Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

BACHELORTHESIS

Automated Generation of Source Files for a ROS-Unity Interface

submitted by

Phil Holdorf

University of Hamburg
Department of Informatics
TAMS - Technical Aspects of Multimodal Systems
Course of studies: Informatics
Matriculation number: 6834144

Primary supervisor: Prof. Dr. Jianwei Zhang
Secondary supervisor: M. Sc. Yannick Jonetzko

Submission date: July 30, 2019

Acknowledgement

I would like to specially thank research associates Lasse Einig and Yannick Jonetzko at TAMS for their continuous support during the writing process.

Abstract

The goal of this project is to ease and speed up the development of Unity3D applications which interact with the Robot Operating System (ROS). For this purpose, the current development process and the available frameworks have been analyzed. Then, a tool was developed as an extension to the existing interfaces. The basis for this project is Siemens' Ros#, a promising tool which enables communication between ROS and Unity. The main contribution of this work is a feature which automates the generation of source files in Unity from imported ROS project files, thereby reducing development time.

Zusammenfassung

Das Ziel dieser Arbeit ist es, die Entwicklung von Unity3D-Applikationen, welche mit dem Robot Operating System (ROS) interagieren, zu vereinfachen und zu beschleunigen. Mit dieser Absicht wurden die verfügbaren Frameworks sowie der aktuelle Entwicklungsprozess analysiert. Es wurde ein neues Tool entwickelt, welches die bestehenden Schnittstellen erweitert. Die Basis für dieses Projekt ist Siemens' Ros#, ein aussichtsreiches Tool welches Kommunikation zwischen ROS und Unity ermöglicht. Der primäre Beitrag dieser Arbeit besteht in der automatiserten Generierung von Quell-Dateien in Unity aus importierten ROS-Projektdateien, um somit die Entwicklungszeit zu reduzieren.

Contents

1 Introduction

Developing robotics software presents a diverse set of challenges, in large part due to the heterogeneous nature of the robot hardware being used. These challenges include interaction with the various sensors and actuators, communication between nodes of a robotics system, as well as simulation and visualization tasks. Attempts have thus been made to provide a unified set of tools and libraries to address these recurring problems. Arguably the most significant of these contributions is the Robot Operating System (ROS) [1], which has become a standard in the scientific community.

One aspect related to ROS is the simulation of robots and their environments. This is an important task for research and development, because it allows the testing of hardware in varying circumstances which could not be easily replicated in the real world without investing a significant amount of time and effort. Although simulation tools are included with a ROS installation in the form of Gazebo [2], there is still room for improvement in this area.

With the recent rise in popularity of game engines such as Unity [3], attempts have been made to integrate them with ROS. The possible research applications for such an interface are numerous. However, a context-independent framework with this purpose has yet to be found and is the topic of this thesis. The approach that is evaluated here is to use Unity with the Ros# [4] extension.

1.1 Motivation

There are several benefits to utilizing a game engine in place of, or in addition to, a well-established robot simulator such as Gazebo. One of the first considerations is a high degree of graphical fidelity. Since it is one of the main selling points of the most used game engines, the quality in this area will likely continue to improve in an effort to remain competitive.

Unity has established itself as one of the most popular game engines. It presents many additional benefits. In the field of robotics research, some of its most exciting features are the built-in support for Virtual Reality (VR) devices, and the extensibility via scripts. This allows the creation of custom user interfaces for controlling and visualizing robot interactions, either with the real world or in a simulated environment. These capabilities make it a versatile research platform for use in robotics software. For example, Unity has been used in combination with ROS for a diverse range of applications, such as: simulating multiple drones in flight [5], monitoring a welding process [6], and controlling a robot arm through a VR interface [7].

Although Unity is often used by professionals in commercial projects, it may be

particularly useful to students and beginner programmers as well. The main arguments for this are the intuitive component-oriented design and the amount of tutorials and documentation available, resulting from the popularity of the engine. As a working environment, it is both highly accessible and offers an array of productivity-enhancing features.

Despite this, there is still a considerable learning curve for Unity developers who are unfamiliar with ROS. Even for those experienced with ROS, it currently takes longer than necessary to integrate Unity with an existing ROS project. The goal here is to save development time by automating recurring processes, and to create further abstractions where possible to reduce perceived complexity. If the entry barrier were to be lowered, it could help ROS projects gain further exposure, and allow more flexibility for possible research applications. Specifically, it allows for closer cooperation between experienced ROS users and those unfamiliar with the framework and environment. It could allow new developers to more quickly contribute solutions for a ROS environment through Unity.

1.2 Thesis goal

The goal of this thesis is twofold: To closely examine an existing ROS-Unity framework in terms of its productivity, and to improve it based on this aspect. The problem to be addressed is the relatively high amount of time and effort it takes to begin developing a Unity application for any given ROS project.

The framework chosen for this task is Ros#, which is developed by Siemens. It aims to provide a general-purpose interface between Unity and ROS, rather than being focused on any specific use case. It is based on RosBridge [8], which enables cross-platform communication with ROS from other systems using a web-socket interface. Ros# provides the Unity-side implementation of this interface.

In order to identify possible improvements to this software setup, this work tries to answer three central questions regarding the workflow:
1. What are some use cases of ROS-Unity applications?
2. Which steps in the development process do these use cases have in common?
3. How can the proposed framework assist to increase productivity at each of these steps?

In addition to these questions, one basic problem was identified immediately. Namely, that the number of message types currently supported by Ros# is limited. New message types must be added manually by the user. The requirement derived from this problem is that it should be possible to automatically generate the necessary message files in Unity for any given ROS project.

1.3 Chapter guide

Chapter 2 gives an overview of related work, focusing on robot simulators, modern use of game engines, and frameworks which are similar to Ros#. This serves to provide context for recent developments, and helps to reveal possible improvements to Ros#.

Chapter 3 explains the foundation which the proposed work is built on. Readers are familiarized with ROS, Unity, RosBridge and Ros#. Problems with the current state of this software setup are identified, focusing on construction of a robot scene within Unity and Ros#.

In chapter 4, an extension to Ros# is presented. A list of features is given. These features are then tested in a sample application, serving as a proof of concept.

Chapter 5 addresses implementation details, giving insight into some of the problems encountered and design decisions that were made.

Finally, chapter 6 is a discussion on how well the requirements could be met and what improvements could be made in the future.

2 Related work

Research on the topics of robot simulation and robot interaction is fairly comprehensive, as groups with different requirements each contribute their own unique approach. The use of game engines for this context is also commonplace. When it comes to general-purpose frameworks combining game engines with ROS, a number of proposals have been made only very recently.

2.1 Robot simulators

A large variety of robot simulators have been in use before and after the release of ROS. Looking at the history of robot simulation provides context for the current state of the art. Some simulators are based on existing game engines. These will be looked at in more detail, to find out what motivated this choice and how their requirements differ.

Gazebo [2] is an open-source 3D simulator released in 2004 for the Player project [9], and is now integrated with ROS as well. It is focused on providing a physics simulation which is as accurate as possible. Several different physics engines can be chosen for it. It remains popular as a standard choice for ROS users.

OpenHRP [10], released in 2004, is a simulator designed specifically for humanoid robots and does not support ROS on its own.

Webots [11] is another early but comprehensive attempt at robot simulation which predates the release of ROS. The software is proprietary, which limits its usefulness in a research context.

Another general-purpose simulator is V-REP [12]. The paper mentions all three of the above simulators, but they were found insufficient at the time due to versatility concerns in their programming interface. ROS is supported by it, and a free educational version is offered.

Following is a selection of simulators which utilize game engines. USARSim [13] is an early attempt of incorporating the Unreal Engine [14] into robot simulation, with development starting in 2002. Initially used for simulated urban search and rescue missions and competitions, USARSim has been widely popular and was developed further into an extensible general-purpose robot simulator. An extension exists which also adds support for ROS [15].

The Search and Rescue Game Environment (SARGE) [16] was an early adopter of Unity in 2008. Its purpose was to provide a training environment for robot operators. The developers decided to switch over from USARSim and Unreal Engine to Unity. Some

of the main reasons for this were Unity's developer-friendly environment and low cost compared to its competitors. They had also considered purpose-built robot simulators such as Webots and Gazebo, which already provided a high quality of physics simulation. However, they concluded that game engines proved to be better tools when it came to creating complex environments in a short time and with a high rendering quality. It should be noted that some of the information in the paper is no longer accurate. For example, both Unity and Unreal Engine now provide free versions.

MORSE [17], released in 2011, was designed with flexibility in mind. It supports aerial, ground and maritime robots. The software is fully open-source and independent of any robot middleware, although it has built-in support for ROS and a number of other communication mechanisms. It is built on top of Blender [18], a popular open-source software for 3D modeling, animating and related tasks. Although not primarily thought of as a game engine, Blender does offer tools for game creation and is extensible using Python scripts, which is exploited by MORSE.

Several surveys of robot simulation tools have been published in the past. Two recent surveys are outlined in the following. In 2014, A survey was conducted, comparing simulators for the purpose of underwater vehicles [19]. This provides some insight into how requirements for simulators may differ between projects. Among the simulators they reviewed were Gazebo, MORSE, V-REP, USARSim, as well as UWSim [20], a tool specific to underwater missions. In the 2014 survey, it was mentioned that MORSE suffers from a high learning curve when it comes to extending the existing environments, due to its reliance on the Blender interface. This indicates that preferences in user interfaces may play a major role in deciding which simulator to use.

In 2017, another survey compared the 3D simulators V-REP, Gazebo, MORSE, Webots and USARSim [21]. The tools were evaluated for a multi-robot patrolling scenario. MORSE and Gazebo were tested in terms of their performance. The result was that MORSE performed slightly better with multiple robots than Gazebo.

