Leopold-Franzens Universität Innsbruck
Fakultät für Mathematik, Informatik und Physik

Institut für Informatik

Quality Engineering

Bachelorarbeit

zur Erreichung des akademischen Grades

Bachelor of Science

ROS-based simulation of Cyber-Physical System with Unreal Engine

von

Manuel Eiter
(Matr.-Nr.: 12113513)

Submission Date:
Supervisor: Dr. Michael Vierhauser
Philipp Zech, PhD

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht.
Ich erkläre mich mit der Archivierung der vorliegenden Bachelorarbeit einverstanden.

Ort und Datum: _____

Unterschrift: _____

Contents

List of Figures

Abstract

Cyber-Physical Systems are playing an increasingly important role in our modern lives, with applications ranging from autonomous vehicles such as drones and cars to Industry 4.0. As a result of this increased involvement, there is a need to ensure that Cyber-Physical Systems interact safely with humans and other systems. Traditionally, this has been done through simulation test runs. However, with increasingly complex tasks and applications, simulations of Cyber-Physical Systems are reaching the limits of what is possible and representable in standard simulators. Here, game engines could offer an alternative to conventional simulators. In order to evaluate this potential, a framework has been developed that allows for Robot Operating System (ROS) simulations within the Unreal game engine. This framework was then used to demonstrate the potential use cases of such alternatives in simulating complex environments or providing accurate visualisations for deep learning algorithms. However, it also highlighted the current inaccuracies of fine-grained simulations when using the ChaosPhysics engine.

1 Introduction

With advances in AI, and new developments in autonomous control and decision-making, the correct functioning of Cyber-Physical Systems (CPSs) is more important than ever. To reduce the risk of physical harm, such systems need to be tested and verified in a wide range of different scenarios to ensure they function correctly even in extreme situations [16]. Accurate simulations are therefore essential, as they play a crucial role in showcasing and testing these often highly complex systems in an environment where it imposes no risk to others or itself [43].

Robot Operating System (ROS) is an open-source robotics middleware suite and is the de facto standard for many robotic CPS applications. The range of ROS-based systems is wide, ranging from swarms of drones to industrial robotic arms. The current widely accepted standard simulation environment is Gazebo [11][37]. Gazebo is an open-source simulator with deep ROS integration, making it uniquely suited to ROS simulations. It is commonly used, at least to some degree, in all stages of a system's lifecycle — from development to testing and deployment.

Relying on a single simulator for the entire simulation process can be dangerous, especially when considering that no simulator can truly simulate real-world physics [20]. It is therefore important to be aware of the limitations of a particular simulator and what these limitations mean regarding the simulation results [1]. For example, the results from a simulator with poor fluid dynamic capabilities are useless when one wants to test marine robotics. For Gazebo, these limitations include physical anomalies, collision accuracy, and lack of graphical representation. Graphical inaccuracy in particular is a major problem when simulators are used to generate visual data for model training, as it is the case for self-driving cars or other Cyber-Physical devices that incorporate vision capability.

Finding other possible and suitable simulation platforms that can counteract some limitations and problems encountered by Gazebo is therefore a crucial task. There are already decent alternatives to simulating with Gazebo, but nearly all of them lack at least one major aspect [9][42][3]. The intention of this thesis is to look beyond the common ROS simulators. This work aims at creating a simulation framework that allows the automatic simulation of ROS-based systems within the Unreal Engine. The idea is to take advantage of the game engine's functionality, including its physics and collision simulation capabilities, graphical accuracy, and further benefits from the surrounding ecosystem (flesh simulation, 3D terrain, and object scans, …). The created tools provide basic functionality to test if Unreal is suitable as simulator and lay a ground for further research in utilizing Unreal for simulation and testing on more general CPSs.

In summary, the aim of this thesis is to create a framework that allows the ROS-based simulation of CPSs within the Unreal Engine. The framework should fulfill the requirements below and showcase the possibility of simulating within Unreal Engine:

(i) Unreal Engine 5 (as game engine)

(ii) ROS2 humble (most recent LTS)

(iii) automatic simulation/configuration

(iv) logging capabilities

The follow-up Chapter 2 includes a short overview of already existing simulators and showcases their requirements. It gives a brief introduction to ROS and its successor ROS2, and finishes off with the possibilities for integrating ROS into Unreal and what is used in this framework. Chapter 3 then focuses on the overall structure of the framework, highlighting key components and giving a high-level introduction that is needed when working with the plugin. More details about the inner workings and possible improvements and extensions to the plugin can be found in Chapter 4. An evaluation of findings found during the development and testing of the plugin, as well as other key findings when working with Unreal Engine 5 and ROS2, are discussed in Chapter 5. Finally, Chapter 6 gives an outlook on the future of simulation in Unreal. Descriptions of how to set up this framework with Unreal can be found in the Appendix (Chapter 7).

2 ROS and Unreal Engine

This section is intended as a brief introduction to the world of the Robot Operating System (ROS), robot simulation and the Unreal Engine. It starts with what ROS is and why it is widely used. Then continues with the basics of Cyber-Physical System simulation, why simulations are needed, and what important role and requirements they have in specific robotic domains. Finally, an introduction is given to the Unreal Game Engine, why it might be suitable as a simulation environment, and what related work already exists in using game engines as simulators.

2.1 ROS

The Robot Operating System (ROS) is a software development kit for robotic applications. The project was started at Stanford University and popularized via the Willow Garage incubator [27]. It is established as a collection of different utilities and algorithms that allow for easier and faster development of robotic systems by enabling code re-usability. One of the core principles of the ROS project is that researchers and engineers should not have to reinvent the wheel every time they start a new robotic project. ROS therefore follows the UNIX design goal to 'make each program do one thing well' allowing for a high modality of robotic systems [17].

Despite its name, ROS is not really an operating system, but rather a middleware suite that sits between the actual operating system, such as Linux, macOS, or Windows and the applications that run on top of it. The ROS ecosystem can be split in three parts:

(i) Middleware: responsible for the communication between the different components. Abstracting away simulation details and providing a standard communication interface (as shown in Figure 2.1)

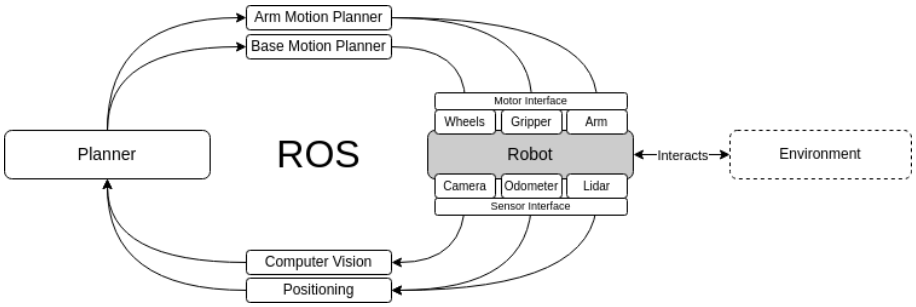

Figure 2.1: Simple architecture of a robot with different actuators and sensors.

(ii) Algorithms: already existing, established, and tested algorithms for common applications like perception or planning

(iii) Developer tools: configuration tools, launchers, simulators, and more.

Over the years, not only the platform ROS provided grew, but also the requirements posted by the industry increased. It was immediately clear that the initial structure of ROS is not capable of keeping up, as it became harder to patch new features like real-time capabilities or security into the system [17].

ROS2

ROS2 [17] is a complete rework of the robot operating system with focus on the previously mentioned features ROS1 lacks. ROS2 is not backwards compatible with ROS1. ROS1 packages will therefore usually not work under ROS2.

The most obvious differences between the two versions are the changes made to the middleware. ROS2 no longer follows a master-slave architecture with a central node (ROS master) and no longer relies on TCP/UDP for communication. It still uses the publisher/-subscriber architecture and uses the data distribution service (DDS) now, as it allows for real-time applications, higher efficiency, scalability, and quality of service. Further, DDS removed the single-point-of-failure ROS1 systems suffered from.

Another noteworthy major change is that ROS is no longer provided via two separate libraries. ROS 1 is available as library for C++[1] and for Python[2]. With ROS2, there is only one C base library (rcl). Other libraries build on top of it making it possible to offer support for other languages like Java or C#, with the benefit, that new functionality only needs to be added to one core library [19][17].

2.2 ROS Simulation Enviroments

Accurate simulation environments are one of the most important tools in robotic research and development. Simulation environments reduce cost, shorten development cycles, and allow for testing without the risk of harm to humans, the robot itself, or other objects [41]. Because of that, different simulation solutions both commercial and open source have emerged, covering different scopes and providing different features. In this section we describe already existing simulation solutions for ROS, what they are capable of and what limitations they have.

The scope of this section is to give a short introduction to what simulation environments already exist and what they are capable of in general. For a better understanding on how broad the requirements of simulators are, older but still used simulation environments are also included.

Player/Stage

Player/Stage project provides multiple tools for simulating multi-robot environments. The project is provided open source.

[1] roscpp
[2] rospy

Player is a TCP/IP socket-based device server which allows multiple devices to connect and exchange messages. Player is therefore used to control robotic sensors and actuators by creating a simple interface that is similar to other UNIX-like device servers. The use of a TCP/IP interface allows Player to be independent of programming languages, as almost every language nowadays has socket support. Due to the socket abstraction, it is also possible that the robotic devices are location-independent [12].
Stage is an open-source 2D robotics simulator, capable of providing virtual worlds in which mobile robots and sensors can be deployed. It uses low-fidelity simulation models in exchange for fast and cheap simulation, and scales linear when a large population of systems is deployed. Stage supports a range of sensors and actuators including sonars, scanning laser rangefinders and different robot bases [12]. By default, Stage runs in real-time, even with many robots in use [36].

Matlab with Simulink

By using the Robotics or a similar Toolbox, the already powerful Matlab environment can be extended by common features of robotics like kinematics, dynamics, and trajectory generation [5].

With the help of the ROS Data Analyser [33], ROS-related data can be visualized and interpreted. The following Viewers are supported (i) Image Viewer, enables loading of Image or CompressedImage message types (ii) Point Cloud Viewer, visualizes PointCloud2 message types (iii) Laser Scan Viewer, shows LaserScan massages (iv) Odometry Viewer (v) XY Plot Viewer (vi) Time Plot Viewer (vii) Message Viewer.

Webots

The Webots project was founded in 1996 and became open source in the second half of 2018. It allows the creation of highly customisable 3D worlds with adjustable physical properties such as friction and mass for simulated objects. There is a large collection of pre-defined sensors and actuators in Webots, but there is also the option to import custom models.

As a physics engine, Webots uses a fork of ODE (Open Dynamic Engine). In comparison to CoppeliaSim and Gazebo, Webots does not have the possibility of changing the physics engine.

CoppeliaSim (formerly V-REP)

Coppelia is a commercial robotic simulator (free education version) developed and maintained by Coppelia Robotics. Together with Gazebo, it is one of the best-known and rated simulation environments [13]. It supports 7 different languages, has a rich collection of robots and objects out of the box and is capable of changing the underlying physical engine [21].

Coppelia offers a wide range of complex features including forward and inverse kinematics, sensor simulation, path and motion planning, collision and distance calculation, and many more, which can be easily integrated thanks to its powerful APIs and plugin interfaces [3][4].

Feature	Webots [42]	CoppeliaSim [3]	Gazebo [11]
Supported Platforms	Linux, macOS, Windows	Linux, macOS, Windows	Linux
Supported Languages	Python, C++, Java	Python, C++, Java, Lua	Python, C++
Licensing	Open-source	Free for education	Open-source
Physics Engine	ODE	ODE, Bullet, Vortex, Newton, MuJoCo	ODE, Bullet, DART, Simbody
Rendering Engine	WREN (OpenGL)	OpenGL based	OGRE [10]
Plugin Support	Limited	Yes	Yes
Render Quality	★★★☆☆	★★★★☆	★★★☆☆
Complex Environments	★★★☆☆	★★★☆☆	★★★★☆
Community Size	★★★☆☆	★★★☆☆	★★★★★
Extensibility	★★★☆☆	★★★★☆	★★★★☆
ROS Integration	★★★☆☆	★★★★☆	★★★★★
Hardware Intensive	★★☆☆☆	★★★☆☆	★★★☆☆
Ease of Installation	★★★★★	★★★★☆	★★★☆☆

Table 2.2: Characteristics of different simulators [22].

