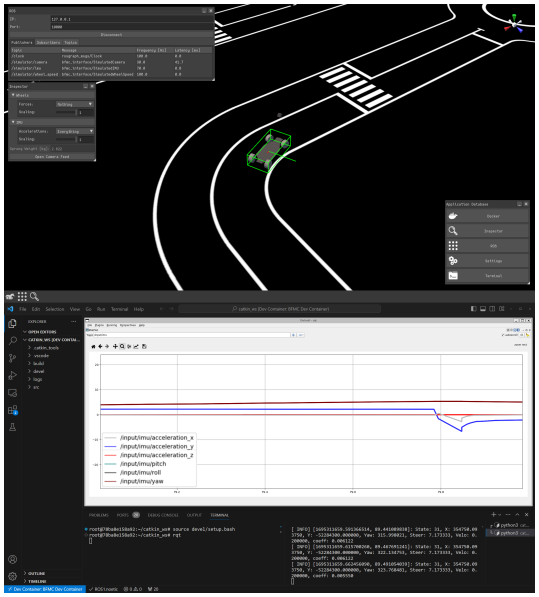

Figure 7. The development, simulation, and validation environment.

VII. RESULTS

Figure 7 shows the development, simulation and validation environment in which the existing ROS toolkit is used to analyze the IMU messages published.

The upper half of the figure is taken up by the simulator itself. The lines originating from the center of the vehicle visualize the accelerations perceived by the vehicle. The two applications seen in the upper left-hand corner are, from top to bottom, ROS which manages the connection established over Unity's ROS-TCP-Connector, and the inspector which can show custom information about objects present in the simulated environment. The latter application currently displays information about the simulated vehicle. Lastly, the application in the bottom right-hand corner is used to launch any of the available applications implemented within the simulator's operating system.

The lower half of the figure depicts Visual Studio Code operating from within the docker container detailed in section VI. Embedded in the lower right-hand corner is a terminal that runs the vehicle software stack, and right next to it another terminal is used to run the `rqt` application whose GUI is placed in the center of the integrated development environment (IDE). `rqt` itself harnesses the `rqt_plot` plugin to visualize the individual IMU messages published by the simulator topic over time.

VIII. CONCLUSION

This paper detailed a comprehensive development, simulation, and validation environment for ROS-based autonomous driving algorithms.

Initially, the conceptual ideas behind the environment were presented. Through the use of Docker and OS-agnostic software, it could be demonstrated that a single unified architecture can support developers and researchers across multiple operating systems. The simulator was designed with extensibility, modularity, and ease of use in mind, aspects largely facilitated by the introduced operating system. Additionally, the custom vehicle simulation ensures complete control over the vehicle's driving dynamics.

Lastly, the development and validation qualities of the environment were presented, highlighting the synergy between the simulator and the existing ROS tools. Practical experience has shown that new users were able to familiarize themselves with the environment and be productive within days. Additionally, a noticeable speedup in development of the underlying vehicle software could be observed.

IX. Future Work

Although the development, simulation, and validation environment already accelerates the development of the underlying ROS software tremendously, the general workflow could be further improved by removing the need to recompile the ROS network after each modification. This could be accomplished by employing a scripting environment inside the simulator that harnesses the ROS-TCP-Connector to publish data directly into the ROS network, circumventing the need for a recompile.

References

[1] Robert Bosch SRL. *Welcome to the Challenge*. URL: https://boschfuturemobility.com/.

[2] Jakob Häringer and Noah Köhler. *Neuartiger Software-Stack für ein autonom fahrendes Fahrzeug*. University of Esslingen, Feb. 2023. URL: https://gitlab.hs-esslingen.de/itmoves-masters/BFMC/-/wikis/uploads/dc65f1260898899e479baa7f70c7398b/Bericht_Forschungprojekt_1_Koehler_Haeringer.pdf.

[3] Jack Collins, Shelvin Chand, Anthony Vanderkop, and David Howard. "A Review of Physics Simulators for Robotic Applications". In: *IEEE Access* 9 (2021), pp. 51416–51431. DOI: 10.1109/ACCESS.2021.3068769.

[4] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. "AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles". In: *Field and Service Robotics*. 2017. eprint: arXiv:1705.05065. URL: https://arxiv.org/abs/1705.05065.

[5] Microsoft AI & Research. *AirSim*. URL: https://github.com/microsoft/AirSim#readme (visited on 09/22/2023).

[6] N. Koenig and A. Howard. "Design and use paradigms for Gazebo, an open-source multi-robot simulator". In: *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*. Vol. 3. 2004, 2149–2154 vol.3. DOI: 10.1109/IROS.2004.1389727.

[7] Open Source Robotics Foundation. *Gazebo : Tutorial : Windows*. URL: https://classic.gazebosim.org/tutorials?tut=install_on_windows&cat=install (visited on 09/22/2023).

[8] E. Rohmer, S. P. N. Singh, and M. Freese. "CoppeliaSim (formerly V-REP): a Versatile and Scalable Robot Simulation Framework". In: *Proc. of The International Conference on Intelligent Robots and Systems (IROS)*. www.coppeliarobotics.com. 2013.

[9] NVIDIA. *NVIDIA Isaac Sim*. URL: https://developer.nvidia.com/isaac-sim (visited on 01/30/2025).

[10] Ahmed Hussein, Fernando García, and Cristina Olaverri-Monreal. "ROS and Unity Based Framework for Intelligent Vehicles Control and Simulation". In: *2018 IEEE International Conference on Vehicular Electronics and Safety (ICVES)*. 2018, pp. 1–6. DOI: 10.1109/ICVES.2018.8519522.

[11] Dr. Martin Bischoff. *Differences between Unity Robotics Hub and ROS*. URL: https://github.com/siemens/ros-sharp/wiki/Ext_RosSharp_RoboticsHub#differences-between-unity-robotics-hub-and-ros (visited on 08/18/2023).

[12] Toyful Games. *Making Custom Car Physics in Unity (for Very Very Valet)*. URL: https://www.youtube.com/watch?v=CdPYlj5uZeI (visited on 08/17/2023).

Constantin Blessing received his Bachelor of Engineering in Computer Engineering in 2020 at Esslingen University of Applied Sciences. Staying at the same university, he is currently pursuing his Master of Science in Applied Computer Science.

Prof. Dr.-Ing. Reiner Marchthaler is a professor of embedded systems with a specialty in autonomous systems at Esslingen University of Applied Sciences.