2.2 Recent use of game engines

In the previous section, three robot simulators were mentioned which were built on top of an existing game engine: MORSE, USARSim and SARGE. The engines used were Blender, Unreal Engine, and Unity, respectively. Three main reasons for using game engines in robotics simulation are identified: Firstly, there is the promise of an especially high degree of graphical fidelity. Secondly, they offer a working environment optimized for quickly assembling complex virtual environments, which may help to increase productivity during development. Thirdly, game engines are an intuitive choice when humans are required to actively interact with the simulation, and often provide excellent tools for this purpose.

A few more specific use cases will be looked at in this section. Many of these applications focus on human-robot interaction (HRI). The aim here is to demonstrate a recent increase in robotics research related to game engines, and to showcase the

diverse nature of these applications.

UnrealCV [22] is a tool which uses Unreal Engine 4 specifically to create realistic images, which are then used as training data for computer vision algorithms.

"The Robot Engine" [23] is a Unity plugin for use in HRI tasks. Robot models can be imported into Unity, where the robot can be animated without requiring programming knowledge. Sensor data captured by a real robot can be processed within Unity to cause a reaction from the robot. These behaviors can then be transferred onto the real robot. Currently, the plugin has only been set up for use with Arduino controllers, lacking support for ROS and other platforms.

In contrast, a system which does combine ROS with Unity is RosUnitySim [5]. The scope of this tool is limited to the context of simulating multiple drones in flight, otherwise known as unmanned aerial vehicles (UAV). Communication between ROS and Unity happens via TCP/IP protocol. One reason that was given for chosing Unity over other engines is the ability to assign closely matching collision models to each object, such as trees and buildings. This is called a mesh collider in Unity, and is opposed to using simpler geometric shapes such as boxes and cylinders for collision detection. Such simple shapes require less processing power in the physics simulation, but are not suited for modeling each individual branch of a tree, for example.

Another ROS-Unity interface exists for the purpose of monitoring an industrial manufacturing process [6]. The program has been tested for a heavy welding process, but is independent of any particular industrial task. They have chosen Unity over a dedicated simulator such as Gazebo, because it allowed them greater flexibility in designing the user interface. RosBridge was used as a communication protocol between ROS and Unity.

Several HRI interfaces incorporate VR into a ROS-Unity context. Three of them are mentioned in the following. All of these examples use ROS, Unity and RosBridge. In 2014, an interface using a head-mounted display (HMD) for teleoperation of a single ground vehicle was developed [24]. A similar project from the University of Hamburg combines a HMD with a hand tracking device to give users control of a robot hand in VR, such that the real hand motions are translated into movements of a real robotic gripper [7]. The third and most recent project uses a VR interface for monitoring and commmanding multiple robots as well as UAVs [25]. In addition to RosBridge, the project also uses RosBridgeLib [26] for its Unity-side communication with ROS.

2.3 Comparing Unreal Engine and Unity

From this selection of projects, it seems that robotics research related to game engines is increasingly based on Unity, although this may be a biased observation. Either way, it is worth noting that Unreal Engine 4 advertises a set of features that are strikingly similar to Unity, including support for VR devices, the ability to build applications for a large variety of operating systems, as well as a cost-free license.

One advantage it has over Unity is full access to the complete source code of the engine.

Unity does make it easy to extend its interface with new functionality. However, if there is a bug in the Unity engine itself, users will have to rely on workarounds until the bug is resolved.

Another difference between the two game engines is that Unreal Engine uses C++ as its programming interface, while Unity uses C#. This comes down to preference, but an argument could be made that C# is easier to use and thus increases productivity. Overall, Unreal Engine seems to be just as suitable for robotics research, although its use in this context may not have been proven as extensively yet.

2.4 Frameworks

It was shown that the possible applications for an interface between robotics and game engines are vast. The recent rise in virtual reality interfaces is especially notable. There are some recurring tasks for these types of applications. Mainly, to enable communication between the game engine and a robotics system (ROS in particular). Until recently, this task had to be solved individually by each developer. This has created demand for a standard framework to build on top of and to speed up development. Some of the attempts at such a framework are presented here. All of these interfaces are between ROS and Unity.

RosBridgeLib [26] is a bare-bones library which implements the RosBridge protocol on the side of Unity. This enables communication with ROS systems, although the number of usable message types is limited. No further tools are provided to aid in the construction of ROS-Unity projects.

RosReality [27] is an effort to provide a VR interface for ROS, which allows viewing and remote controlling robots over the Internet. It includes a tool to import robot descriptions into Unity, which are known as URDF files. On the project's Github page[28], it is noted that the bridge between ROS and Unity (ROS Reality Bridge) is provided as legacy code, and that their new projects are based on Ros#. This seems to indicate that RosReality is no longer actively developed.

A project named SIGVerse [29] has the goal of reducing development costs for HRI tasks involving VR devices. It offers a number of features for constructing experiments involving a human and a robot in a virtual space. Participants can complete the experiment over the Internet. Data of the experiment is then recorded for use in robot learning functions, which includes a recording of the interaction. The developers chose not to use RosBridge for remote communication, due to a bottleneck in performance when transmitting large binary messages such as camera images. Instead, data transfer is accomplished with the use of the BSON format and a TCP/IP connection.

In the same year, an interface with a similar objective was developed at the University of Hamburg [30]. The focus of this project is on accessibility, allowing the creation of HRI prototypes without requiring expert knowledge on ROS. It was developed and tested with VR teleoperation and control tasks in mind. The use of HMDs and a hand tracking device is supported. Robot models can be imported from UDRF files or 3D mesh files.

Overall, it can be noted that all of the frameworks shown here are very recent developments. They mostly share similar goals. Besides the task of connecting ROS and Unity, they provide additional features to aid in their specific research interests, which are all related to HRI. The differences lie in the specific hardware and use cases which they are most equipped for.

Ultimately, Ros# is preferred as a general-purpose ROS-Unity framework because it abstracts from the context it is used in. Furthermore, and unlike RosBridgeLib, it provides additonal tools such as a URDF importer. Ros# seems to be in an early development stage, and new features are still being actively developed. If it establishes itself as a standard framework, it can be expected that context-specific features will be added as plugins by the community. It remains to be seen how well Ros# can fulfill the requirements of current and future projects. Some problems with the software are identified in this work, and a solution is offered as an extension to the software.

3 Basics

3.1 ROS - Robot Operating System

The Robot Operating System (ROS) is an open-source framework for developing, testing and running robotics software [1], [31]. At its core, it is a system for passing messages between processes, often distributed over many different types of devices. On top of that, it can be extended with a large set of tools and libraries. The motivation behind its extensible architecture is to enable robotics researchers and developers to share reusable functionality in a convenient way. This can include device drivers, common algorithms, debugging tools and so on. It is worth noting that despite its name, ROS is not an operating system. It is middleware which runs on Ubuntu, Mac OS X, and several more Linux distributions. The main programming languages used by ROS developers are C++ and Python, although a few others are available as well.

3.1.1 Communication concepts

In ROS, communication between processes can be both synchronous and asynchronous. Using asynchronous communication means that messages are sent without waiting for a response. In contrast, synchronous communication follows a request-response pattern, which means that a process sending a request will wait until the response is received before continuing work. Understanding ROS communication requires knowledge of a few basic concepts:

Nodes ROS is designed to be highly modular, allowing it to be used effectively in a large-scale distributed environment. Therefore, ROS systems are commonly comprised of many separate processes which are called nodes. Each node performs a certain task and exchanges information with other nodes through messages.

Client libraries The code running on a node is written by using a ROS client library. These libraries implement the ROS interface in many different programming languages. The work in this thesis utilizes the two most common client libraries, which are roscpp for C++ and rospy for Python. This highlights portability as another important design goal of ROS.

Master To initiate communication, nodes must first contact the ROS master node. The master can locate each node, and it tracks which nodes are communicating. After contacting the master, nodes will then communicate peer-to-peer. This means that nodes are loosely coupled and do not depend on each other. The master is only

involved in registering and de-registering nodes. The console command 'roscore' can be used to launch the ROS master.

Messages Nodes exchange data via messages, which are defined in msg files. The format is very simple: the files consist of fields, which are pairs of a datatype and a field name. Beyond primitive types and arrays, fields can also use other message types, or they can define a constant. Messages and data types will be looked at in more detail later.

Listing 3.1: An example for a complete message file

```
int32 first_value
uint16[] second_value
uint8 a_constant=5
```

Topics Topics are the basis for asynchronous, many-to-many communication in ROS. A topic is defined by a name and a message type. Defining a new topic within ROS is also called 'advertising'. Sending messages on a topic is called 'publishing'. Any node can publish messages on a given topic, and any node can then 'subscribe' to a topic to receive all newly-published messages. Topics are best used for continuous streaming of data, such as when sensor data should be made available in regular intervals to many recipients.

Services Services are nodes which can be called by another node, enabling synchronous communication. A set of inputs is given to the service, and an output is produced. Services are defined in srv files, which have two parts: a request message and a response message, separated by three dashes. The implementation of a service is defined separately from its srv file.

Listing 3.2: An example service file

```
string request_value1
int32 request_value2
---
int32 response_value
```

Parameter Server The parameter server provides storage for values in a key-value format. It resides on the master. This can be used for global configuration of a ROS system, although a service could be used in a similar way.