Gazebo

For simulations with ROS, Gazebo is currently the most popular simulation solution. Gazebo is open source and has libraries and tools for most use cases, including ground-based vehicles or robotic arms. One of the main strengths of Gazebo is its broad support of advanced physics engines such as ODE, Bullet, or DART. These physics engines allow accurate simulation of collisions, friction, and dynamics even in complex simulation environments. Accurate simulations make Gazebo well suited for testing robots.

Like CoppeliaSim, Gazebo can simulate a wide range of sensors, including LIDARs, cameras, and even GPS. The simulator is able to produce realistic sensor data, which helps to verify the correctness of algorithms before they are deployed. Coupled with its integration into ROS, this further enables in-depth testing of all parts of the robotic system [21].

2.2.1 Comparison

Table 2.2 summarises the main features of the previously mentioned ROS capable simulators. As representatives, the table includes Webots, CoppeliaSim and Gazebo, as they are among the most commonly used simulators for ROS applications.

All three simulators support the major operating systems and have support for at least two programming languages. They also all have a freely available version. Webots and Gazebo are open source, while CoppeliaSim is free for educational purposes only. In terms of physical simulation capabilities, all three simulators have built-in physics engines, but only CoppeliaSim and Gazebo support switching between different engines. Gazebo has by far the largest community, with several active forums and community portals that facilitate problem-solving and development [22].

Gazebo has the best ROS and ROS2 integration as it is built in. Webots and CoppeliaSim require plugins to enable ROS support. Plugin interfaces are available for all three simula-

tors, making them extensible for new functions and features that are not natively available.

2.2.2 Shortcomings

Studies have shown that the physics engines used by current simulators all have certain limitations in terms of physical accuracy. ODE and DART fail to simulate slip under certain conditions [6]. Bullet's collision detection sometimes fails when using special collision objects or very large and very small object sizes at the same time [28]. MuJoCo shows relatively poor performance in terms of contact force and penetration [44]. DART's simulation pipeline is not suitable for simulation scenes with many objects [6]. It is therefore beneficial for a simulator to have the ability to switch between physics engines, as there is no clear winner and each engine has its strengths according to its design and optimisation [8]. Most studies also conclude that physics engines designed specifically for game engines have the upper hand in terms of stability, collision detection and speed, at the cost of less accuracy [8][28][15].

As Table 2.2 suggests, the shortcomings of Webots, CoppeliaSim and Gazebo are their inability to provide high quality real-time rendering when simulating large environments. Complex simulations involving multiple actors and many modules push the simulators to their limits and lead to loss of either:

(i) Rendering quality: image quality, lighting aspects

(ii) Physical accuracy: friction, collision detection, accuracy, stability

(iii) Simulation performance: real-time capability, frames per second, delta time between physical steps

A major problem is that rendering quality, physical accuracy and simulation performance are in tension with each other. Attempts to increase one of these will result in a decrease in at least one of the others. That larger time steps have a negative impact on the accuracy of physics engines is nothing new [8]. In order to achieve high overall results, it is therefore necessary for simulators to provide optimised solutions so that no sacrifices have to be made.

Webots and CoppeliaSim both use a custom rendering pipeline based on OpenGL, while Gazebo uses OGRE (see Table 2.2). Although the OpenGL pipeline and OGRE allow for some tweaking and customisation, this requires massive additional work, as additional library loading and extensive shader customisation must be performed. Depending on the setup and version, Webots, CoppeliaSim and Gazebo therefore don't include advanced features like real-time lighting systems (e.g. global illumination), ray tracing or complex material systems (e.g. subsurface scattering or layered materials). Also simple, because the focus is on simulation accuracy rather than photorealistic renderings.

2.3 Robotic Simulation

Simulation is one of the biggest topics in robotics research today. Simulations are crucial for proving and demonstrating concepts, developing new algorithms and improving existing ones. Accurate simulators and physics engines are therefore essential, allowing for an easy

transition between development and production. To produce relevant results, simulators must meet the different requirements of the different domains in which robotics is applied. The most common and biggest robotic research domains are [2]:

- **Mobile ground robotics** — focuses mainly on autonomous ground vehicles. Active research includes navigation, control, perception and simultaneous localization, and mapping [2]. The main task is to move a robot in its environment. To do this, robots need to be able to determine their position (GPS) or their relative position to obstacles (LiDAR, camera, etc.). Additionally, they need to plan their path depending on accessibility, traffic and various other factors. As ground robotics is one of the largest fields, there are already many well-established simulators such as Gazebo, CoppeliaSim and CARLA [7].
- **Manipulation robotics** — mainly focuses on stationary robots that interact and manipulate their environment by grasping and moving objects. This field is all about robotic arms and grippers. It deals with (i) path planning and reachability of objects (ii) dynamic environments in which objects and obstacles might change their location (iii) collision detection and avoidance (iv) touching and sensing through the gripper. Simulators need to support forward and dynamic kinematics, and have physical accuracy regarding friction, grasping (force) and gravity.
- **Medical robotics** — require a high standard of verification. Coupled with the wide range of complex tasks, this means that there are no general-purpose simulators for medical robots, and there may not be any in the future. Medical simulations nearly always need to be fine-tuned to their task and will only work well in their designed domain.
- **Marine robotics** — comes in two categories, one for underwater vehicles and the other for surface vehicles. Although both categories have some similarities, such as the need for physical accuracy in fluid dynamics, there are also some differences, such as the need for pressure and stress simulation for underwater vehicles, and wave simulation for surface vehicles.
  When it comes to simulator availability, the situation is split in two. For underwater vehicles, there are a number of solutions, either as Gazebo extensions or as dedicated simulators. For surface simulation, however, the situation is different. Due to the difficulties in simulating the environment of waves, wind and currents, there is really only UWSim [24], a Gazebo extension, as an established solution.
- **Aerial robotics** — is largely focused on the research and development of unmanned aerial vehicles (UAVs). Modern simulators already allow for complex environments with accurate modelling of turbulence and fluid mechanics constraints.
  There are a number of simulators available, including Gazebo and AirSim [35]. AirSim is developed by Microsoft and is based on Unreal Engine.
- **Learning for robotics** — by utilizing machine learning is a data-intensive task. Gathering relevant and accurate data is the most essential part of machine learning. It is almost always impossible to collect the amount of data needed for model training in real world situations. When it comes to reinforced learning, it is impossible to train robots in the real world without imposing risks to others or themselves.

2.4 Unreal Engine

Unreal Engine is an established Game Engine developed by Epic Games. Its main focus is on providing a platform that allows game developers to bootstrap games by utilizing the features already provided by the engine. It provides a variety of features out of the box including physic simulation, particle effects, efficient rendering, advanced lighting (including ray-tracing) and other more game-related functionality like actor control, level loading or multiplayer hosting. All of that is provided with cross-platform compatibility.

Apart from game development, Unreal Engine has also made its way to the filming industry. This is due to its capabilities of creating and rendering accurate scenes, allowing for cheap movie shots without the need of building the sets in reality.

The combination of physics simulation and scene rendering makes Unreal Engine also interesting for simulation tasks. In fact, there are already a number of specialized simulators that use Unreal Engine under the hood. A prominent example is AirSim [35], which is developed by Microsoft. It is designed for autonomous vehicles like drones and cars, and uses Unreal Engine for its visuals and physics. AirSim is used for developing and testing autonomous systems. Another example is CARLA, an open-source simulator for autonomous driving research [7].

2.4.1 Simulations

With the new version 5, Epic Games introduced new major features for graphic rendering and physic simulations. The new features are intended to update the older version 4, bringing the latest research and technological changes to Unreal Engine.

The most relevant changes introduced, and relevant for this work, are the introduction of Chaos Physics as successor to PhysX physic-engine, the new Lumen lighting system and the new mesh system Nanite. These new systems allow for simulations with high-fidelity graphics.

Key benefits of simulating with Unreal Engine:

Graphical accuracy	Its advanced graphical capabilities allow the gathering of training data in different conditions (day or night, sunshine or rain, …)
Collision system **Destruction system**	This system enables an easy detection of collisions during simulation. Further, there is the option to simulate destruction caused by collisions.
Environment capabilities	The editor allows the creation of complex levels and environments in which robots can be tested. It is possible to load complete maps and geo-scans.

Complex assets — There are predefined assets provided, which range from simple doors to skeletal mesh bodies. The latter creates the possibility to simulate human-robot interaction.

2.4.2 Physics

The new Chaos Physics engine, although praised as a huge step-up to the long-existing PhysX engine, still has a lot of childhood weaknesses to overcome [14]. The biggest drawback, in direct comparison, is the lack of hardware support. PhysX is developed by NVIDIA and benefits from the dedicated hardware accelerators built into its graphic cards. Another big problem is that the software is new and is still undergoing active development. It therefore hasn't undergone extensive testing and with every new release, there is the danger of new bugs.

A big advantage of Chaos Physics is the possibility of running it async to the game engine. Running physics simulation on a separate thread allows for fixed-rate calculation, improving determinism of the simulations.

2.5 Integration of ROS and Unreal Engine

When it comes to Unreal Engine 4, then there are a handful of plugins for communicating with ROS:

Plugin	Branch	UE Version	ROS Version	Platform
rclUE [25]	master	4.27	ROS2 Foxy	Ubuntu 20.04
	devel	5.1	ROS2 Foxy	Ubuntu 20.04
ROSIntegration [18]	master	4.26	ROS1 & ROS2	Ubuntu & Windows
UROSBridge [39]	master	4.27	ROS1	Ubuntu & Windows

Table 2.4: Plugins that bridge ROS and UE communication

For Unreal Engine 5 and ROS2, the offerings are manageable. Only rclUE offers a `devel` branch that supports Unreal Engine 5 and ROS2. The catch is that rclUE only supports Linux, and Unreal Engine 5 is not running very stable under Linux.

This work therefore focuses on UROSBridge and making it compatible with Unreal Engine 5 and ROS2. The process of doing so involves stripping away features of the original codebase that are no longer needed and updating the Unreal API function calls where required. The main advantage of using UROSBridge is that the plugin is system-independent, allowing Unreal to run on both Windows and Linux.

There are already a handful of similar frameworks that run on Unreal Engine 4 and use ROS1. The intention is to choose one of the existing frameworks and make it Unreal Engine 5 and ROS2 compatible by incorporating the requirements mentioned above.

3 Architecture and Components

The introduced framework provides Unreal Engine with the necessary functionality to be integrated as a simulation solution into the ROS ecosystem, similar to other simulators like Gazebo or CoppeliaSim. A comparison of Figure 2.1 and Figure 3.1 illustrates how Unreal Engine can be used as a simulator for ROS applications. From Figure 3.1 we can see the required interface (ROSBridge), that allows us to pass messages in and out of Unreal. The figure also shows the logging and configuration features mentioned earlier. Below is a summary of all the features:

(i) Communication with ROS
(ii) Interface for robot controlling
(iii) Information gathering and forwarding to a logging interface
(iv) Robot model loading (SDF and URDF)
(v) Configuration loading from a JSON file (startup and simulation configuration)

In order to provide this features, two plugins were designed.

(i) `ROS2UE5` is the core plugin and is responsible for ROS communication, robot control, model loading and configuration import.
(ii) `SeqLog` is responsible for fetching the log information and passing it on to the seq data-sink, where it is stored and can be evaluated later on.

The `SeqLog` plugin will be built from the ground up. It is meant as a blueprint that one can copy and alter into a logging interface for a different data-sink.

The `ROS2UE5` plugin is the core plugin for this work, building on top of the existing UROS-Bridge [40] and URoboSim [39] modules developed by the Institute for Artificial Intelligence at the University of Bremen. These modules were designed for UE4 and ROS1. For the scope of this paper, they were updated to work with UE5 and ROS2. During the process of upgrading, modules that were not required for this work were removed. There

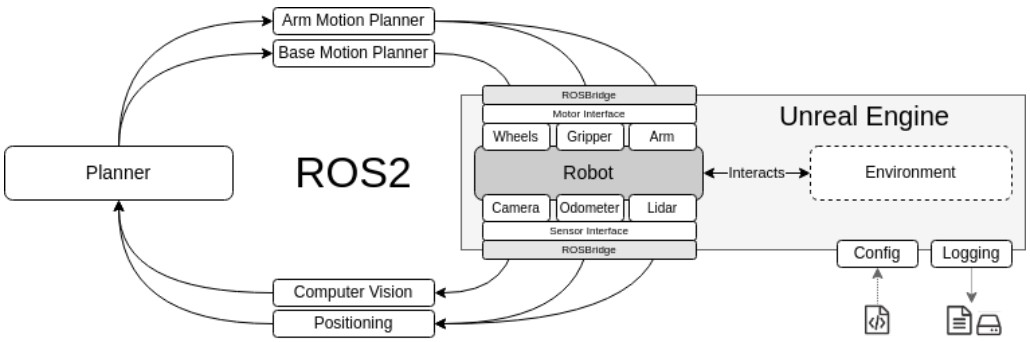

Figure 3.1: ROS architecture with Unreal Engine as simulator

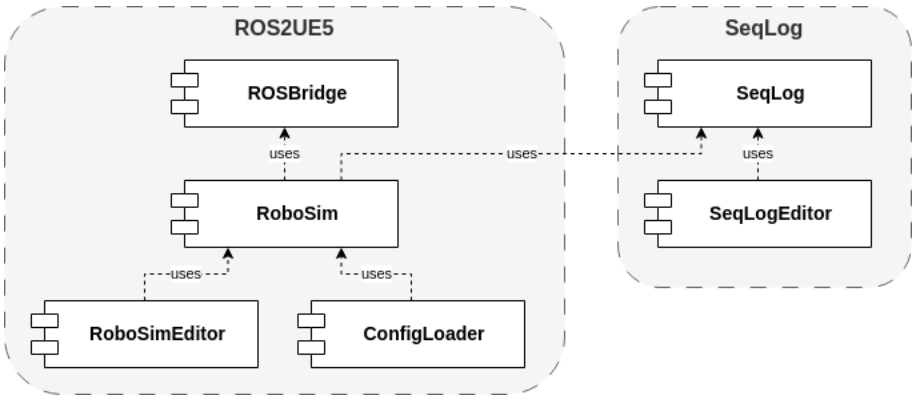

Figure 3.2: Components used in Unreal Engine

is a branch in the git repository for this work, kept for future reference, which contains all the methods and components that were removed and not updated to work with UE5 or ROS2.

The two introduced plugins, `ROS2UE5` and `SeqLog`, are internally made up of six separate components, which are depicted in Figure 3.2. The four components that make up the `ROS2UE5` plugin are required for establishing the communication channel with ROS and to handle the control of the Cyber-Physical System that is being simulated. The `SeqLog` plugin is responsible for forwarding log messages to the seq data-sink, where they are stored for later evaluation.

ROSBridge provides the interface to the `ros-bridge` websocket. It handles the connection and communication to the socket by providing the necessary subscriber, publisher and service interfaces. These interfaces link the simulated robots from within Unreal to the ROS ecosystem.

SeqLog gets the logging information out of Unreal and into the seq data-sink for later evaluation. Seq was chosen as it is easy to use and comes with a pre-configured docker container, which eases the reproducibility of the demo. The use of the `SeqLog` plugin is optional and can also be replaced by any similar plugin by altering it to the desired needs.

SeqLogEditor handles the configuration part of the `SeqLog` module. It uses the settings capabilities of the Unreal Editor to provide a way for centralized configuration.

RoboSimEditor holds the logic for the components that are relevant to the Unreal Editor. This includes the parsers which produce the robot description data assets and their corresponding factories. The data assets can then be spawned into scenes by calling the model builder of the `RoboSim` module.

RoboSim houses the core parts for simulating with Unreal Engine. It's the central point and coordinates the joint working with the other modules and components. RoboSim is responsible for the whole lifecycle of Cyber-Physical device simulation. It imports the data assets into the scene, handles the communication with the `ROSBridge` module and logs back relevant information. Multiple components are integrated into this module including the communication component, controller component and logger component.

ConfigLoader is responsible for loading and initializing the editor and scene upon startup. It is responsible for the configuration component, which is given a configuration file as a start parameter, containing environment information such as a map, robots, weather or general simulation information such as communication ports and logging settings.

The following sections cover the different components that make up the whole project in more detail, describing not only their general use, but also their composition and the possibility of further extensions and improvements.

3.1 ROSBridge

Simulating ROS-based CPSs inside Unreal requires the establishment of a communication channel that connects ROS and Unreal. A convenient way for any non-ROS system to establish a connection to it is by using the `rosbridge_suite` [32]. It includes a server that implements the rosbridge v2 protocol and provides a JSON interface to ROS. `rosbridge` supports web sockets or TCP connections for communication.

The overall communication flow is shown in Figure 3.3. As described earlier, the `rosbridge` splits the communication into two parts. The first one is ROS-related, dealing directly with the ROS environment, consisting of nodes, topics, and services. And the second is Unreal-related, receiving and sending messages to the websocket and calling the corresponding callback functions inside Unreal.

Upon closer inspection, one can see, that all the communication work inside Unreal is handled by the ROSBridgeHandler. Without going into too much detail (more details can be found in the implementation section of `ROSBridge` Section 4.1.1): The core task of the `ROSBridgeHandler` is to spawn a runnable instance on a second thread. This thread runs independently of the main Unreal thread, and receives and transmits information. The `ROSBridgeHandler` hereby runs through all subscribers, publishers and services, checks if messages have arrived or need to be delivered, and handles them accordingly. For getting messages into Unreal, callback functions are used. For getting messages out of Unreal, messages are pushed into a delivery queue, which is constantly emptied — also by the ROSBridgeHandler.

Message Passing via `rosbridge`

As mentioned previously, `rosbridge` is a ROS meta package that enables communication to ROS via a web socket, using JSON format. It is available in different versions with compatibility for different clients. The rosbridge version compatible with humble is delivered via the `rosbridge_suite` package, which is open source, running under the BSD-3-Clause licence.

Messages in `rosbridge` are JSON objects where a “op” field is present. This field indicates what operation the message is and how this message should be treated when received. The possible operations that come with the rosbridge v2 protocol are available in its documentation [31].

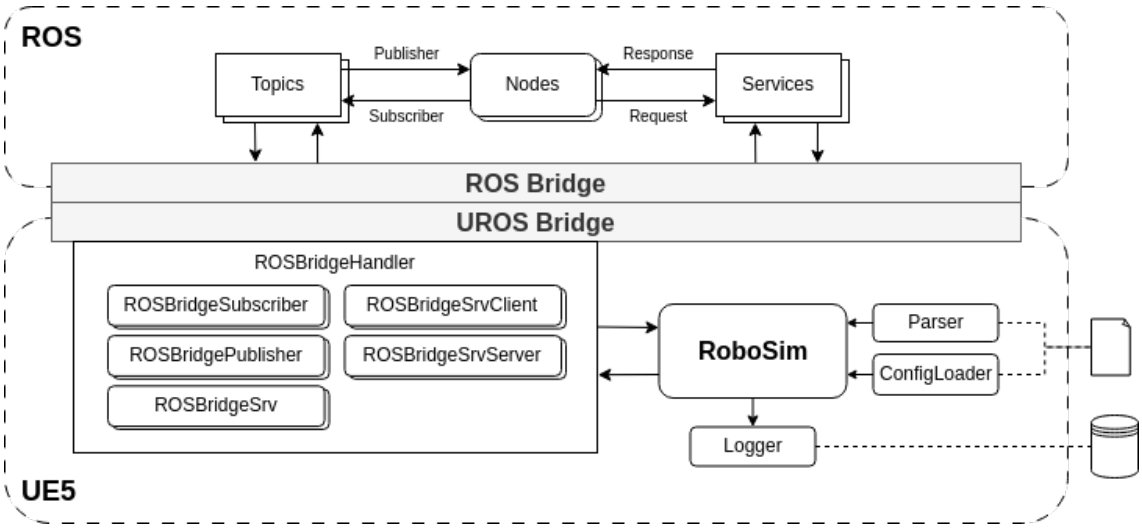

Figure 3.3: Information flow and Component diagram

Sending Messages

Before publishing to a topic, this topic first needs to be created and advertised to `rosbridge` . To do that, an `advertise` operation is sent to `rosbridge`. This message contains the name and type of the topic. Upon receiving a `advertise` message, the rosbridge checks if the topic already exists and if the given type is valid. If no topic with the given name exists and the type is valid, the topic is created.

For sending a message to a topic, a `publish` message with topic type and message is sent to the rosbridge websocket. The message body is in JSON format and must conform with the topic type. Valid messages are published by the rosbridge in the given topic and are from there on present in the ROS environment.

When shutting down the application, previously advertised topics need to be `unadvertise`, otherwise the `rosbridge` is unable to determine if a topic should still be advertised or not. When rosbridge receives an `unadvertise` message, it checks if there are other clients that still advertise the topic. And if not, it will stop advertising it to the ROS environment.

Receiving Messages

In order to receive messages from a specific topic, `rosbridge` must be informed. This is done using the `subscribe` operation. If multiple components subscribe to the same topic, an ID specifier distinguishes them. Otherwise, the `rosbridge` will not be able to correctly forward messages at the desired rate and will not be able to correctly handle unsubscribes.

`rosbridge` stores new messages inside a queue and forwards them at the provided throttle rate. The default throttle rate is 0 which is necessary to utilize the real-time capabilities of ROS2.

Upon shutdown, previously subscribed topics are unsubscribed. When multiple subscriptions to the same component exist, the ID specifier is provided to only unsubscribe the corresponding subscription.

3.2 SeqLog & SeqLogEditor

The ability to collect and output results is crucial for simulators, but for different projects, different data is of value. In order to keep this demonstration simple, the event scope of the default logger is kept at a minimum and only includes collision detection (`CollisionLogger`) and state output (`JointLogger`). However, it is possible to extend the logging mechanism to include advanced features such as relative position, velocity, force, sensor data, or camera feeds.

The logging mechanism is also intended to be independent of the data-sink and storage. Although the project only includes a 'SeqLog', other logging systems can be easily integrated by creating a custom output device.

Output Device

The default output device for the demo project is provided as separate plugin 'SeqLog'. This plugin can also be used to craft a custom interface to different logging systems and endpoints. To get full compatibility, the `ULoggerComponent::Log` method needs to be updated accordingly.

Logger

Loggers have two main purposes. First, they check if certain events have occurred, and second, they extract data that should be forwarded to a log. Every device that should be tested needs to have a `LoggerComponent`. This component can register multiple loggers that are ticked during simulation runs. The logger types can be added to the system as required.

3.3 RoboSimEditor

For use inside Unreal, robotic models need to be stored in a way that can later on be converted into 3D assets, that can be spawned in the scene. To provide a convenient way of using existing robotic models, the project implements an SDF and a URDF parser. These two are the most common robotic description formats.

To get the 3D models of a Cyber-physical device into Unreal, we first need to import the SDF or URDF file. This creates a `RDDataAsset` inside the content browser of the Unreal Editor. When `RDDataAsset` is spawned into the scene, it gets passed to the model builder, which creates and loads the meshes, joints, and links according to the data asset.

In order to get a correct `RDDataAsset` during import, we must ensure, that the meshes of the device are present in the FBX format used by Unreal (Section 3.3). `RDDataAsset` holds all the relevant information needed for a robotic model to be spawned into the world.

SDF

Simulation Description Format (SDF) is an XML Format that is used to describe objects and environments for robotic simulators. SDF is pretty powerful, as it can not only be used to describe robotic models but also worlds, physical properties within worlds, lighting, and actors.

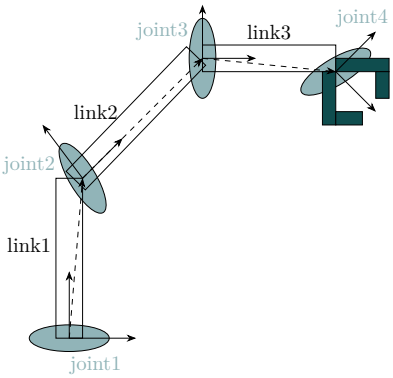

Figure 3.4: coordinate frames — URDF

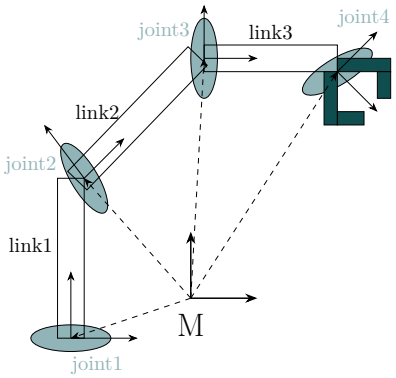

Figure 3.5: coordinate frames — SDF

For the scope of this work, only the model element of the SDFormat is used. The model element is capable of describing a whole robot or other physical objects. Although SDF also supports world and physic description, they are not well suited for use with Unreal Engine, as their attributes are primary tailored for use with Gazebo and the physics engines it can run. Furthermore, not even all attributes and elements that are described by the specification for use within the model element are supported by the developed parser, as some of them are very exotic and not needed for this project. An overview of the supported elements and attributes can be found in the later sections and in Figure 3.6.