3.1.2 File system

In ROS, all software is shipped as **packages**. This concept is used by the community as well as by ROS developers. The main benefit of this structure is that packages can be installed directly through ROS commands, which in turn allows dependencies of each package to be resolved automatically. Packages are directories with a certain structure and a 'package.xml' file. This file includes information such as the author, version and

license used, as well as a list of dependencies on other packages. Message files are found in the 'msg' folder, and service files in the 'srv' folder.

To build the source code within a package into libraries, executables and related files, ROS uses a custom build system named **catkin** . The build configuration for a package is contained in the file 'CMakeLists.txt'. In order to use custom message and service definitions in a script, CMakeLists.txt must be modified to include references to these files. To start the build process, the command 'catkin_make' is used.

An option exists to run multiple nodes at once in a single terminal. This is done by adding **launch files** and using the 'roslaunch' command. Since most applications are made up of several nodes or depend on nodes in other packages, this allows bringing up a whole system in one command. The launch file allows configuration of this process by specifying parameters and more.

3.2 Unity

Unity is a game engine and game development platform for 3D and 2D games [3]. At the time of writing it is free to use for any project not exceeding $100,000 per year in revenue or funding. Its popularity compared to other free game engines is among the highest, mainly competing with the Unreal Engine.

3.2.1 Working environment

Apart from a high quality rendering engine and a physics engine, Unity provides extensibility via a C# programming interface, several tools related to setting up environments and user interfaces, modifying game physics and lighting, animating objects, debugging, profiling and more. In general, Unity satisfies the requirements that are shared by all types of games, which helps to keep the working environment clean and thin. Beyond that, support for any particular game genre or a specific work flow is left to the community.

The Unity editor is highly extensible with custom user interfaces. These are written in C#, with access to the same libraries as code that would be written for a game, as well as a dedicated library for editor scripts. Such extensions are created by the community and made available on the Unity Asset Store, some of which are offered for free. One can also find a variety of 3D and 2D models, complete environments or object sets, and AI scripts such as pathfinding algorithms. The integration of this content sharing platform is in some ways similar to the ROS model of package sharing. It offsets the lack of an open source repository for the Unity engine.

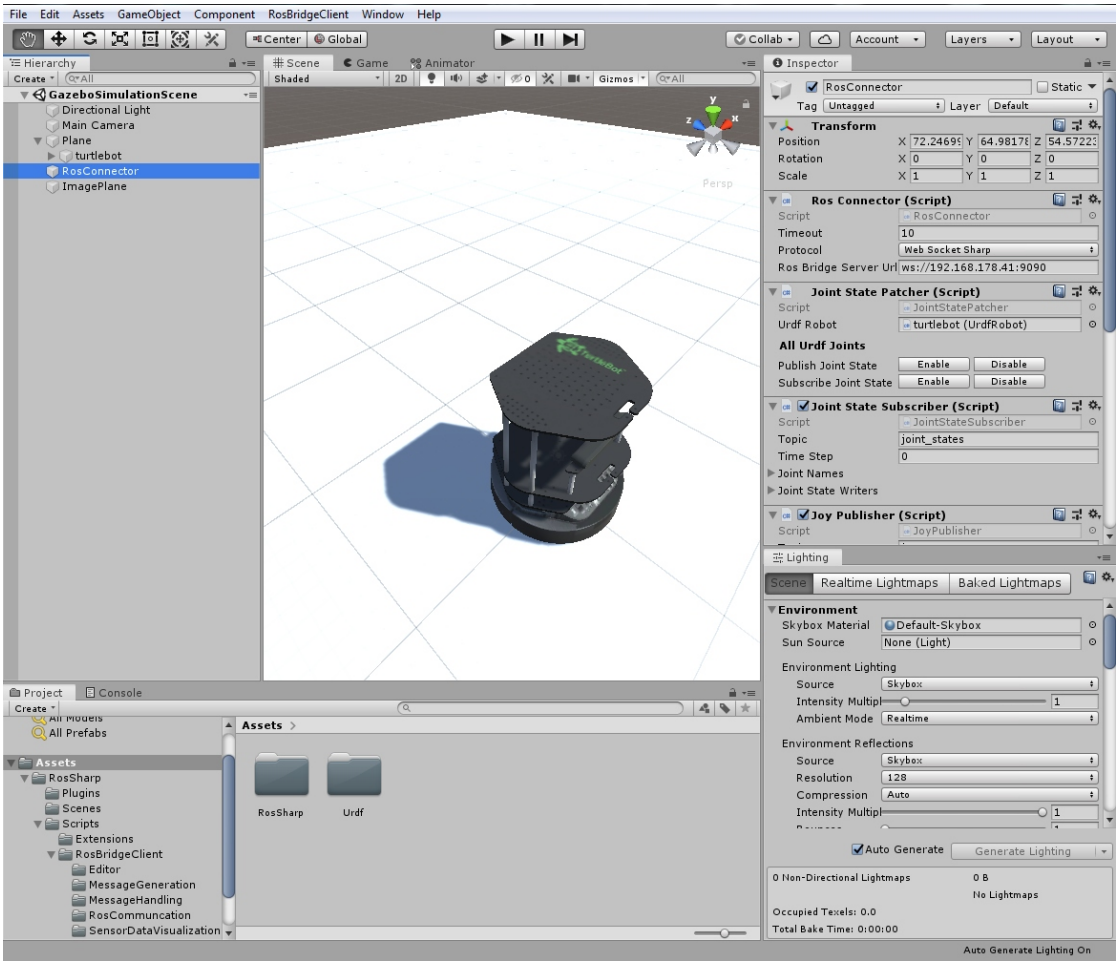

Figure 3.1: The Unity editor view. On the left is the scene hierarchy, listing and grouping objects in the current scene. The scene itself is visible in the center. On the right, the inspector window lets users edit the properties of a selected object, as well as all the scripts attached to it. At the bottom, the project view shows the available resources and scripts. Each window or view can be rearranged by clicking and dragging.

3.2.2 Scripting

At the core of the Unity scripting engine is the GameObject. They exist in the scene hierarchy and have a Transform component attached to them, giving them a position, rotation and scale in 3D space. A variety of optional components can be added which are used for rendering, physics, animation and other purposes. Any script which inherits from MonoBehaviour can be attached to a GameObject. This requirement discourages the use of polymorphism, as multiple inheritance is not supported in C#. The idea is that each GameObject can be treated as an independent entity composited of many

small, re-usable parts.

Extending MonoBehaviour in a class will make it function as part of the Unity framework. There is no main entry point in Unity. Instead, scripts are called through inherited methods when certain events are triggered. Initilization is done in the Awake and Start methods, which are only called once in the lifetime of an object. The Update method is called once per rendering frame, and FixedUpdate is called once per physics frame, the duration of which can be configured in the project settings. The full scripting documentation is available online and is quite comprehensive [32].

3.3 RosBridge

RosBridge allows any application to communicate with a ROS system over a network through a web socket [8], [33]. This is most useful on operating systems which do not support ROS, thus providing a basis for ROS-Unity interfaces. It consists of three ROS packages, all of which are contained in the 'rosbridge_suite' package:

rosbridge_library This package handles conversion between JSON strings and ROS.

rosbridge_server Provides a web socket for communication.

rosapi Defines services which expose a subset of ROS commands to the network.

In order to communicate with ROS from a remote device using RosBridge, the RosBridge Protocol [34] must be implemented locally. This means generating and sending JSON objects to the rosbridge server. All rosbridge messages must include the field 'op', indicating which operation to use. Another optional field is 'id', which identifies and groups one or more messages together as part of an exchange between the client and the server. The supported ROS-related operations are: 'advertise', 'unadvertise', 'publish', 'subscribe', 'unsubscribe', 'call_service', and 'service_response', which do as their names suggest. These operations define further fields which can be required or optional.

```
{ "op": "publish",
  "id": some_string,
  "topic": some_topic,
  "msg": {"x" : 5.3, "y" : 0, "z" : -3.1}
}
```

Listing 3.3: An example for a rosbridge message publishing a Point message.

3.4 Ros#

Ros# is an open-source extension for the Unity editor which implements the RosBridge protocol, thus enabling communication between ROS and Unity [4]. Additionally, it provides tools to improve integration with ROS, such as an importer for robot

descriptions (URDF files). Ros# can be used for robot & sensor visualization, as a simulation environment, and for teleoperation. This chapter provides an overview of its components and features, highlighting some of the areas in which the software can be improved. An example project will demonstrate how to connect to ROS as well as publish and subscribe to topics. Following after that is a look at how a project can be extended with custom messages and behaviors.

3.4.1 System overview

The repository at ros-sharp/master contains three modules: Libraries, ROS and Unity3D. Following is a summary of these modules and a look at some relevant features.

ROS Module On the ROS side, three packages are provided, the most integral of which is the file_server package. It includes the service node file_server, which simply outputs the file contents of a requested file in any ROS package. The launch file ros_sharp_communication.launch launches the rosbridge websocket and the file server node.

Libraries Module This module provides an abstraction of the RosBridge protocol, access to the file_server service and URDF transfer functionality, as well as some context-specific ROS messages encoded as CSharp files. When compiled, two binary files are created: RosBridgeClient.dll and URDF.dll. Both of these files must be copied into the Unity project folder.

Unity3D Module This module can be split into two functionalities: extensions to the Unity Editor, and Unity run-time scripts which are specific to the example applications. Some of the examples are re-usable for other ROS-Unity projects.

3.4.2 Example project

An example for a project setup with instructions can be found at the Ros# wiki page[35]. The example sets up a simple scene in Unity. It allows teleoperation of a Turtlebot [36] using arrow keys. The inputs are sent to a Gazebo simulation on a remote machine, but a connection to a real Turtlebot could be established in the same way. Following is an overview of the steps required to set up such a project.

First, the URDF model of the Turtlebot is imported into Unity. On the ROS side, a launch file is used which uploads the URDF data to the parameter server. On the Unity side, the URDF transfer dialog is used to initiate the import process. This creates a Turtlebot game object in Unity.

The example then introduces the script component RosConnector. This is added to a game object in the scene. After choosing a protocol type and entering the IP address of the ROS machine, the application is ready to send and receive communications at runtime.

The next step is subscribing and publishing to topics. This requires manual configuration on many components inside the Unity editor, and will differ for each new

project, as different topics and behaviors will be used. Although tutorials for Ros# are provided, it currently lacks documentation for its individual components, so this process comes with a learning curve, especially for users unfamiliar with ROS. The topics and ROS nodes used in the example are illustrated in figure 3.2 .

The rest of the scripts are responsible for various physics and rendering tasks. The Turtlebot is simulated within Unity as a physical object, and the camera image received from Gazebo is projected onto a plane.

After starting Gazebo on the ROS machine via the launch file provided and hitting Play in Unity, the result should be a Turtlebot model that can be moved with arrow keys. The inputs are published to ROS, and the captured camera image and robot state are displayed in Unity.

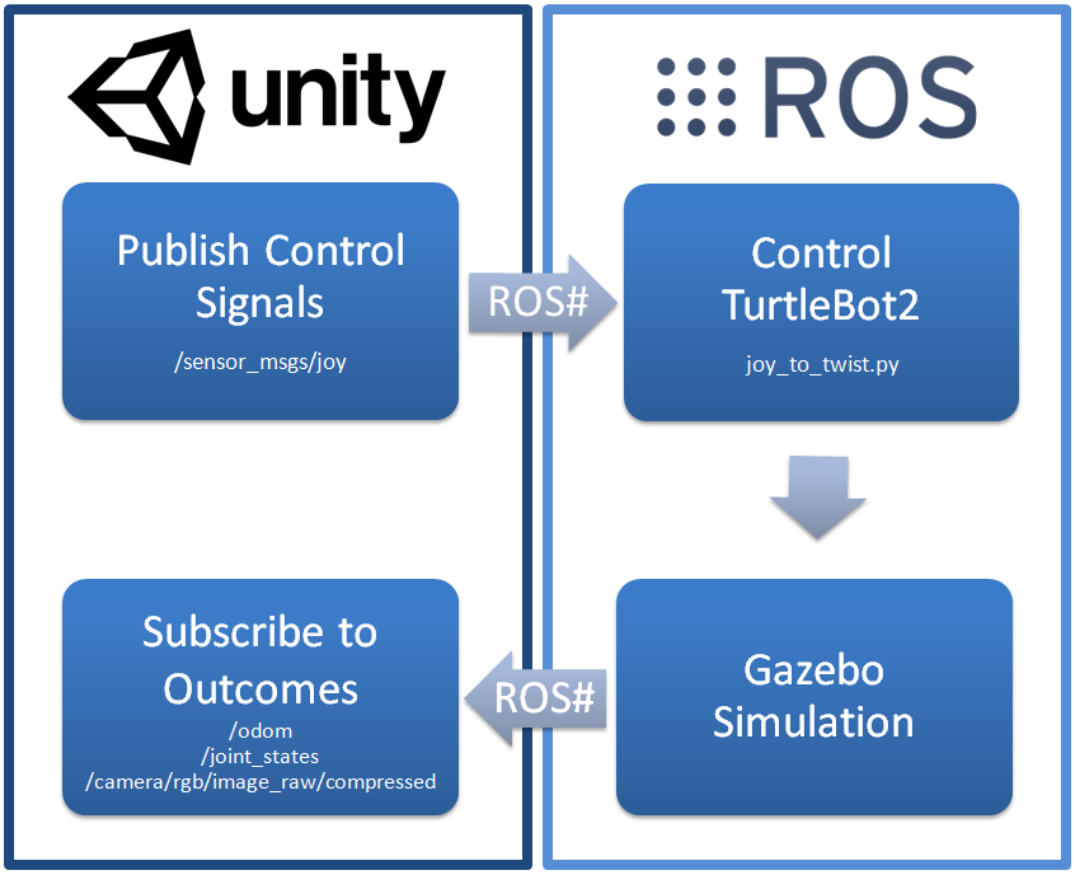

Figure 3.2: Communications between Unity and ROS for the Gazebo simulation example. The diagram shows the published and subscribed topics and the ROS nodes involved. The arrow keys are used to control the Turtlebot in Unity. These inputs are then published to ROS, where they are converted into Twist messages and sent to a simulated Turtlebot running in Gazebo. Sensor data from this simulation is sent back to Unity to be processed. Image source: Ros# wiki [35]

3.4.3 Customizing a project

When creating new applications based on Ros#, one of the first things to add is new ROS messages in the form of CSharp files. The message types included in Ros# are very limited compared to those of a standard ROS installation. Recently, a new dialog was added to Ros# which allows generating the necessary CSharp files for any message needed in the project. However, this method still requires manual input, and there is one other issue with it: The field types for a message are chosen from a drop-down menu, the contents of which are hard-coded as an enumerable type. At the time of writing, very

few custom message types are supported. Some primitive types are also missing, such as Byte and Boolean.

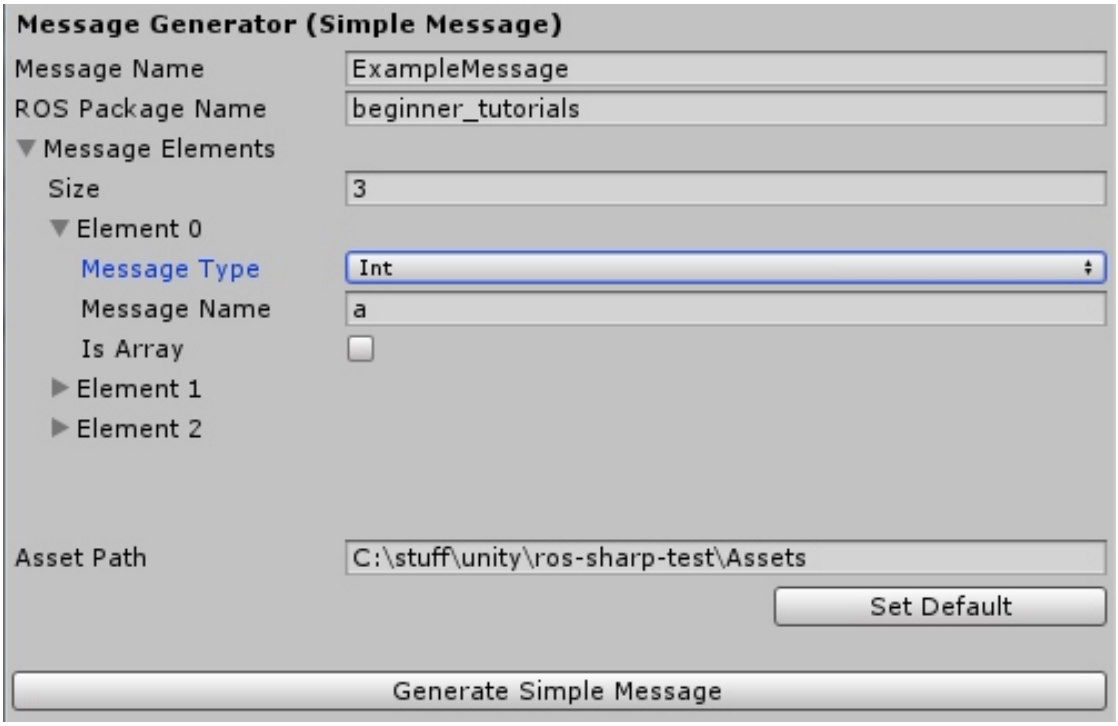

Figure 3.3: The message generation window included with Ros#. Messages are customizable by setting a name and add a number of message elements by choosing a type from the drop-down menu.

Similar problems arise when adding new subscriber and publisher classes in Unity. The process for this is currently undocumented, although the approach used in the example application can be carried over to other ROS topics. No feature exists yet to automate this recurring task.

A few other features are provided by Ros#. Robot models can be built using the Unity editor and exported as ROS-compatible URDF files. The file server provided by Ros# can be contacted as a service through the RosBridge interface, but on the Unity side, the necessary code to establish the connection and locate files within a package is missing. Beyond that, a rather minimalist approach was taken with the framework. No further support is currently available for specific use cases such as HRI and VR devices.

4 Design

This chapter introduces an extension to Ros# which aims to improve it in terms of productivity and extensibility. The design decisions made before and during development are documented, including a list of features and a description of the user interface. Finally, a method is proposed by which to test if the application fulfills the requirements. This includes an installation guide and a proof of concept.

4.1 Problem summary

The main problems with the current state of Ros# were identified in section 3.4 : When starting a new Unity project utilizing an existing ROS package, considerable effort is spent on recurring setup tasks. Essentially, the Unity project has to be made ready before actual development can begin. This is especially important when trying to rapidly set up prototypes for a given hardware configuration.

The recurring tasks include producing CSharp code for ROS messages, services, publishers and subscribers. Additionally, the script components of the RosConnector, publishers and subscribers must be added to game objects. There are two aspects which make these recurring tasks act as a barrier to rapid prototyping: the first is the time spent on manual labor, which can add up if a large number of messages and topics are used. The second is the learning curve developers are faced with when configuring objects within Unity, due to a lack of documentation for individual Ros# components. This learning curve is especially notable for developers unfamiliar with either ROS or Unity.

4.2 Proposed features

The extension to Ros# adds another dialog option to the Unity editor. This dialog allows finding ROS messages on the remote machine, transferring them, and generating CSharp code for them automatically.