To import SDF robotic models, the plugin provides a parser that reads the XML format and converts the different elements into the corresponding robot description components inside Unreal. The version supported by the parser is the currently latest version of SDF, version 1.11. As a base for the SDF-parser, the existing `URoboSim::USDFParser` was updated. The old `USDFParser` was written in UE4 and only supported SDF version 1.2. For this project, it therefore not only was migrated to UE5, but also upgraded to the new version in order to accommodate the new conventions of SDF. An example of such a change is the new convention for transformation relations between parent and child frames (changed with version 1.5). Other now commonly used features like the `relative_to` attribute are now supported and integrated into the reworked parser. The full list of changes that were applied to the `SDFParser` in order to be 1.11 ready is:

- support for the new `relative_to` attribute on the `pose` elements
- updating of changed or deprecated dependencies of the Unreal API
- support for `<use_parent_model_frame>` within the `link` element – as described in the SDF specification for version 1.5

When using SDF as a model format, it is important to know which version is being used and how this affects the behaviour of certain elements. Versions after 1.7 were mainly subject to additions to the specification. Earlier versions often saw breaking changes. The most essential breaking changes to be aware of are the introduction and later removal of the `use_parent_model_frame` element and the introduction of frames in the combination with the `relative_to` attribute [23].

URDF

The Unified Robot Description Format (URDF) is also an XML format that is commonly used for describing robotic models. Other than SDF, it is purely used for describing robotic models and therefore only supports `link` and `joint` elements, which is also shown in the direct comparison table in Figure 3.6.

Comparing just the XML format structure of URDF and the model part of SDF, it is immediately apparent that SDF mostly uses separate element tags to store values, whereas URDF is more focused on using attributes directly on the associated element.

Although SDF and URDF are quite similar, there are some fundamental differences. The biggest difference is in their definition of coordinate frames. In URDF the coordinate frames are defined recursively, as shown in Figure 3.4. The origin element within the joint element can be seen as the transformation between the parent link and the child link.
In SDF, the pose element within the link element is used to define the pose of the link relative to the model and joints are then positioned relative to their child link. This behaviour is shown in Figure 3.5.

Converting meshes

Natively, Unreal Engine only supports importing 3D meshes from FBX or OBJ files, but most existing robot descriptions have their collision and visual meshes in DAE format, which is not supported by Unreal. So, in order to get the meshes into Unreal, they first need to be converted into FBX. This is done by using the Blender 3D Editor, that can automatically convert all meshes by using a python script that utilizes the Blender Python API.

A problem of converting from DAE or OBJ to FBX is, that information about the original reference frame is lost. This is a problem when the reference frame was rotated. So for example, if the provided meshes don't have their z-axis facing upwards, their rotation within Unreal Engine is wrong. Users must ensure that the orientation of the meshes provided is correct, as there is no way of telling whether the flipping of the upward axis is desired and compensated for in the model origin element.

3.4 RoboSim

As the central part, the `RoboSim` component is responsible for putting all together. In total, the responsibility of the component can be split roughly in five tasks:

(i) Creating the model of the robot from the robot description

(ii) Spawning the model within the Unreal Engine environment

(iii) Providing callback functions for the ROS communication

(iv) Controlling the joints of the robot

(v) Gathering and forwarding log in formation to the logging interface

URDF

```
<robot name>
  <link name>
    <inertial>
      <origin xyz rpy>
      <mass value>
      <inertia ixx ixy ixz iyy iyz izz>
    <visual name>
      <origin xyz rpy>
      <geometry>
        <box size>
        or <cylinder radius length>
        or <sphere radius>
        or <mesh filename>
      <material name>
        <color rgba>
        <texture filename>
    <collision name>
      <origin xyz rpy>
      <geometry>
        <box size>
        or <cylinder radius length>
        or <sphere radius>
        or <mesh filename>
  <joint name type>
    <origin xyz rpy>
    <parent link>
    <child link>
```

SDF

```
<sdf version>
  <model name>
    <frame name attached_to>
        <pose ...>
    <pose ...>
    <link name>
        <pose ...>
        <inertial>
          <mass>
          <density>
          <pose ...>
          <inertia>
            <ixx>
            <...>
        <collision name>
          <pose ...>
          <geometry>
        <visual name>
          <pose ...>
          <geometry>
        <sensor name>
          <pose ...>
    <joint name type>
      <parent>
      <child>
      <axis>
      <pose ...>
```

Figure 3.6: Nodes and attributes as defined in the standards.

Actor spawning

Although much of the model loading is already done in the `RoboSimEditor` component, the parsed joints and links still need to be loaded and connected. The end product of the build step is, therefore, an actor that the Unreal Engine can control and simulate. As part of the actor creation process, all joints and links are created and the corresponding meshes and collision boxes are loaded and generated. The interfaces required for ROS communication, joint control and logging are also instantiated and assigned to the actor.

The actor spawning is heavily dependent on the correct working of the parsing and import of the `RoboSimEditor`. The two components are linked together via the `RDDataAsset` which is created by the `RoboSimEditor` and consumed by the `RoboSim` component.

ROS Communication

The use of the `ROSBridgeHandler` (as briefly described in Section 3.1) requires the use of callback functions to allow asynchronous communication with ROS. These callback functions are close to the actor, as they need to be able to call the joint controllers and update the joint state. Furthermore, in order to publish information to ROS, the communication interface must be able to immediately receive changes. All of this not only applies to joints, but also to other actors (e.g. sensors or wheels) that need to send or receive messages to ROS.

The whole process of communication at class level is described in much more detail in

Section 4.1, Section 4.4.3 and Figure 4.2.

Logging

The process of passing logging information to the `SeqLog` component is a simple call to the appropriate method. Alternatively, the internal Unreal Engine logging mechanism can be used. This allows logging messages to be sent using the `UE_LOG`. The advantage of this is a higher decoupling of the `SeqLog` component, as it only has to listen for messages on the log stream addressed to it. The disadvantage is that using the `UE_LOG` is less performant than a direct call, and Unreal log messages can only be strings. All information would have to be converted into a json string and parsed back into an object on the receiving end.

3.5 ConfigLoader

The option to start Unreal by providing a configuration file is intended to abstract some of Unreal's complexity away from the user. The combination of setup script and configuration file should be able to create a system capable of bootstrapping a standard simulation environment. This should ease the use for all users that want to check if simulating with Unreal is viable for them.

As several configuration files are used to standardize work between different applications and components, they should meet the following requirements: (i) *readable* – elements and attributes must be understandable within the context (ii) *clear* – a set value should produce the expected result (iii) *editable* – it should be easy to change settings (iv) *simple* – as the scope of the work is to provide a proof if simulations with Unreal Engine are feasible, the configuration should be kept at a minimum, only providing parameters of interest.

Configuration File

The configuration file is meant as an easy way to set the different parameters and options that can be configured within Unreal Engine. It is not intended to allow all possible configurations to be changed, but rather to make a trade-off between what is commonly needed and used in simulation scenarios and what is not. There are just too many parameters that can be changed in Unreal, which would make the configuration file cluttered and hard to understand.

JSON is used as the format for the configuration file. Supported parameters are displayed in Table 3.2. The table also provides a brief description of what the parameters and attributes are for, and what their value formats are. Most of the entries can be left out, reverting them back to their default values. An example configuration with all default values can be found in the appendix (Section 7.3). It is further possible to retrieve a configuration interactively by running the `setup_config` script.

Currently, launching with configuration file only allows the spawning of robotic models. Other objects or elements the robot should interact with need to be included in the map that is loaded. Furthermore, it is not possible to configure components other than subscribers, controllers, or loggers to the robot.

environment.map	The name of the UE level map. This map must already exist in the Unreal project. The default map is the empty map.
robots[].robot	Can either be the name of the imported data asset (as it can be found in the content browser of Unreal) or the path to an SDF or URDF file.
robots[].subscribers[]	Defines the subscribers that are added to the `SubscriberComponent`. Subscribers can also be added by hand later on.
robots[].controllers[]	Defines the controllers that are added to the `ControllerComponent`. Controllers can also be added by hand later on.
ros.bridge.ip	IP-address of the corresponding ROS bridge a connection should be established to.
ros.bridge.port	Port of the corresponding ROS bridge, a connection should be established to.
logging.destination.type	The type of the logger used. Currently, only 'Seq' is possible.

Table 3.2: Selection of the most essential configurations supported

Simulation Runner

The simulation runner is a simple tool, that allows automatic runs of simulations can be performed. The core of the tool is the `sim_runner` and the config creator `auto_sim_config`. The `sim_runner` gets one or more of the previously mentioned configuration files, their respective ROS launch scripts, and how many iterations one wants to simulate as input and runs them automatically.

As depicted in Figure 3.7, it is only possible to run a simulation via providing a `ueScenario`. If someone wants to run more complex simulations with the sim-runner, it is recommended to not spawn the models via the configuration file, but to specify everything within Unreal beforehand and then only launch the map.

Configuration within Unreal

All options that can be set via the configuration file can also be changed directly inside the Unreal Editor. For experienced Unreal users, it is possible to tweak different settings and parameters inside Unreal to change the behaviour of simulation physics, speed, and accuracy. It is also possible to change the limits (torque, force, joint constraint, …) of the robotic models, interaction between the robot and the environment, and many more things that can be expected from a game engine.

```
"restartBridge": false,
"ueProject": "RoboDemo/RoboDemo.uproject",
"simulations": [
    {
        "iterations": 3,
        "name": "MoveIt2 Pick and Place Demo",
        "ueScenario": "config.json",
        "ros2Launch": "pick_place_demo.launch.py",
        "ros2Pkg": "moveit2_tutorials",
        "timeout": 10,
        "maxSimTime": 25
    }
]
```

Figure 3.7: Example configuration for the `sim_runner`

Setup Script

The setup script is intended as an easy and fast way to get ready for simulating with Unreal. It is a command line tool that goes through all steps needed, creating the correct mesh files and configurations along the way.

The steps automated by the script are:

(i) Reading the robotic models

(ii) Converting meshes into the supported format

(iii) Generating the configuration files from user-dialogue

(iv) Starting the simulation / Unreal Editor

(v) Importing models

(vi) Placing the models in the scene

Running the setup results in a simulation environment with the defined robotic models imported and the configurations set. The final step is simply to press play.

4 Implementation

This chapter discusses the components and their functionality in more detail, by looking at how they are implemented on a class level. The focus here is only on the Unreal Engine plugin and its modules, not on the ROS or Python scripts that are included in the tool's repository [30]. More details about the Python scripts (conversion, setup and startup tool) can be found online in the repository [30] or in the appendix (Section 7.2). The structure of this section follows the same as the previous Chapter 3.

The class diagram in Figure 4.1 shows all six components (SeqLog and SeqLogEditor grouped as one). Within the components, all 'relevant' classes and their connections are drawn. Note that some classes, such as all the `DataAsset` classes for the three plugin components and their corresponding factories and builders, are omitted. This is because they only show secondary ways in which the `URROSCommunicationComponent`, `URLoggerComponent` and `URControllerComponent` can be created, and would therefore only make the diagram harder to read. Within every component are light grey boxes grouping together certain classes. These groups are only drawn to get a better understanding of which larger task a class belongs to.

From the class diagram in Figure 4.1, we can already see the main functions that each component provides and how they are in relation to one another.

ROSBridge: Provides two main functions, the 'ROSBridgeHandler' and the 'Websocket'. The 'Websocket' allows connecting to the socket created by the rosbridge module within the ROS environment. The 'ROSBridgeHandler' uses this socket to exchange messages with ROS. The communication is done in a separate thread (`FROSBridgeHandlerRunnable`) so there is no performance impact.

SeqLog & SeqLodEditor: The 'Settings' part is needed to extend the Unreal Engines settings with fields to specify the logging server and port. The logging itself is done by the 'Logger' which provides the two methods of direct logging or using the `UE_LOG` (`FSeqLogOutputDevice`).

RoboSimEditor: Consists of the 'Factory/Parser' part and the 'RobotDescription DTOs'. The 'Factory/Parser' is able to import SDF and URDF files and convert them into a `URDDataAsset`. This data asset can then be used by the `URModelFactory` and `URModelBuilder` to spawn the actor in the scene.

RoboSim: The classes in the 'ROSCommunication' group are responsible for passing messages to the actors and use the 'ROSBridgeHandler' to do this. The classes within the 'Controller' box update the actors, while the 'Logger' part is responsible for passing messages to the `SeqLog` component. The job of the 'Factories/Builders' is to create the 'Actor/Model' within the Unreal Engine environment.

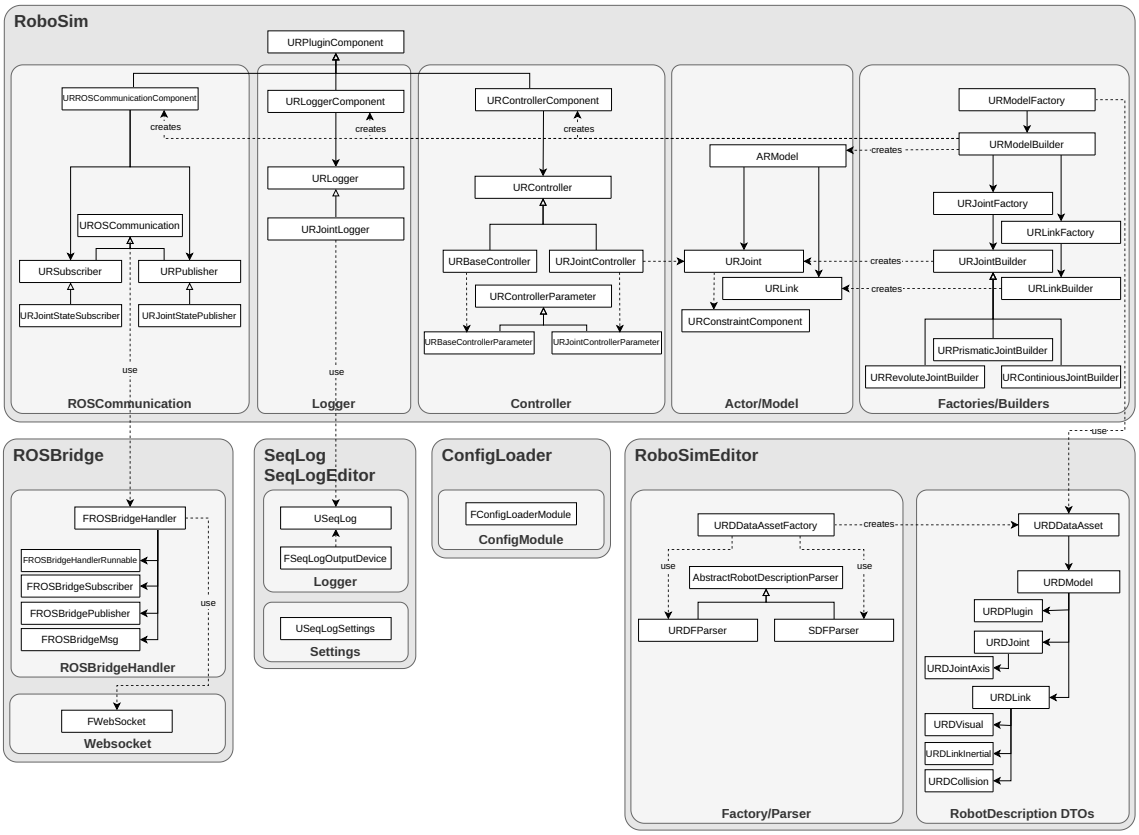

Figure 4.1: Class diagram

ConfigLoader: Basically consists of a class that runs on editor startup and checks if a configuration file has been passed. If so, the appropriate configurations are set.

4.1 ROSBridge

The communication between the ROS ecosystem and the running simulation within the Unreal Engine can be split into four parts:

(i) ROS-Bridge, which is part of the ROS environment

(ii) `FWebSocket`, which extends the default websocket with further features

(iii) `FROSBridgeHandler` and `FROSBridgeHandlerRunnable`, which are responsible for responding to callback or message changes

(iv) `RROSCommunication`, which is finally responsible for updates regarding the running simulation

The inter working of three of them is shown in Figure 4.2. Not shown is the part regarding `RROSCommunication` class, as this is part of the `RoboSim` component (Section 4.4). It uses the `PublishMsg` and `Subscriber::Callback` methods as an interface to publish or receive messages.

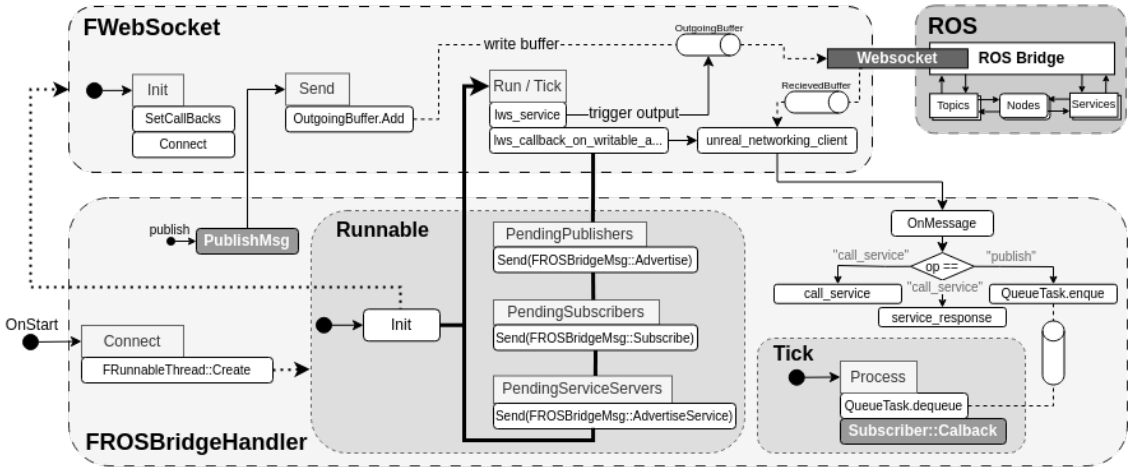

Figure 4.2: Information flow and Component diagram

4.1.1 UROSBridge

The UROSBridge module handles the message exchange from the ROS-Bridge and Unreal Engine. The following sections document how the communication between ROS and Unreal Engine is implemented and how it is used inside the plugin.

At the lowest level, the communication is handled by libwebsockets. A pure C library that provides lightweight clients and servers for different web protocols, among them the *WebSocket* protocol. Websockets are used as it allows full-duplex communication over a single TCP connection. This makes real-time communication between clients and servers possible. This is crucial, as real-time communication allows using the full potential of ROS without the worry for timing or latency problems.

The function calls to libwebsockets are abstracted away through the `FWebSocket` class. It implements much easier to handle methods like `SetReceiveCallBack` and `Send`. The `FROSBridgeHandler` is using these methods later on for providing publish and subscribe functionality, as well as services to Unreal Engine.

OnStart

In order to be able to use the ROSBridge communication stack, the selected game instance inside Unreal has to extend the `UROSBridgeGameInstance`.

When starting the simulation, Unreal calls the `OnStart` method of the selected Game-Instance. This results in the creation of `FROSBridgeHandler` and a call to its `Connect` method. With this call, the runnable is created and started on a separate thread, the callback functions get set, and the web socket establishes a connection to the set address and port — as set in the configuration.

Publish

Whenever a publisher publishes a message, a call to the `FROSBridgeHandler::PublishMsg` method is triggered. This method converts the `FROSBridgeMsg` into a `FString` and forwards

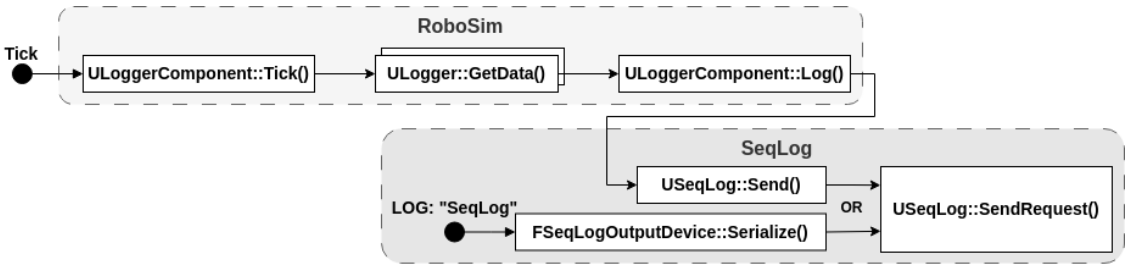

Figure 4.3: Logger data gathering & SeqLog forwarding

it to the `FWebSocket::Send` method, where it is written into the `OutgoingBuffer`. The `OutgoingBuffer` is consumed through the `Run`-loop that ticks the `FWebSocket` on every pass.

The consumption of the buffer does not depend on ticking within Unreal, as the `Run`-loop is part of the `FROSBridgeHandlerRunnable`, that runs on the separate thread.

Subscribe

Incoming messages are picked up by libwebsocket and trigger a callback to the `unreal_networking_client` method. This method checks what kind of callback is triggered. If the callback is a `LWS_CALLBACK_CLIENT_RECEIVE` (for receiving), then the `OnMessage` method, which is set as received-callback method during initialization, is called with the data from the `ReceivedBuffer` as parameter.

The `OnMessage` method further checks what kind of message it got. When it turns out to be a "publish" message, a `FProcessTask` consisting of the subscriber, topic, and message is created and added to the processing queue.

The queue is used to sync incoming messages with the ticking interval of Unreal Engine. This happens as the `FROSBridgeHandler::Process` method is called via a tick method. Within the `Process` method, the task is dequeued and the callback to the subscriber is triggered with the message as parameter.

4.2 SeqLog & SeqLogEditor

The logging system enables an easy way of gathering and forwarding any data that is produced during simulation. Focus is hereby placed on the flexibility and independence between data collection and data storage. To achieve this, the logging process is split in two.

The data gathering part is directly integrated into the `RoboSim` module and therefore allows the collection of (i) incoming data – coming from ROSBridge (ii) outgoing data – messages sent out (iii) and simulation data – collision states, velocities, forces, and everything else that exists inside Unreal during simulation and is not already sent out via the ROSBridge.

The logging part needs to be provided by an extra plugin – as demonstrated with the 'SeqLog' plugin. The plugin must provide some sort of send method for the `RLoggerComponent` to pass data to. Alternatively, the plugin can be a `FOutputDevice`, in which case the `RLoggerComponent` can use Unreal's internal logging system to pass data.

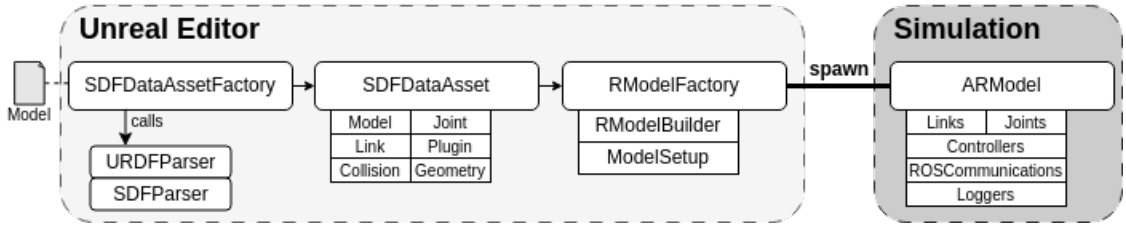

Figure 4.4: Information flow and Component diagram

4.3 RoboSimEditor

In order to import ".urdf" and ".sdf" files into the Unreal Engine Editor, two things must be implemented. First, a parser that is able to traverse the given file format, read the nodes and attributes, and create the corresponding Unreal entities like meshes, data assets, joints, etc. is needed. Second, a factory is needed, that tells Unreal where and how to use this parser and enables the import dialogue for the given file suffix.

4.3.1 USDFParser & UURDFParser

The parser is a plain object that implements the different methods for converting the nodes described in Figure 3.6 into a `URDDataAsset`. The figure is not showing all possible elements, only the ones that are in use or might be interesting for future work. For simplification purposes, only the supported attributes are listed. SDF is further missing some attributes and the inertia elements are collapsed into "...". A full list of all elements, with description, default value and behaviour can be found in the SDF specification [34] or in the URDF specification [38]

To standardize the working of models within Unreal Engine, URDF and SDF files get parsed into the same `DataAsset` structure. This simplifies the structure of the Plugin as only one `DataAssetFactory` and `DataAssetBuilder` is needed. It further ensures that all features that get provided by other Plugins work regardless of the original modelling format. One thing to remember is that although the modelling formats of SDF and URDF look similar, there are some things that behave differently, especially when older versions of SDF are used. For example, the behaviour of the pose element in SDF prior to 1.5 was that all poses were relative to the model reference frame. Newer versions on the other hand behave similarly to URDF where a relative reference frame can be specified (see 3.3 and 3.3). These things are accounted for within the parser, before the translations and rotations are set on the `DataAsset`.