If a message has dependencies to other messages, these files should also be retrieved in the process. For example, attempting to generate the message 'geometry_msgs/Twist' will also generate the dependency 'geometry_msgs/Vector3'. Each generated message class is uniquely identifiable by its fully qualified type name. This type name can be mapped to the ROS message name.

Since there can be a very large number of messages to be displayed to the user, these messages should be organized in some way. In the dialog, the user should be given the option to select individual messages to transfer. Additionally, it should be possible to

search for messages in specific packages. ROS also offers a command to list all messages in active topics, so this option should be included as well.

4.3 User interface

Figure 4.1 showcases the user interface for the message generation window. The workflow for this dialog can be sectioned into three stages: Message search, selection, and code generation.

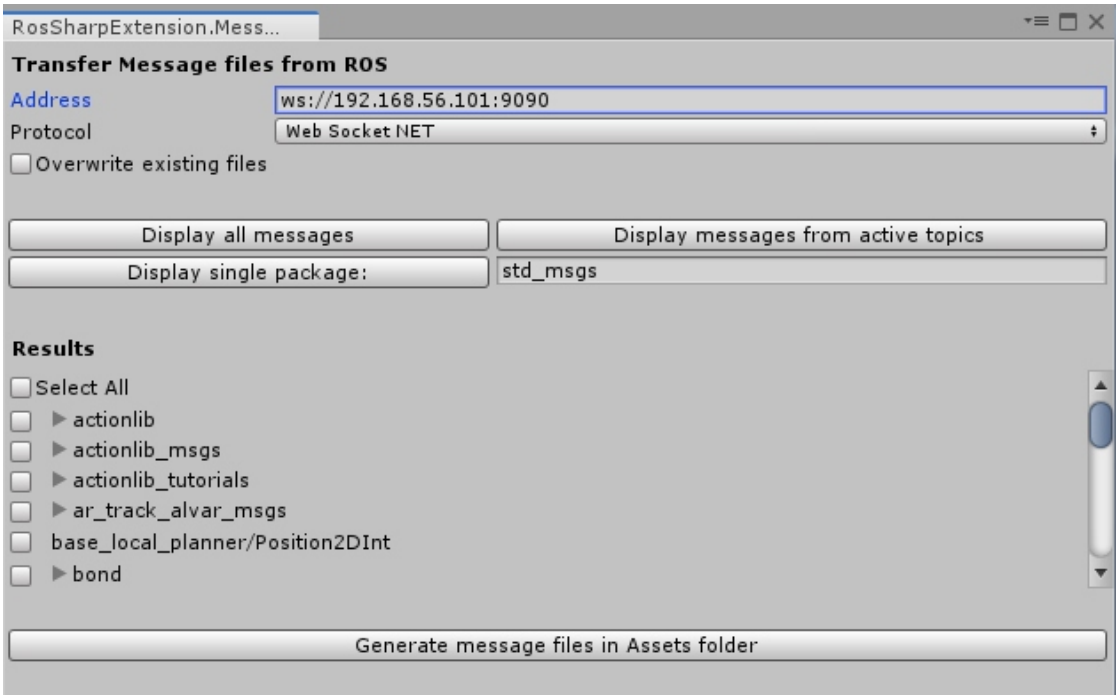

Figure 4.1: The message transfer window, showing the setup and search options at the top, the search results and message selection in the middle, and the generate function at the bottom. The window is accessible through a menu in the Unity editor.

Searching messages ROS message types which are present on a remote machine can be listed in Unity through a user interface. There are three options available for listing messages. The first option lists all messages saved on the remote machine. The second option lists only messages which belong to a currently active topic. Finally, the third option allows the user to input a package name, then lists all messages within that package. An option exists to allow or disallow existing files from being overwritten. After initiating one of the search options, there is a wait time before messages are displayed. An important note is that the tool remains functional even

when there are compile errors within the current Unity project. Otherwise, there could be a case where message files are missing but cannot be retrieved by using the tool.

Selecting messages From the list of messages, the user can make a selection. The messages are grouped by their package names for ease of navigation. Each package can be expanded to reveal the contained messages. There is an option to select and unselect all messages within a package, and another option to select all messages across all packages.

Generating code From the selected messages, C# class files can be automatically generated and saved into the Unity project directory by pressing a button, which initiates another waiting time. From the user's perspective, the file transfer and code generation is a single step. When all source files are generated, the Unity project will be recompiled and the user will be notified that the process has finished.

4.4 Installation

In Ubuntu, the Ros# package 'file_server' must be installed. Then, the package 'ros_sharp_extension' must be copied into the 'src' directory of the ROS work space, followed by a 'catkin_make' command executed from the work space directory. To launch the software, run the following command:

```
$ roslaunch ros_sharp_extension ros_sharp_extension.launch
```

On Windows, the dependencies to compile RosSharpExtension.dll are: Microsoft Visual Studio 2017, .Net Framework 4.6, RosBridgeClient.dll (included in Ros#), UnityEngine.dll, UnityEditor.dll. As of Unity version 2019.1.1f1, the latter two libraries can be found in the directory 'Unity\Editor\Data\Managed' of the Unity installation. After compilation, the resulting binary file RosSharpExtension.dll must be placed in the Unity project directory, alongside RosBridgeClient.dll.

4.5 Proof of concept

In order to test the viability of the design, it should be proven that the automated message generation feature is functional, since that is the most fundamental requirement of this project. There are four steps to this proof:

1. An example Unity project is used as a base. This project uses messages which were originally included in Ros#.

2. The Ros# messages are deleted, resulting in compile errors from missing types.

3. The messages are added back via the message transfer window.

4. The resulting project should now function the same as in step 1.

To execute these steps, some preparation was needed. The example project used in step 1 is included in the ros-sharp-extension software under the "Unity" directory [1] . It is based on the set up created in section 3.4.2 , with some modifications which are explained below.

Firstly, the namespaces of the Ros# messages were changed to match ROS package names. This was done in both the RosBridgeClient library and in the Unity project. Not doing so would mean that the generated message classes are named differently from the original Ros# classes, causing compile errors. An example of this naming mismatch is the namespace 'Navigation', which contains messages from the package 'nav_msgs'.

The second modification was to change the types of some Ros# messages, because they did not match the types defined in the ROS messages. Consequently, to prevent compile errors, type casts are required in places where these types had been used. An example for a type mismatch is found in the Vector3 class: The fields x, y and z are defined as float in Ros#, but ROS defines them as float64, which maps to the type double in C#.

A fork of Ros# has been created which includes only these changes [2] , and is based on the version committed on April 12, 2019. This fork is only used for the proof of concept. The message generation plugin will function regardless of whether the forked project is used or the ros-sharp master branch.

After completing these steps, the project compiles successfully using the newly-generated messages. An example of a generated message file can be seen in listing 4.1 .

[1] Example project:
https://github.com/Phil-Holdorf/ros-sharp-extension/tree/master/Unity/Proof%20of%20Concept/Project%201

[2] Fork of Ros#: https://github.com/Phil-Holdorf/ros-sharp

```
/*
This message class is generated automatically with 'CustomMessageGenerator' of RosSharpExtension
*/

using Newtonsoft.Json;
using RosSharp.RosBridgeClient;
using RosSharp.RosBridgeClient.Messages;
using RosSharp.RosBridgeClient.Messages.geometry_msgs;

namespace RosSharp.RosBridgeClient.Messages.geometry_msgs {
  public class Twist : Message {
  [JsonIgnore]
  public const string RosMessageName = "geometry_msgs/Twist";

  public Vector3 linear;
  public Vector3 angular;

    public Twist() {
      linear = new Vector3();
      angular = new Vector3();
    }
  }
}
```

Listing 4.1: An auto-generated Twist message class

5 Implementation

The proposed software can be separated into two modules: The ROS side and the Unity side. The ROS module runs on an Ubuntu 16.04 machine, while the Unity module runs on a Microsoft Windows machine. Alternative platforms may be possible but have not been tested for this software. Communication between these two modules happens via RosBridge.

The ROS module consists of a single package 'ros_sharp_extension'. The package contains the services 'list_messages' and 'list_messages_in_package', which are implemented as python nodes. These services expose the functionality of the ROS console commands 'rosmsg list' and 'rosmsg package'. The Unity module is a library named 'RosSharpExtension'. It is a plugin for the Unity Editor which adds the message transfer window as shown in section 4.3 .

The chapter is structured along two communication steps between ROS and Unity that occur in the message transfer process: Listing messages and generating class files. A different ROS service is called in each step, requiring user input between the two service calls. These steps can be broken down further into smaller steps. The focus here lies on explaining the control flow of the application. Several problems that were encountered during development are also highlighted.

5.1 Listing messages

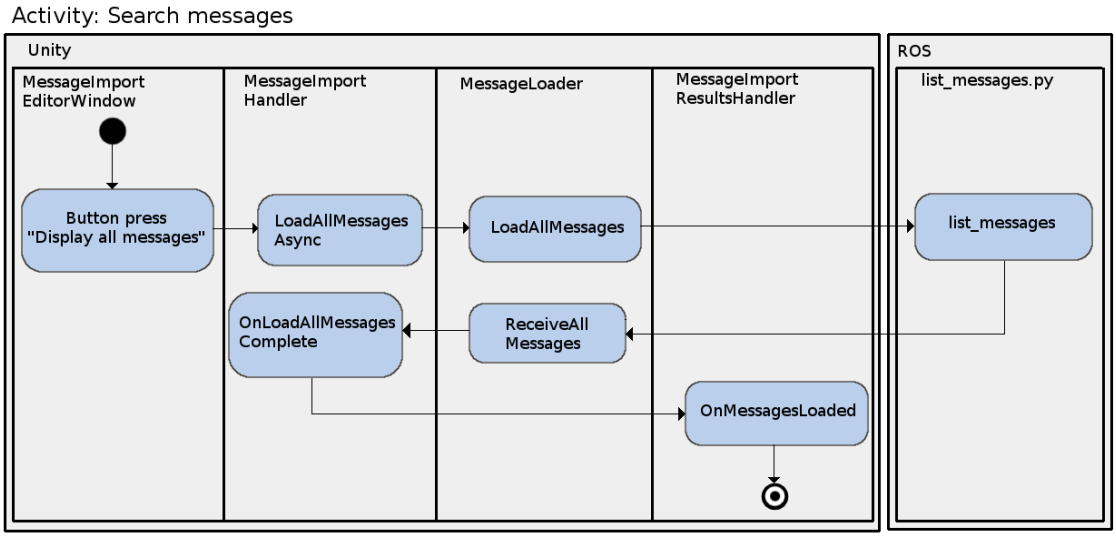

Figure 5.1: Activity diagram illustrating the control flow of a message search operation. Upon pressing a button, the input is passed through a layered class hierarchy and propagated to ROS as a service call to list_messages. The service returns the list of messages to its caller, which is then processed further to be displayed within Unity. For simplicity, side effects of method calls are omitted here. Also not shown is the RosBridge communication layer, because the MessageLoader is an abstraction on it.

Shown in figure 5.1 are all the main scripts involved in the message listing step when choosing the option "Display all messages". The control flow is analogous for the other two search operations: listing messages in a single package, or listing messages of active topics. The methods will have different names, but the execution order is the same. In the following, each of the scripts will be detailed further in regards to their individual responsibilities.

MessageImportEditorWindow: Handling UI layout

This class is responsible for the layout and rendering of the Unity editor window 'Transfer messages from ROS'. It was ensured that the class is essentially stateless. All user input which affects the state of the program is delegated to the classes MessageImportHandler and MessageImportResultsHandler. This separation was done in the interest of maintainability. The window class has access to the state of the two handler classes, and uses this to decide which UI components should be displayed.

MessageImportHandler: Handling UI state

The public state of this class consists of the options selected by the user and the current state of the operation. User input is received from the MessageImportEditorWindow, which changes the state. When pressing one of the 'Load Messages' buttons, an instance of MessageLoader is created in a new thread. A callback method is given to the MessageLoader as an argument, which is invoked when the list of messages is received from the ROS side. The list is then passed to the MessageImportResultsHandler.

MessageLoader: Communicating with ROS

The MessageLoader class is quite simple. It provides methods which initiate the ROS service calls for listing messages. It provides an abstraction on the RosSocket class defined by Ros#. By extension, it also abstract from the communication logic provided by RosBridge. This separation is useful because service calls are rather long-winded, as can be seen in listing 5.1 below.

```
1 rosSocket.CallService
2   <ListMessagesRequest, ListMessagesResponse>
3   ("ros_sharp_extension/list_messages",
4     ReceiveAllMessages,
5     new ListMessagesRequest()
6   );
```

Listing 5.1: The code for making a service call in Ros#. The arguments are the request and response class types, the name of the service, a reference to a callback function which is called on completion, and an instance of the request class.

MessageImportResultsHandler: Organizing results

This class addresses the task of managing the "Results" section of the message import window. It does not access UI components, but it organizes the results and exposes them to the UI layer, as well as offering public methods to alter the state. From the list of messages received by the ROS service, a hierarchy is created, which is represented by a sorted dictionary. This allows grouping messages under their package names and automatically sorting the list. Three more dictionaries are used to store message selection, package selection, and which packages are folded or unfolded in the view.

5.2 Generating class files

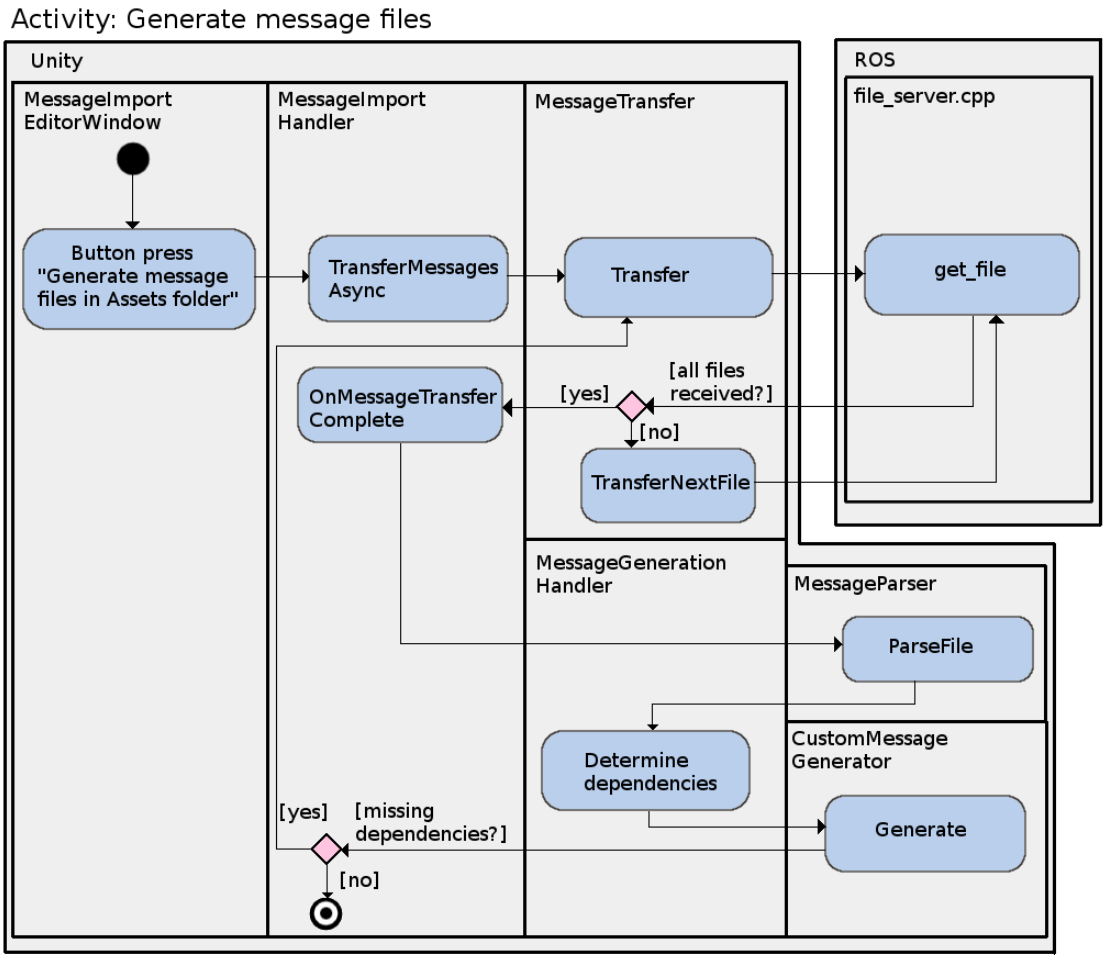

Figure 5.2: Activity diagram of the message generation control flow. A button press initiates the process. The message selection made by the user is passed through to the file server on the ROS side. After all files are transferred, the message generation process begins. Each message file goes through a three-step process: Parsing the file contents, determining if the file depends on other messages, and generating a class file. If there are missing dependencies, the transfer process is repeated until all dependencies are resolved.

The call hierarchy shown in figure 5.2 is similar to the one in the previous section. The important part is to describe what happens after the message transfer has completed. The process of parsing raw message files, resolving dependencies and generating class files will be looked at in detail.

MessageParser: Converting text to objects

A ROS message can be defined by its combined name and a list of fields, where the combined name is a concatenation of the package name and the message name. The package and message name are separated by a slash symbol, as in 'geometry_msgs/Twist'. The types as well as the syntax for message files are defined in the ROS documentation[37] .

Fields in a message are represented by the **CustomMessageElement** class. It consists of the name of the field, a message type denoted by a combined name, an array marker, and a primitive type marker (indicating whether it is a built-in ROS type).

The first step in message parsing is reading each line in a given file, removing the comments, and extracting the individual words. Constants are currently not supported, so if one of the words is an equal sign, the line will be skipped. The expected word count is exactly two: a type name and a field name, separated by a whitespace character. If the words contain squared brackets, it means that the field is an array. A look-up table is then used to find out if the field has a primitive type, as shown below in listing 5.2. If it is primitive, it will lack a package name. If it is not primitive but still lacks a package name, it means that the referenced message type resides in the same package as the message file.

```
"bool" -> bool
"int8" -> sbyte
"uint8" -> byte
"int16" -> short
"uint16" -> ushort
"int32" -> int
"uint32" -> uint
"int64" -> long
"uint64" -> ulong
"float32" -> float
"float64" -> double
"string" -> string
"char" -> byte
"byte" -> sbyte
```

Listing 5.2: Reference for a mapping from primitive ROS message types to equivalent C# types (Pseudo code). This is used by the message parser.

There are some ROS types which require special attention. These are handled on a case-by-case basis. These types are 'time', 'duration' and 'Header'. Firstly, time and duration are built-in ROS types without an equivalent C# type. Instead of using primitives to represent these types, they are represented as separate classes, both containing two integers 'secs' and 'nsecs'. Header is a special type because it does not require the package prefix in a message definition, so it must be added to its combined name.