To make the `URDDataAssetFactory` able to import custom files like ".urdf" or ".sdf", the `UFactory` interface needs to be extended. Only two methods of `UFactory` need to be overridden to allow for importing. The first one is the `FactoryCanImport` method, which receives the filename to import and returns a boolean, whether this factory can import the file type. The second method is `FactoryCreateFile`, which is responsible to create a new object by importing the given file name (for this task the two parsers get called).

4.4 RoboSim

As discussed in Section 3.4, the `RoboSim` component is responsible for orchestrating model spawning, communication, simulation updating and logging. The tasks can be divided into pre-simulation and in-simulation. Model spawning is the only task that is done before the simulation starts and is handled by the `URModelFactory` and `URModelBuilder` classes. Communication, updating and logging are all done while the simulation is running and are handled by the `RROSCommunication`, `RController` and `RLogger` classes. Since the functionality provided by these three is needed per actor, they are wrapped and assigned to an actor via their component classes (see Figure 4.1).

4.4.1 URModelFactory

The final step for getting the robot model in Unreal is spawning the corresponding `RDModel` actor in the scene. This is done by `URModelFactory` and `RModelBuilder`. They read the `DataAsset` and try to build the robot according to this blueprint.

The `URModelFactory` extends the `UActorFactory` interface. This interface is used by Unreal to check which factory is capable of spawning an actor given a certain input object. If a factory is found, the `SpawnActor` method of the factory is called. It is responsible for spawning the actor in the scene.

When a `DataAsset` is dragged from the content browser into the scene, Unreal triggers all registered `CanCreateActorFrom` methods to check if the factory is capable of creating an actor from the given `URDDataAsset` class.

`URDDataAsset` can have multiple robotic models defined within them. The `SpawnActor` method iterates over all models, spawns simple actors in the world, and then calls the `URModelBuilder::Load` function with the data and the `ARModel` actor as parameters. The further setup and creation of links, joints, constraints, and components are done by the `RModelBuilder`.

4.4.2 RModelBuilder

The `RModelBuilder` gets as input the `URDModel` description and produces a spawned `ARModel` actor as output. Within the `Load` method, the following tasks are performed.

`LoadLinks`	Link loading is done by the `RLinkFactory`. It creates a new `URLink` for each link in the model and sets their corresponding pose, collisions, visuals and other parameters.
`LoadJoints`	Joint loading is done by the `RJointFactory`. It creates `URJoint` as well as their constraint components and gets the parent and child links assigned.
`BuildKinematicTree`	With this method, the position of the joint constraint is set depending on the position of its parent link. Also, the collision is attached to the parent component and the joint is finally added to the parent link.

`SetupPlugins`	Adds additional plugins to the model (out of this work's scope)
`SetupROS`	Enables ROS communication capability for the model. The component allows adding publishers, subscribers, or services to this model that can be used for controlling the joints or publishing the state. (Section 4.1)
`SetupControl`	Registers the control component for the model. The control component allows the registration of controllers such as the joint controller, which is needed to define the motion (kinematic or dynamic) of the model's joints.
`SetupLogger`	Puts a logger component to the root of the model. With the logger component, different loggers can be added that read certain values and forward them to an external data-sink. (Section 4.2)

4.4.3 RROSCommunication

The last part missing in the ROS communication chain is the one, that is responsible for updating the running simulation. The `RROSCommunication` class inside the `RoboSim` component is responsible for that. To be more precise, its job is to link the `FROSBridgeHandler` and the simulated actors, by providing the final subscriber, publisher, and services. To provide functionality on a per simulated object level, a `RROSCommunicationComponent` needs to be attached to every robot. The `RROSCommunicationComponent` provides an interface for further attaching subscribers, publishers, and services. On simulation start, all subscribers, publishers, and services are registered in the `FROSBridgeHandler` and triggered when a related event occurs.

The demo subscriber provided in the project is the `URJointStateSubscriber`. It extends the `URSubscriber` base class by implementing the required `void URSubscriber::CreateSubscriber()` method. This method is responsible for creating and wrapping the callback function inside a class that is compatible with `FROSBRidgeSubscriber`. As mentioned earlier, callback functions are called by the `FROSBridgeHandler` when new messages arrive for the subscribed topic. For the callback methods themselves, there are no restrictions on what they can do. They can be extended and adapted to all the needs and possibilities that the Unreal API offers.

In terms of functionality, the previously mentioned rule regarding the possibilities of the Unreal API also applies for publishers. Custom publishers must extend the `URPublisher` class and implement the `void URPublisher::Publish()` method. This method is then again wrapped inside a `FROSBridgePublisher` class, in order to be compatible with the `FROSBridgeHandler`. Other than that, it is again up to the developer on what topic and message type a publisher works with.

4.4.4 RController

The controller class is responsible for setting up the correct simulation behaviour when the Unreal Engine starts a simulation run. In the case of the concrete `RJointController`, the initialisation phase sets up the joints for either kinematic or dynamic simulation.

During the run, the controller is responsible for triggering the update of the corresponding object in the scene. For the `RJointController` this would mean updating the joint position or velocity to the desired state. This is simply done by calling the appropriate method on the `URJoint`, which internally triggers a call to an Unreal joint constraint, triggering updates in the physic and simulation scene.

4.4.5 URLoggerComponent & URLogger

Every Cyber-Physical device that should gather information must have a `RLoggerComponent` attached to it. It is hereby irrelevant if the device is controlled by ROS, modelled by Unreal, or even controlled through direct user input.

Different logger types are then assigned to a `RLoggerComponent`. On every 'tick' during simulation, the logger component checks all its loggers for updates. The updates are then collected, and corresponding logging messages are created.

Logger types are created by extending the `URLogger` object. They must implement the `TSharedPtr<FJsonObject> URLogger::getData(const float& InDeltaTime)` method, which is used by the `RLoggerComponent` to retrieve data. The benefit of this abstraction between logger types and the logging component is that logger types can handle asynchronous events, such as retrieving the `OnComponentBeginOverlap` event.

4.5 ConfigLoader

In order to allow an automated start of the simulation, the framework supports the loading of certain properties via a configuration file. This initial load supports everything needed to load the scene and the robot. It is also possible to define other simulation settings such as the ROS bridge or the logging interface via a configuration file.

4.5.1 ConfigLoader

The startup loading of the configuration, which is only relevant when a configuration file is passed while launching Unreal from the command line, is handled mainly by the `FConfigLoaderModule::OnEditorInit` method. This method is registered by the module to be launched when the `FEditorDelegates::OnEditorBoot` event is triggered. Loading the configurations itself is straightforward, which only revolves around the process of parsing the provided JSON file and changing the settings as described earlier in Section 3.5.

4.5.2 URoboSimGameInstance

The game instance is responsible for bootstrapping everything needed for the simulation. It is triggered by the 'on start' event of Unreal. The main setup jobs are to create a new thread for the ROSBridge, trigger all registrations to the ROSBridgeHandler, handle the start of all controller components and initialize all other Unreal Engine properties.

In robotic simulation, it is common to not only simulate kinematic behaviour, but also dynamic behaviour where robots are driven/controlled by the forces or velocities they

apply to their joints.
Although not relevant and not directly used in this work, the **URControllerComponent** was kept and upgraded to UE5. This controller component is responsible for setting up the drive motors and other properties that are required when simulating dynamic behaviour.

5 Evaluation

This chapter summarises the final results of the framework, draws some comparisons between Unreal and traditional simulators, and what to look out for when simulating with Unreal Engine. For the comparison Table 2.2 is extended by a column for Unreal.

5.1 Framework

The final framework is best demonstrated by the demo project [26]. This demo includes all the features provided by the framework:

- Robot description import
- Logging System
- ROS communication channel
- Configurable automatic simulation runs

To set up the demo project the git repository [29] can be downloaded, and the README file provides instructions that guide users through the setup and installation of Unreal Engine 5, ROS2 humble, Blender (optional) and MoveIt2 (optional). Another repository is available for download, specifically for testing the import functionality individually by including already FBX converted meshes. More detailed instructions and descriptions on how to set up an Unreal Engine project using the framework's plugins can also be found in the Appendix 7.1.

5.2 Simulation with Unreal

The factor of how useful Unreal Engine is for simulations varies drastically from case to case. As discussed in Section 2.3, there are already successful examples of Cyber-Physical System simulations using Unreal Engine for simulation in the fields of mobile ground robotics [7] and aerial robotics [35]. This is not surprising, as driving and flying are features that have been supported by game engines for many years. Both examples use Unreal's graphical capabilities to gather information or test algorithms for self-driving vehicles. However, for simple simulations, the extra work required to set up a running simulation with Unreal may not be worth it.

The next two sections focus on the main advantages of simulating with Unreal: high fidelity graphics, environmental interaction and a large ecosystem. The last three sections discuss some of the problems and lessons learned while setting up the Pick and Place demo.

Figure 5.1: Example of a Franka Emika Panda robot in a visual complex scene

5.2.1 Visuals and Lighting

Visual accuracy can be of high value when one needs to gather a lot of data for machine learning, or quality imagery, in general, is needed. The rendering and lighting systems, provided by Unreal, create realistic-looking scenes and are easy to set up and change. This ease of change also applies to weather conditions and daytimes, making data collection in broad scenarios possible.

There are many examples where the capabilities to simulate different weather for the same scene is useful. The most prominent might be self-driving vehicles, as they must be tested in a variety of lightning and weather conditions to verify the correctness of their algorithms in all cases. Simulating is hereby practical, as it is hard to reliably find edge case conditions, like extreme snow storms or flash floods, in reality.

5.2.2 Complex Environments

Interaction with the environment is one of Unreal's greatest strengths. It benefits directly from the fact that game engines are all about actors (controlled by players) interacting with the environment. This can be utilized for testing, where instead of a player taking control, control is handed over to an algorithm.

A further benefit is the well established ecosystem surrounding game development. This includes not only the tools that can be used for modelling, but also the existing collection of physical assets, geoscans, city maps and other objects that have already been modelled and can be used seamlessly within Unreal Engine. This gives researchers the ability to test their systems in accurate replicas of reality. And the ability to share these scenarios on a common platform.

Unreal also scores, when simulations with dynamic changes in the environment are needed. Cars driving around a city or people moving through a warehouse are features, already included in the engine. This makes testing Cyber-Physical Systems that have to deal with human interaction or interruption easy. Allowing to check how the systems behave and if they correctly change their planned paths and avoid collisions.

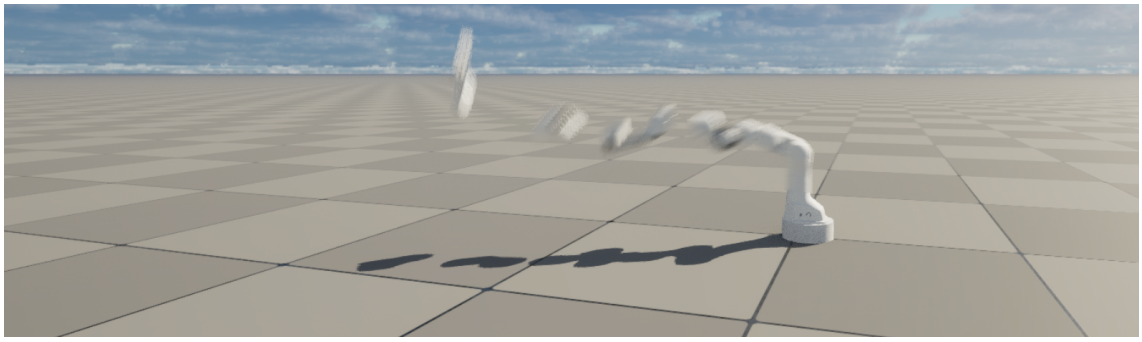

Figure 5.2: An extreme case of 'jumping', as it happens when physical simulation for joints are not disabled during kinematic simulation

5.2.3 Physical behavior

Physical simulations with Unreal Engine can be tricky. Due to it being a game engine, its focus is on performance optimized simulations. It prioritizes responsiveness and visual appeal over physical and real-world accuracy. The engine therefore struggles with long kinematic chains. This is especially a problem when larger forces are applied to end effectors, which causes the corresponding joint constraints inside Unreal to get out of their linear or angular limits. When these limits are reached, Unreal Engine forces them back into position, causing the joint to jump which can lead to unexpected jumping or even oscillation of the entire robot, as other joints might now be outside their limits.

5.2.4 Grasping

A further problem is the simulation of grasping objects, especially for solid objects. Unreal Engine has three ways to deal with collisions between two objects. The first one is "ignored", which is not suited for grasping simulation, as the objects just pass right through each other. The second one is "overlap", which is suitable for simulating the grasping process manually by, for example, creating a physical constraint between the finger and the object. And the third one is "block" which lets the build in physics engine handle and simulate the collision.