MessageGenerationHandler: Resolving dependencies

This class is mainly responsible for determining the list of message dependencies for a given file. The naming stems from the fact that it also initiates the file parsing and code generation. Although the implementations for these two steps are offloaded into other classes, which are part of the generation process as a whole.

If the code were to be refactored, this class should be renamed DependencyHandler or similar. It should only define one public method named GetDependencies, receiving as input an already parsed file. Message generation should be initiated in the MessageImportHandler class, after all dependencies have been received and parsed. The dependencies on MessageParser and CustomMessageGenerator should not exist in this class. With these changes, the class would only have a single responsibility, which should help to improve its maintainability. Since the class does provide the list of dependencies as its expected output, this architectural oversight was seen as a low-priority issue.

CustomMessageGenerator: Writing class files

Ros# already provides a class for generating messages, however it was not suitable for this project for several reasons. The first is that it resides in the Unity module, not in the RosBridgeClient library. Referencing the Unity module would be problematic, as it would require our code to be copied into the Unity project folder instead of being pre-compiled. In that case, the message generation could not be used if the Unity project does not compile.

Moreover, the Ros# message generator was not made with extensibility in mind. It serves the user interface that they provide, and can be seen as dependent on it. It also does not provide an option to generate primitive field types. The only types allowed are those deriving from the Message class.

One noteworthy and necessary feature of the message generator is the ability to identify valid C# identifiers. ROS messages may define fields such as 'interface', 'event', or other reserved key words. One way to circumvent this is to add an '@' to the beginning of the identifier in those cases. Apart from that, the task of generating code is fairly trivial, as it is simply a matter of appending strings.

6 Discussion

This chapter addresses the thesis goal described in section 1.2, and the problem summary in 4.1 . The topic of the discussion is how well the problems could be solved, what is still missing or unclear, and how this work could be improved on in the future.

6.1 Evaluation

The goal was to examine Ros# in terms of productivity, and to extend the software, improving it in this regard. To help identify possible improvements, some further questions have been asked:

1. What are some use cases of ROS-Unity applications?
2. Which steps in the development process do these use cases have in common?
3. How can the proposed framework assist to increase productivity at each of these steps?

The first and second question were answered in section 2.2 and section 2.4 , respectively. It was found that the use cases for ROS-Unity interfaces are very diverse. They include a simulator for multiple flying drones, a monitoring task for industrial processes, and several projects related to human-robot interactions (HRI). Among the recent examples found during research, the most common applications were those that utilized virtual reality devices.

The research revealed that a ROS-Unity framework should be applicable to any scenario. On the other hand, the trend towards VR applications is noticeable and could deserve further support by this framework. The shared requirement between these applications was the task of connecting ROS to Unity, which had previously been solved on a case-by-case basis.

The specific steps to achieve this connection were mainly described in section 3.1 , which explained ROS communication concepts. Building on that, sections 3.3 and 3.4 introduced a framework for the Unity-side communication, consisting of RosBridge and Ros#.

There are some common development steps which were left somewhat unaddressed by Ros#. These were described in the problem summary in section 4.1 : When using only Ros#, Unity scripts had to be written manually to implement messages, services, publishers and subscribers. Afterwards, the publishers, subscribers and RosConnector scripts had to be added to appropriate game objects.

The third question was then answered by the proposed features in section 4.2. Time can be saved by automating a part of the development process, thereby increasing productivity. The features were limited to the generation of C# code for ROS message files in Unity. The following sections evaluate if the tool works as intended and describes various issues that still remain.

6.1.1 Functionality

The message generation feature is explained in detail in section 4.3, focusing on the user interface. The proof of concept in section 4.5 shows that the message generation feature is usable. That is, a Unity project will compile and run successfully with the generated message files. In the case of previously existing message files, these can be replaced with generated files as well, but there are some difficulties that arise, which are explained below.

In the process of setting up the proof of concept, it became clear that care should be taken when creating message classes manually. This is because there were inconsistencies between the message types of Ros# and ROS itself, which meant that the code which used these messages had to be updated when the message definitions changed to the generated classes. Although the message generation produces the expected output which is equivalent to ROS messages, these inconsistencies can lead to some development overhead when switching from manually defined classes to generated classes.

When introducing new messages to a project for the first time, this overhead would not occur. In that case, no code would exist which references the new messages, therefore no code has to be updated.

However, the described overhead does not have to be a drawback. It is still possible to use both manually defined as well as generated message types within the same project, as long as it does not create name conflicts with existing types.

For this thesis, it was requested by the advisors that the generated messages are assigned to the same namespace as the Ros# messages. It would be a trivial task to change the output namespaces to prevent such conflicts: The output namespace is simply defined as a string in the message generator class, to which the ROS package name of the message is then appended.

6.1.2 Performance

The process of loading, displaying and generating messages is rather slow. On a weaker machine (1.83Ghz), these steps combined could take up to a few minutes to complete. On a computer which is better equipped to run Unity (3.6Ghz), each step generally takes a few seconds. The source of this bottleneck was not investigated further. The reasoning is that the application is run in the editor and not in game mode, which means it is not impacted by framerate requirements. Moreover, message generation is not intended to be a frequently used operation.

Modern computers should be capable of completing the steps in several seconds, which was deemed acceptable. If performance becomes a concern, Unity's built-in profiling tool could be used to identify and resolve potential bottlenecks. This will give information on resource consumption, time spent in certain code areas, and more.

6.1.3 Testing

No formal code testing was done on this project. The proof of concept can be followed to show functionality in a pre-defined test case. This proof can also be extended to generate a very large amount of messages: None of the message files from the used ROS installation and related components were causing compile errors. However, it was not verified that all of them had in fact been generated successfully and whether they followed the expected output. The components of this installation are referenced in the Ros# wiki tutorials, which was described in section 3.4.2 .

These are only positive test cases and cannot be a replacement for formalized unit tests. Several automated tests come to mind which could have increased confidence in the correctness of the application. The generated message files each have an expected output which could have been tested against the actual output. Test cases could be created for messages which contain one or more fields, and which include arrays, constants and comments, with both legal and illegal C# identifiers.

Unit tests would also reveal behaviors in the case of communication failures and other exceptions such as file system errors. Currently, an error message will be displayed in the Unity console when the ROS machine is unreachable. However, it is possible that the message loading and message generation fails without a message and stops responding, in the case that ROS is reachable but encounters an error.

Ros# has used a system where every step in the file transfer between ROS and Unity was displayed with a status field. This can be seen in the URDF transfer window. This approach can aid debugging efforts when one of the steps fails. However, this approach was deemed too noisy and took up too much space to be included in the message generation window.

Finally, potential users of the software could have been involved at several points in the design and development stages. Users of both ROS and Unity could have been interviewed to help determine the requirements of the tool. Instead, the requirements were determined by researching related work and consulting with two ROS users, who also supervised this thesis. After the implementation phase, users were not asked to use the tool first-hand. The user interface is of such a small scale - a single window with a handful of options - that a user acceptance test was not expected to provide further meaningful insight. If the tool is to be developed further, user feedback should be incorporated to determine the direction of the software and address usability concerns.

6.2 Outlook

As it stands, the software is usable and provides a simple addition to the toolset needed by ROS-Unity developers. While it is a step towards increased productivity in this area, there are still a number of features which would have been desirable, but had to be left out due to time constraints.

6.2.1 Messages

When it comes to message generation, there are a few aspects which are left unaddressed. ROS message files can contain constants and documentation in the form of comments. Both of these are currently not included in generated message files, as they are not necessary for using the messages in a script and communicating with ROS. Including them would mean one less look-up of ROS documentation, which might help Unity developers. In the case of constants, some users might expect to find them inside a message class definition. If they are missing, it could inconvenience users who have used a particular message type before in other projects.

Another known issue related to messages is the lack of support for fixed length arrays, which can be used to represent matrices or other data. This can be seen in the ROS message TwistWithCovariance [38] . Currently, a fixed length array will not be initialized with the correct length in the message constructor. Instead, all array fields are initialized as empty arrays. This can lead to an error if developers try to assign a value at an index without first re-initializing the array with the correct size.

In the event that a message definition changes on the side of ROS, the message file would have to be generated again or be modified manually. To help with that, an automatic update function could be added in the future. ROS provides an Md5 hash for every message type, acting as a unique identifier. This could be used to compare message definitions on a ROS system with those present in a Unity, thereby determining if there are any outdated message files.

It is worth noting that when work begun on this thesis in January, the message generation dialog had not yet been added. This meant that code for new message files had to be written manually, followed by a recompilation of the RosBridgeClient library. As Ros# is still in development, additional work in this area can be expected. To contribute to this effort, the work described in this thesis will likely be offered to Ros# for review and potential integration in some form. Before this can be done, the current developments of the project need to be taken into consideration to prevent any conflicting features.

6.2.2 Other file types

As mentioned in the problem summary in section 4.1 , it would have been desirable to extend the automated code generation feature to other types of files. Namely: Services, publishers and subscribers. For this purpose, new dialog options could be added to the user interface, either as separate windows or consolidated into a single window.

Starting with service files, these could be generated in much the same way as message files. Ros# defines services as two classes in a single file: a request class and a response class, each inheriting from Message. Since the ROS command rosservice can only display currently active services [39] , another node would have to be created which lists all service files nested in a given package.

In the case of publishers and subscribers, this could work by displaying a list of ROS topics, from which users make a selection. However, ROS can only display topics that are currently active [40] . This means that users would need to type the name of each topic, as well as the message type for it. Alternatively, users could launch all the ROS nodes in a system beforehand, thus giving access to the list of active topics. This would save the effort of typing out each topic.

After the topic selection has been made, the related scripts can be generated with simple communication templates. These can be based on the existing Ros# publisher and subscriber scripts as a reference. Since topics each refer to a certain message type, the message classes could be generated in the same step. These scripts would serve as an immediate starting point for users to orient themselves on and add their own custom behaviors.

The task of adding scripts to game objects is likely not suited to being automated, since publishers and subscribers can reasonably be attached to a variety of objects in a Unity scene, such as robots or "manager" objects, or even multiple objects. At most, an option could be offered to specify an existing object to add the scripts to. This could be helpful if the number of topics is quite large, making the configuration a repetitive process.

After adding the features described in this chapter, the software could be considered complete in regards to its purpose of automated file generation. This includes adding unit tests for increased confidence, as well as involving users to provide feedback in the interface design and the selection of new features.

7 Conclusion

The software shown in this work is an extension of the existing ROS-Unity framework Ros#, which is still under active development by Siemens. The added feature is a dialog option which enables users to automatically generate C# code for ROS message files. In the past, these files had to be produced manually, which acted as a barrier to rapid prototyping of various robot setups in Unity.

The functionality of the tool was shown in a proof of concept application, where existing message files were successfully replaced with generated files. Not all the desired features could be completed in time. In the future, the automated code generation should be extended to ROS services, publishers and subscribers. Automated unit tests should be added to increase confidence in the correctness of the results, and user involvement should take more of a focus in the future design process.

While this work marks an improvement and an increase in productivity due to time savings, it is only one step towards the goal of creating a versatile framework for ROS-Unity developers. Users without expert knowledge will still face obstacles that could be smoothed out with a more complete set of features. As it stands, production of ROS source files still remains a conscious task for developers, bringing with it a time investment and a learning curve. Only a part of this task has been succesfully automated.

Ideally, future developments will bridge the gap between experts of the fields of robotics and interactive software, thus allowing more frequent contributions related to human-robot-interaction and robot simulation. In the author's opinion, it is likely that Ros# will become the dominant choice for ROS-Unity developers, and contributions are likely to increase.

Bibliography

[1] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[2] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, volume 3, pages 2149–2154. IEEE, 2004.

[3] Unity Technologies. Unity3d. https://unity.com, Retrieved June 19, 2019.

[4] Siemens. Rossharp. https://github.com/siemens/ros-sharp, Retrieved June 19, 2019.

[5] Yuchao Hu and Wei Meng. Rosunitysim: Development and experimentation of a real-time simulator for multi-unmanned aerial vehicle local planning. *Simulation*, 92(10):931–944, 2016.

[6] Enrico Sita, Csongor Márk Horváth, Trygve Thomessen, Péter Korondi, and Anthony G Pipe. Ros-unity3d based system for monitoring of an industrial robotic process. In *2017 IEEE/SICE International Symposium on System Integration (SII)*, pages 1047–1052. IEEE, 2017.

[7] Dennis Krupke, Lasse Einig, Eike Langbehn, Jianwei Zhang, and Frank Steinicke. Immersive remote grasping: realtime gripper control by a heterogenous robot control system. In *Proceedings of the 22nd ACM Conference on Virtual Reality Software and Technology*, pages 337–338. ACM, 2016.

[8] Christopher Crick, Graylin Jay, Sarah Osentoski, Benjamin Pitzer, and Odest Chadwicke Jenkins. Rosbridge: Ros for non-ros users. In *Robotics Research*, pages 493–504. Springer, 2017.

[9] Brian Gerkey, Richard T Vaughan, and Andrew Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the 11th international conference on advanced robotics*, volume 1, pages 317–323, 2003.

[10] Fumio Kanehiro, Hirohisa Hirukawa, and Shuuji Kajita. Openhrp: Open architecture humanoid robotics platform. *The International Journal of Robotics Research*, 23(2):155–165, 2004.

[11] Olivier Michel. Cyberbotics ltd. webots™: professional mobile robot simulation. *International Journal of Advanced Robotic Systems*, 1(1):5, 2004.

[12] Eric Rohmer, Surya PN Singh, and Marc Freese. V-rep: A versatile and scalable robot simulation framework. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1321–1326. IEEE, 2013.

[13] Benjamin Balaguer, Stephen Balakirsky, Stefano Carpin, Mike Lewis, and Christopher Scrapper. Usarsim: a validated simulator for research in robotics and automation. In *Workshop on Robot Simulators: Available Software, Scientific Applications, and Future Trends at IEEE/RSJ*. Citeseer, 2008.

[14] Epic Games. Unreal engine. https://www.unrealengine.com, Retrieved June 25, 2019.

[15] Stephen Balakirsky and Zeid Kootbally. Usarsim/ros: A combined framework for robotic control and simulation. In *ASME/ISCIE 2012 international symposium on flexible automation*, pages 101–108. American Society of Mechanical Engineers, 2012.

[16] Jeff Craighead, Jennifer Burke, and Robin Murphy. Using the unity game engine to develop sarge: a case study. In *Proceedings of the 2008 Simulation Workshop at the International Conference on Intelligent Robots and Systems (IROS 2008)*, 2008.

[17] Gilberto Echeverria, Nicolas Lassabe, Arnaud Degroote, and Séverin Lemaignan. Modular open robots simulation engine: Morse. In *2011 IEEE International Conference on Robotics and Automation*, pages 46–51. Citeseer, 2011.

[18] Blender. http://blender.org, Retrieved June 19, 2016.

[19] Daniel Cook, Andrew Vardy, and Ron Lewis. A survey of auv and robot simulators for multi-vehicle operations. In *2014 IEEE/OES Autonomous Underwater Vehicles (AUV)*, pages 1–8. IEEE, 2014.

[20] Mario Prats, Javier Perez, J Javier Fernández, and Pedro J Sanz. An open source tool for simulation and supervision of underwater intervention missions. In *2012 IEEE/RSJ international conference on Intelligent Robots and Systems*, pages 2577–2582. IEEE, 2012.

[21] Farzan M Noori, David Portugal, Rui P Rocha, and Micael S Couceiro. On 3d simulators for multi-robot systems in ros: Morse or gazebo? In *2017 IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*, pages 19–24. IEEE, 2017.

[22] Weichao Qiu and Alan Yuille. Unrealcv: Connecting computer vision to unreal engine. In *European Conference on Computer Vision*, pages 909–916. Springer, 2016.

[23] Christoph Bartneck, Marius Soucy, Kevin Fleuret, and Eduardo B Sandoval. The robot engine—making the unity 3d game engine work for hri. In *2015 24th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)*, pages 431–437. IEEE, 2015.

[24] R Codd-Downey, P Mojiri Forooshani, A Speers, H Wang, and M Jenkin. From ros to unity: Leveraging robot and virtual environment middleware for immersive teleoperation. In *2014 IEEE International Conference on Information and Automation (ICIA)*, pages 932–936. IEEE, 2014.

[25] Juan Jesús Roldán, Elena Peña-Tapia, David Garzón-Ramos, Jorge de León, Mario Garzón, Jaime del Cerro, and Antonio Barrientos. Multi-robot systems, virtual reality and ros: Developing a new generation of operator interfaces. In *Robot Operating System (ROS)*, pages 29–64. Springer, 2019.

[26] Michael Jenkin and Mathias Ciarlo. Rosbridgelib. https://github.com/MathiasCiarlo/ROSBridgeLib, Retrieved June 28, 2019.

[27] David Whitney, Eric Rosen, Daniel Ullman, Elizabeth Phillips, and Stefanie Tellex. Ros reality: A virtual reality framework using consumer-grade hardware for ros-enabled robots. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1–9. IEEE, 2018.

[28] Ros reality github page. https://github.com/h2r/ros_reality, Retrieved June 29, 2019.

[29] Yoshiaki Mizuchi and Tetsunari Inamura. Cloud-based multimodal human-robot interaction simulator utilizing ros and unity frameworks. In *2017 IEEE/SICE International Symposium on System Integration (SII)*, pages 948–955. IEEE, 2017.

[30] D Krupke, S Starke, L Einig, F Steinicke, and J Zhang. Prototyping of immersive hri scenarios. In *International Conference on Climbing and Walking Robots and the Support Technologies for Mobile Machines*, pages 537–544. World Scientific, 2017.

[31] Ros wiki documentation. http://wiki.ros.org, Retrieved July 06, 2019.

[32] Unity scripting reference. https://docs.unity3d.com/2019.2/Documentation/ScriptReference/index.html, Retrieved July 29, 2019.

[33] Rosbridge wiki page. http://wiki.ros.org/rosbridge_suite, Retrieved June 19, 2016.

[34] Rosbridge protocol definition. https://github.com/RobotWebTools/rosbridge_suite/blob/groovy-devel/ROSBRIDGE_PROTOCOL.md, Retrieved July 29, 2019.

[35] Ros# wiki, instructions for an example application.

[36] Turtlebot. https://www.turtlebot.com, Retrieved July 29, 2019.

[37] Ros message documentation. http://wiki.ros.org/msg, Retrieved July 29, 2019.

[38] Message definition for twistwithcovariance. http://docs.ros.org/jade/api/geometry_msgs/html/msg/TwistWithCovariance.html, Retrieved July 29, 2019.

[39] Rosservice command documentation. http://wiki.ros.org/rosservice, Retrieved July 29, 2019.

[40] Rostopic command documentation. http://wiki.ros.org/rostopic, Retrieved July 29, 2019.

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Bachelorstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel — insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen — benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Hamburg, den 30.07.2019 Phil Holdorf

Veröffentlichung

Ich stimme der Einstellung der Arbeit in die Bibliothek des Fachbereichs Informatik zu.

Hamburg, den 30.07.2019 Phil Holdorf