The problem with letting the physics engine simulate the collision is, that closing the fingers nearly always causes them to go into collision with the object on two (or more) sides. This causes the physics engine to produce counter forces, which causes the object to "glitch" out of the gripper most of the time.

For solid objects, the only real possibility is therefore to handle the grasping manually. This can be done by catching the overlap event and then joining the object and robot together via a physical constraint. The benefit of this is, that we are still able to simulate the interaction between finger and object and are even capable of specifying and simulating damping and friction between them.

Epic Games is currently developing Chaos Flesh, which allows for soft body simulation. Soft body simulation is used for a more realistic collision simulation, allowing an object to deform when force is applied. When completely introduced into Chaos Physics, this could

enable better simulation of grasping objects, but as of writing, Chaos Flesh is in an early beta version and not capable of simulating the needed high-resolution collision meshes for accurate grasping.

For the Pick and Place demo simulation, a simple physical constraint, which just holds the object in place, is used. There are no advanced physical simulations between object and gripper taking place.

5.2.5 Kinematics and Dynamics

Unreal Engine works great for pure kinematic simulations. For kinematic simulations, the physical simulation of the joints are disabled. This is necessary, as otherwise the earlier described jumping of the robot would inevitably occur. This jumping occurs for kinematic solutions, because the new position of each joint is directly set, leading to the break of the constraint limitations. Disabling the physical simulation of the robot does not influence the behaviour of other objects hitting it. The model in its entirety is still present inside the physics engine and causes collision and "block" events with other objects.

For dynamic simulations, the physic constraints get driven via angular motors. The constraints can herby be set to a target force and/or target velocity the joint should reach. Using dynamic control requires some fine-tuning and adjustments to reach satisfying results. Without fine-tuning, which also involves setting and checking the mass and center of mass for the links, controlling of the robot might not work as expected or may require forces that are different from those needed in reality. Results retrieved via dynamic simulations should therefore always be used with caution.

5.3 Comparison

To getter a general overview on how Unreal Engine compares to the traditional simulators, Table 5.2 extends the previously shown table by a further column.

Table 5.2 immediately shows that Unreal beats the other three in terms of render quality and ability to render large, complex environments. However, this is offset by the additional hardware required and the more difficult setup process. Generally speaking: Unreal Engine is the way to go when advanced simulations (graphical and environmental) are needed and the additional cost in hardware and human resources doesn't matter. If the goal of the simulations is just to demonstrate or prove a simple concept, the easier setup and lower requirements offered by Gazebo (or the others) pay off.

Another big feature, not shown in the table, is Unreal's ability to use AR. This can be used to better view the simulation or to directly simulate human interaction within the virtual environment. Without going into too much detail, all the features offered by the game engine that are not offered by the simulators can be seen as a kind of 'extra features' to consider when thinking about using Unreal, as there may be a valid use case within the Cyber-Physical domain for the given feature.

Feature	Webots [42]	CoppeliaSim [3]	Gazebo [11]	Unreal Engine
Supported Platforms	Linux, macOS, Windows	Linux, macOS, Windows	Linux	Linux, macOS, Windows
Supported Languages	Python, C++, Java	Python, C++, Java, Lua	Python, C++	C++
Licensing	Open-source	Free for educationl	Open-source	Editor is free
Physics Engine	ODE	ODE, Bullet, Vortex, Newton, MuJoCo	ODE, Bullet, DART, Simbody	Chaos
Rendering Engine	WREN (OpenGL)	OpenGL based	OGRE [10]	Custom
Plugin Support	Limited	Yes	Yes	Yes
Render Quality	★★★☆☆	★★★★☆	★★★☆☆	★★★★★
Complex Environments	★★★☆☆	★★★☆☆	★★★★☆	★★★★★
Community Size	★★★☆☆	★★★☆☆	★★★★★	★★★★★
Extensibility	★★★☆☆	★★★★☆	★★★★☆	★★★★★
ROS Integration	★★★☆☆	★★★★☆	★★★★★	
Hardware Intensive	★★☆☆☆	★★★☆☆	★★★☆☆	★★★★★
Ease of Installation	★★★★★	★★★★☆	★★★☆☆	★★☆☆☆

Table 5.2: Feature table, with a column for the Unreal Engine.

Unreal Engine — High		
#	GPU_{peak}	$CPU_{max/min}$
1	98%	48% - 35%
2	97%	50% - 36%
3	98%	48% - 36%
4	98%	55% - 36%
5	99%	49% - 36%

Unreal Engine — Medium		
#	GPU_{peak}	$CPU_{max/min}$
1	85%	41% - 30%
2	86%	47% - 36%
3	88%	48% - 32%
4	85%	49% - 32%
5	85%	39% - 30%

Unreal Engine — Low		
#	GPU_{peak}	$CPU_{max/min}$
1	70%	43% - 31%
2	67%	38% - 25%
3	70%	36% - 25%
4	69%	34% - 25%
5	71%	34% - 22%

Gazebo		
#	GPU_{peak}	$CPU_{max/min}$
1	48%	42% - 26%
2	37%	35% - 27%
3	41%	37% - 26%
4	33%	35% - 27%
5	28%	36% - 27%

Table 5.3: GPU and CPU load while running the pick and place simulation.

#	Unreal_{start}	Gazebo_{start}
1	6.44	3.12
2	6.06	3.27
3	6.04	3.03
4	6.22	3.41
5	6.79	3.26

Table 5.4: Start up time of Unreal and Gazebo

Hardware Load

As shown in Table 5.2, Unreal is capable of high quality rendering, but this comes at the cost of high resource intensity, requiring more memory and processing power. To get a better idea of how resource-intensive the Unreal Engine is, some simple benchmarks were run. The measurements Table 5.3 were taken while running the pick and place program (PandaGraspMap) at different rendering qualities as offered by the Unreal Engine (low: 999x362, medium: 1418x515, high: 1738x630). Each simulation run took about 15 seconds, and 5 runs per quality were performed. Within each quality, the runs were performed sequentially, i.e. the Unreal Engine was not restarted until after the quality had been changed. Furthermore, the viewport was not moved during the runs to avoid any additional rendering. The same simulation was run with Gazebo to get some comparative values. The AMD Adrenaline tool was used to obtain the values, from which only the peaks and lows are used for comparison. PC hardware specification: AMD Radeon RX 6600 (GPU) and AMD Ryzen 7 5800X 8-core processor (CPU).

The results show, that even the lowest quality requires more GPU power than Gazebo. Also, the result of the highest renderings might not be accurate, as the GPU seems to have reached its performance limit. As expected, CPU usage is not as affected by changes in rendering as the GPU. The AMD Adrenaline graphs also suggest that the start of each simulation run requires more CPU intensive work than when the simulation is running.

Another indication of the higher hardware requirements of the Unreal Engine can be seen by comparing the start-up times (see Table 5.4). The additional time is critical because the current implementation of the simulation runner (Section 3.5) starts a new Unreal Engine instance per simulation. The startup time could therefore be the biggest time-consuming part of running several small simulations.

5.4 Discussion

Considering game engines as simulator alternatives may have huge potential in certain scenarios, especially when high fidelity graphics and environment interaction are a primary concern. However, researchers must always weigh up the advantages and disadvantages before deciding which simulator to use.

Regarding the usefulness of Unreal Engine and especially the new version 5 of the game engine as a simulator, the results are mixed. On the one hand, the new rendering and lighting features are extremely powerful and especially useful when accurate images are

needed for training and testing systems or algorithms.

On the other hand, Epic Games seems to have taken a step backwards in terms of simulation accuracy by abandoning PhysX in favour of its own Chaos physics engine. But this may only be in the short term. In the long run, the move could pay off, especially as new features like soft-body physics and fluid dynamics are introduced. Furthermore, once the current bugs and inaccuracies (compared to PhysX) have been ironed out, the physics engine promises to be much more powerful, allowing for faster testing and simulation.

6 Outlook & Conclusion

The framework described in this thesis was created to demonstrate the potential of game engines for simulating Cyber-Physical Systems. The main findings are already covered in Chapter 5. This final section summarises some issues mentioned, what future work can be done and what the overall conclusion is regarding ROS-based simulation of Cyber-Physical Systems with Unreal Engine.

6.1 Future Work

There are many ways to improve the framework. Some improvements or additions that would increase its usefulness are:

(i) **Logging Capabilities** — The logging capabilities are currently minimal and only show how this feature can be used. To be really useful, a more accurate collision logger and more complex environment and interaction loggers can be implemented.

(ii) **Grabbing Control/Simulation** — One could also explore the possibilities of ChaosFlesh and how it can be used to simulate soft body grasping. But not only that, one could also explore the destruction functionality of the Unreal Engine, which can be used to simulate and test the behaviour of systems when they break something or have to handle something fragile given to them.

(iii) **Sensor Support** — For real interaction with the environment, the integration of sensor capabilities is inevitable. Further work could focus on creating a common sensor library that contains functionality for the most common sensor types.

(iv) **Simpler Simulation Automation** — The current approach to simulation is to launch multiple simulation runs sequentially. The simulation could benefit from starting multiple instances of the Unreal editor and running simulations in parallel.

6.2 Conclusion

Simulation with game engines may not be suitable for all tasks and applications. But there are some situations where their capabilities can be of great benefit. In particular, where there is a complex interaction between the system and the environment, researchers and engineers can benefit from the responsiveness of game engines. Choosing Unreal Engine for simulation tasks also gives you access to a wide range of other third-party assets. In particular, Epic Games own marketplace can be a boost for researchers, allowing them to benefit from terrain generators, NPC controllers and more.

Of course, using game engines isn't without its own challenges. As mentioned in Chapter 5, the current problems with the introduction of the new ChaosPhysic engine are particularly problematic. Although Epic Games has stated that the new engine is a huge improvement in terms of determinism and overall capabilities, the early stages of the engine show that it still has some catching up to do to reach the performance and accuracy levels of its predecessor, PhysX. At the same time, we should not forget the new features introduced with ChaosPhysics which, when they become stable in the future, will improve the possibilities for grabbing and particle/fluid simulations. In particular, soft-body simulations will offer new advantages in showing the effects of forces applied by robotic end effectors or collisions between certain devices. This allows researchers to assess the potential damage to Cyber-Physical Systems and test new fail-safe mechanisms without the risk of harming anyone in the evaluation process.

Finally, I would like to discuss the suitability of the Unreal Engine as a simulator. As discussed, there are already established and proven simulators for simulating ROS-based Cyber-Physical systems, and while they may not offer the same features and functionality as Unreal, they are much easier to set up and get started with. So the decision really comes down to whether the extra features of the Unreal Engine (high fidelity graphics, AR, etc.) are worth the extra resources that need to be invested to make everything run smoothly.

7 Appendix

This chapter provides instructions on how to set up the Unreal Engine plugins and run the ROS demo project.

The plugins and demo project mentioned in the sections below can be found on GitHub:

- **ROS2UE5** — `https://github.com/ManuETR/ROS2UE5`
- **ROS2UE5-tools** — `https://github.com/ManuETR/ROS2UE5-tools`
- **SeqLog** — `https://github.com/ManuETR/SeqLog`
- **RoboDemo** — `https://github.com/ManuETR/RoboDemo`

7.1 Setup

The Framework was developed and tested with ROS2-humble and Unreal Engine 5.3, running on a Windows system with WSL (Ubuntu).

For the demo project, the following dependencies are needed:

Unreal Engine	Although only tested with version 5.3 it is recommended to use the newest version to benefit from the newest Unreal features.
ROS2*	Install the latest LTS version to benefit from long term support. The Plugins are ROS2 version independent, as the communication is abstracted via the ROS bridge.
ROS Bridge*	Is required for communication between ROS2 and UE5.
Seq docker*	The Seq datalust docker image is the easiest way to set up Seq. Seq is used for logging and is only needed when working with the `SeqLog` plugin.
Blender(*)	If the `convert_to_fbx` tool is needed, it is necessary to install blender and its corresponding CLI/python interface.
Python3	For running the scripts inside the `ROS2UE5-tools` repository.

Further instructions can be found in the `ROS2UE5` readme – “Installation and Setup Guide”.

Setup Script

To automate this process, there is a `setup.sh` script in the ROS2UE5-tools repository which handles the installation of the parts marked with '*'. The script also clones all three plugin-repositories.

7.2 Demo Project

The demo project is an Unreal Engine 5.3 project. It is intended to showcase the features of the framework and comes with following contents:

- Panda and UR5 robot-arms already imported
- Space map with spawned panda arm
- Warehouse map with spawned panda arm, grasp component, and an object to grasp
- Demo assets to bootstrap other demo maps

7.2.1 Simulation

The provided simulation is intended to demonstrate a "full-run" of the system[1].
Before running the simulation:

(i) Install all dependencies listed in Section 7.1

(ii) Clone the `ROS2UE5-tools`

(iii) Clone the `RoboDemo` project

(iv) Make sure that the `ROS2UE5` and `SeqLog` plugins are inside the "Plugins" directory of the project

(v) Download the Seq docker container and make sure it is running

If everything is set up:

(i) Run the `sim_runner.py` script (inside `ROS2UE5-tools`)

(ii) Enter the path to the simulation config file[2].

(iii) Enter the path to your `UnrealEditor.exe`[3] or `UnrealEditor` binary.

After the two prompts, the `sim_runner.py` takes over. It launches the rosbridge server (if not launched), starts the UnrealEditor with the config and autostart arguments set, and launches the ROS program, as Figure 7.1 shows.

[1] But not all tools of the `ROS2UE5-tools` are used.
[2] Press enter when you want to run the default demonstration configuration.
[3] If you are using Windows and Unreal Engine 5.3 with default installation path, you can just press enter.

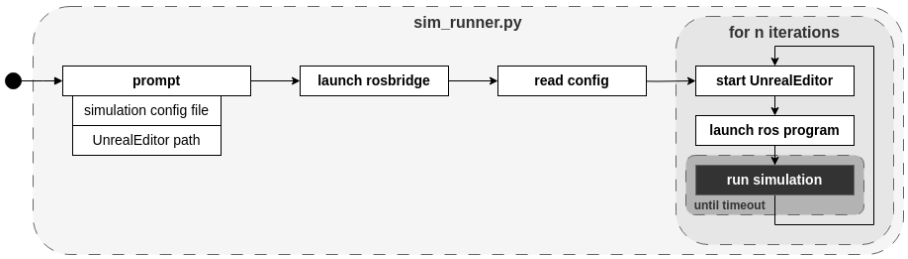

Figure 7.1: Steps performed by the `sim_runner.py`

7.3 Configuration files

There exist two types of configuration files. One is consumed by the UnrealEditor and is used to automate the scene setup — loading of robots, maps, loggers, logging destination settings, …. The other is used by the `sim_runner.py` script and allows for automated simulation runs by specifying the number of iterations, UE scene-config, ROS package to launch, ….

```
{
    "environment": {
        "map": "PandaGrapMap",
        "weather": "sunny",
        "time": "noon"
    },
    "robots": [],
    "ros": {
        "bridge": {
            "ip": "127.0.0.1",
            "port": 9090
        }
    },
    "logging": {
        "enable": false,
        "frequency": 1,
        "destination": {
            "type": "Seq",
            "endpoint": "http://localhost:5341"
        }
    }
}
```

Figure 7.2: Example Unreal Scene: `config_grasp.json`

```
{
    "restartBridge": false,
    "ueProject": "C:/Users/Manuel/Documents/Unreal Projects/RoboDemo/RoboDemo.
        uproject",
    "simulations": [
        {
            "iterations": 3,
            "name": "MoveIt2 Pick and Place Demo",
            "ueScenario": "\\\\wsl.localhost\\Ubuntu\\home\\manuel\\ROS2UE5-
                tools\\config_grap.json",
            "ros2Launch": "pick_place_demo.launch.py",
            "ros2Pkg": "moveit2_tutorials",
            "timeout": 8,
            "maxSimTime": 25
        }
    ]
}
```

Figure 7.3: Example Simulation Runner: `simulation_config.json`

Bibliography

[1] Agrawal, A., Zech, P. and Vierhauser, M. [2024], 'Coupled requirements-driven testing of cps: From simulation to reality', *arXiv preprint arXiv:2403.16287* .

[2] Collins, J., Chand, S., Vanderkop, A. and Howard, D. [2021], 'A review of physics simulators for robotic applications', *IEEE Access* **9**, 51416–51431.

[3] *Coppelia Robotics* [n.d.], `https://www.coppeliarobotics.com/`. Accessed 20-08-2024.

[4] *CoppeliaSim Introduction* [n.d.], `https://hades.mech.northwestern.edu/index.php/CoppeliaSim_Introduction`. Accessed 20-08-2024.

[5] Corke, P. I. [1996], 'A robotics toolbox for matlab', *IEEE Robotics & Automation Magazine* **3**(1), 24–32.

[6] Dongho, K. and Jemin, H. [n.d.], 'Simulation benchmark', `https://leggedrobotics.github.io/SimBenchmark/`. Accessed 2024-11-29.

[7] Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A. and Koltun, V. [2017], CARLA: An open urban driving simulator, *in* 'Proceedings of the 1st Annual Conference on Robot Learning', pp. 1–16.

[8] Erez, T., Tassa, Y. and Todorov, E. [2015], Simulation tools for model-based robotics: Comparison of bullet, havok, mujoco, ode and physx, *in* '2015 IEEE International Conference on Robotics and Automation (ICRA)', pp. 4397–4404.

[9] Farley, A., Wang, J. and Marshall, J. A. [2022], 'How to pick a mobile robot simulator: A quantitative comparison of coppeliasim, gazebo, morse and webots with a focus on accuracy of motion', *Simulation Modelling Practice and Theory* **120**, 102629.
URL: *https://www.sciencedirect.com/science/article/pii/S1569190X22001046*

[10] *Features* [n.d.], `https://www.ogre3d.org/about/features`. Accessed 2024-11-30.

[11] *Gazebo* [n.d.], `http://gazebosim.org/`. Accessed 07-10-2024.

[12] Gerkey, B., Vaughan, R. T., Howard, A. et al. [2003], The player/stage project: Tools for multi-robot and distributed sensor systems, *in* 'Proceedings of the 11th international conference on advanced robotics', Vol. 1, pp. 317–323.

[13] Ivaldi, S., Padois, V. and Nori, F. [2014], 'Tools for dynamics simulation of robots: a survey based on user feedback', *arXiv preprint arXiv:1402.7050* .

[14] Laaksonen, M. [n.d.], 'Unreal engine 5 compared to unreal engine 4', `https://www.linkedin.com/pulse/unreal-engine-5-compared-4-mika-mike-laaksonen`. last accessed: 11.09.2024.

[15] Liao, C., Wang, Y., Ding, X., Ren, Y., Duan, X. and He, J. [2023], 'Performance comparison of typical physics engines using robot models with multiple joints', *IEEE Robotics and Automation Letters* **8**(11), 7520–7526.

[16] Lyu, X., Ding, Y. and Yang, S.-H. [2019], 'Safety and security risk assessment in cyber-physical systems', *IET Cyber-Physical Systems: Theory & Applications* **4**(3), 221–232.
URL: *https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/iet-cps.2018.5068*

[17] Macenski, S., Foote, T., Gerkey, B., Lalancette, C. and Woodall, W. [2022], 'Robot operating system 2: Design, architecture, and uses in the wild', *Science Robotics* **7**(66), eabm6074.
URL: *https://www.science.org/doi/abs/10.1126/scirobotics.abm6074*

[18] Mania, P. and Beetz, M. [2019], A framework for self-training perceptual agents in simulated photorealistic environments, *in* 'International Conference on Robotics and Automation (ICRA)', Montreal, Canada.

[19] Maruyama, Y., Kato, S. and Azumi, T. [2016], Exploring the performance of ros2, *in* 'Proceedings of the 13th International Conference on Embedded Software', EMSOFT '16, Association for Computing Machinery, New York, NY, USA.
URL: *https://doi.org/10.1145/2968478.2968502*

[20] Mouret, J.-B. and Chatzilygeroudis, K. [2017], 20 years of reality gap: a few thoughts about simulators in evolutionary robotics, *in* 'Proceedings of the genetic and evolutionary computation conference companion', pp. 1121–1124.

[21] Nogueira, L. [2014], 'Comparative analysis between gazebo and v-rep robotic simulators', *Seminario Interno de Cognicao Artificial-SICA* **2014**(5), 2.

[22] Noori, F. M., Portugal, D., Rocha, R. P. and Couceiro, M. S. [2017], On 3d simulators for multi-robot systems in ros: Morse or gazebo?, *in* '2017 IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)', pp. 19–24.

[23] *Pose Frame Semantic* [n.d.], `http://sdformat.org/tutorials?tut=pose_frame_semantics&ver=1.5`. Accessed 08-08-2024.

[24] Prats, M., Perez, J., Fernández, J. J. and Sanz, P. J. [2012], An open source tool for simulation and supervision of underwater intervention missions, *in* '2012 IEEE/RSJ international conference on Intelligent Robots and Systems', IEEE, pp. 2577–2582.

[25] *rclUE* [n.d.], `https://github.com/rapyuta-robotics/rclUE/tree/devel`. last accessed: 12.09.2024.

[26] *RoboDemo* [n.d.], `https://github.com/ManuETR/RoboDemo`. last accessed: 12.11.2024.

[27] *Robot Operating System* [n.d.], `https://formant.io/resources/glossary/robot-operating-system/`. Accessed: 2024-07-22.

[28] Roennau, A., Sutter, F., Heppner, G., Oberlaender, J. and Dillmann, R. [2013], Evaluation of physics engines for robotic simulations with a special focus on the

dynamics of walking robots, *in* '2013 16th International Conference on Advanced Robotics (ICAR)', pp. 1–7.

[29] *ROS2UE5* [n.d.], `https://github.com/ManuETR/ROS2UE5`. last accessed: 12.11.2024.

[30] *ROS2UE5-tools* [n.d.], `https://github.com/ManuETR/ROS2UE5-tools`. last accessed: 12.11.2024.

[31] *ROSBridge Protocol* [n.d.], `https://github.com/RobotWebTools/rosbridge_suite/blob/ros2/ROSBRIDGE_PROTOCOL.md`. Accessed 11-08-2024.

[32] *rosbridgesuite* [n.d.], `https://github.com/RobotWebTools/rosbridge_suite`. last accessed: 13.09.2024.

[33] *ROS Data Analyzer* [n.d.], `https://de.mathworks.com/help/ros/ref/rosdataanalyzer-app.html`. Accessed 24-07-2024.

[34] *SDF Specification* [n.d.], `http://sdformat.org/spec?ver=1.11&elem=model`. Accessed 08-08-2024.

[35] Shah, S., Dey, D., Lovett, C. and Kapoor, A. [2018], Airsim: High-fidelity visual and physical simulation for autonomous vehicles, *in* 'Field and Service Robotics: Results of the 11th International Conference', Springer, pp. 621–635.

[36] Staranowicz, A. and Mariottini, G. L. [2011], A survey and comparison of commercial and open-source robotic simulator software, *in* 'Proceedings of the 4th International Conference on PErvasive Technologies Related to Assistive Environments', PETRA '11, Association for Computing Machinery, New York, NY, USA.
URL: *https://doi.org/10.1145/2141622.2141689*

[37] Takaya, K., Asai, T., Kroumov, V. and Smarandache, F. [2016], Simulation environment for mobile robots testing using ros and gazebo, *in* '2016 20th International Conference on System Theory, Control and Computing (ICSTCC)', IEEE, pp. 96–101.

[38] *URDF Specification* [n.d.], `http://wiki.ros.org/urdf/XML/model`. Accessed 06-08-2024.

[39] *URoboSim* [n.d.], `https://github.com/urobosim/UROSBridge/tree/Fix5.2?tab=readme-ov-file`. last accessed: 12.09.2024.

[40] *UROSBridge* [n.d.], `https://github.com/robcog-iai/UROSBridge`. last accessed: 13.09.2024.

[41] Vivan, G. P., Goberville, N., Asher, Z., Brown, N. and Rojas, J. [2021], No cost autonomous vehicle advancements in carla through ros, Technical report, SAE Technical Paper.

[42] *Webots* [n.d.], `https://cyberbotics.com/`. Accessed 07-10-2024.

[43] Weyer, S., Meyer, T., Ohmer, M., Gorecky, D. and Zühlke, D. [2016], 'Future modeling and simulation of cps-based factories: an example from the automotive industry', *IFAC-PapersOnLine* **49**(31), 97–102. 12th IFAC Workshop on Intelligent Manufacturing Systems IMS 2016.
URL: *https://www.sciencedirect.com/science/article/pii/S2405896316328397*

[44] Yoon, J., Son, B. and Lee, D. [2023], 'Comparative study of physics engines for robot simulation with mechanical interaction', *Applied Sciences* **13**(2).
URL: *https://www.mdpi.com/2076-3417/13/2/680*