TECHNISCHE UNIVERSITÄT WIEN

AUTOMATION & CONTROL INSTITUTE
INSTITUT FÜR AUTOMATISIERUNGS- & REGELUNGSTECHNIK

Development of a graphical user interface for programming ROS-based robots

MASTER THESIS

Conducted in partial fulfillment of the requirements for the degree of a

Diplom-Ingenieur (Dipl.-Ing.)

supervised by

Ao.Univ.-Prof. Dipl.-Ing. Dr. techn. Markus Vincze

submitted at the

TU Wien

Faculty of Electrical Engineering and Information Technology
Automation and Control Institute

by

Alexander Semeliker, BSc

Vienna, July 2019

Vision for Robotics Group
A-1040 Wien, Gusshausstr. 27, Internet: http://www.acin.tuwien.ac.at

Acknowledgement

I dedicate this work to my parents, who sadly passed away during writing this thesis. I am eternally grateful for all they have done for me and every moment I could spent with them.

Vienna, July 2019

Abstract

Teaching robot programming methods and abstracting the complex implementation to a level, where very little technical expertise is required are two key points of the field of robotics. This work presents the design, implementation and evaluation of Rockly, a web-based graphical user interface for programming ROS-based robots. Robot Operating System (ROS) is a robotics middleware and is considered as the de-facto standard framework for robot software development. The second fundamental framework used for development is Blockly - an open-source JavaScript library made by Google, which provides a visual code editor and a code generation interface for web applications. Programming then is done by simply connecting blocks. Uniting these frameworks is the one of the key features of Rockly, hence it enables users to easily deploy and run their applications on many different platforms and robots. Rockly - a portmanteau of ROS and Blockly - provides three key components: an interface to manage code blocks, an interface to create demos, and an interface to manage demos. The interface was implemented and tested on the HOBBIT PT2 robot platform as well as evaluated by moderate experienced robot developers. With respect to the results of it, it can be concluded, that Rockly enabled all partcipants to work more continuously to find a tool supported solution significantly faster than by coding, which in most cases was executable and correct.

Kurzzusammenfassung

Das Lernen von Roboterprogrammiermethoden und das Abstrahieren der direkten, mittlerweile sehr komplexen Implementierung auf ein Level, das sehr wenig technische Expertise voraussetzt sind zwei aktuelle Diskussions- und Forschungspunkte im Robotikbereich. Im Rahmen dieser Diplomarbeit wurde eine web-basierte grafische Benutzeroberfläche - genannt Rockly - entwickelt, implemtiert und evaluiert, die Lösungsansätze dafür bieten soll und mit Hilfe derer ROS-basierte Roboter programmiert werden können. ROS ist eine Middleware und de facto das Standard-Framework in der Entwicklung von Robotik-Software. Das zweite Framework, das dem entwickeltem Tool zugrunde liegt, ist Blockly - eine von Google entwickelte Open-source JavaScript Bibliothek für Web-Applikationen, die eine grafische Programmieroberfläche sowie eine Schnittstelle bereitstellt, mit der aus ebenjener grafischen Oberfläche ein ausführbarer Code generiert werden kann. Ein Programm besteht damit lediglich aus Blöcken, die zusammengefügt werden. Durch die Verwendung dieser beiden Frameworks ist es möglich, dass Rockly, dessen Name sich aus den beiden Frameworks ableitet, auf vielen unterschiedlichen Plattformen eingesetzt werden. Es stellt drei Dienste bereit: eine Oberfläche zum Verwalten der Programmierblöcke, eine Oberfläche zum Erstellen von Programmen und eine Oberfläche zum Verwalten von erstellten Programmen. Die entwickelte Oberfläche wurde auf der HOBBIT PT2 Roboterplattform implemtiert und im Rahmen einer Evaluierungsstudie von angemessen erfahrenen Robotik Entwicklern bewertet. Auf Grundlage der Ergebnisse lässt sich zusammenfassen, dass Rockly es allen Teilnehmern ermöglichte kontinuierlicher und signifikant schnellereine Lösungen zu finden, verglichen zum herkömmlichen Programmiern, die in den meisten Fällen ausfÃ¼hrbar und korrekt waren.

Contents

List of Figures

List of Tables

Listings

1 Introduction

The significance of discussions about autonomous systems and robotics have been rising constantly within recent years. There are lots of supporters and lots of skeptics about it. People get exicted about the technical progress, approaches and possible application fields as well as feeling concerned about the future labour market. Independently from their point of view, most of them are interested in hands-on experience with robots and get in touch with them. Fairs are one of the most common meeting places for people in order to see robots in action and get in touch. From the developer's and presenter's sides of view this requires to prepare different use cases to demonstrate a robots' capabilities to the interested parties. Creating different behaviours periodically requires time not every company or team could effort. Another big contact point between society and robotics can be found in education, where students often make their first steps in programming using proper robots and interfaces.

1.1 Motivation

As the field of interest is getting bigger and bigger it is necessary to provide an accessible form of interacting with robots and programming them. The de-facto standard framework for robot developing is ROS[1], which provides different communication patterns and standardized interfaces. The mightiness of it faces the fact that not only programming skills are necessary to bring robots to life, but also a detailed knowledge of the robot's architecture itself is required. For minimizing the necessary knowledge transfer in the above mentioned academic institution case a more abstract interface to prepare robots for fairs would be advantageous. Also students would benefit from such a tool since commonly used robots (e.g. Nao[2] or Pepper[3]) come up with specific programming softwares, which provide different handlings and often are limited to very few different platforms.

[1] http://wiki.ros.org/
[2] https://www.softbankrobotics.com/emea/en/nao
[3] https://www.softbankrobotics.com/emea/en/pepper

The main problem of a properly fitting interface is to maintain the powerful capabilities of ROS while providing it in a accessible way. The first solution, which maybe comes into most people's mind - especially when thinking of educational robotics - may be a graphical User Interface (UI). Such an approach would result in a very high abstraction level, which requires a clever design. On the other hand, a more abstract Application Programming Interface (API) based on ROS communication patterns could be an option. This would enable users to create more complex applications and demos, but perhaps limit the accessibility for non-technicians and robot newbies. Additionally, designing such a customized API would require a higher implementation effort since it must prevent the code from crashing if it is used wrongly and should be provided for Python and C++ as these are the common ROS programming languages. Independently from the solution's design it should also provide user friendly maintainability and the option to further improve it without the necessity to understand each development step and line of code.

1.2 Problem Statement

Scope of this work is to design an interface in order to abstract the implementation process of ROS-based robots, thereby minimizing the familiarization period when first working with a robot, such as gathering knowledge about communication interfaces and patterns. Additionally, the proposed design should be implmented on the HOBBIT PT2 platform (see Figure 1.1) with at least the following tasks should be available:

- Move HOBBIT in straight directions
- Turn HOBBIT a given angle
- Move HOBBIT's head
- Control HOBBIT's eyes
- User interaction via HOBBIT's tablet
- Control HOBBIT's arm
- Navigate HOBBIT to a given location

The interface should be designed such as it can be used remotely and can be transferred to any other ROS-based robot in a user-friendly way. This means, that the required knowledge about the architecture of the interface itself - such as underlying frameworks - should be minimized. Finally, the interface should be evaluated with a proper method.

Figure 1.1: HOBBIT - The Mutual Care Robot

1.3 Proposed Solution

To tackle the above mentioned problems, this thesis presents the design, implementation and evaluation of Rockly, a graphical UI. The proposed architecture is shown in Figure 1.2.

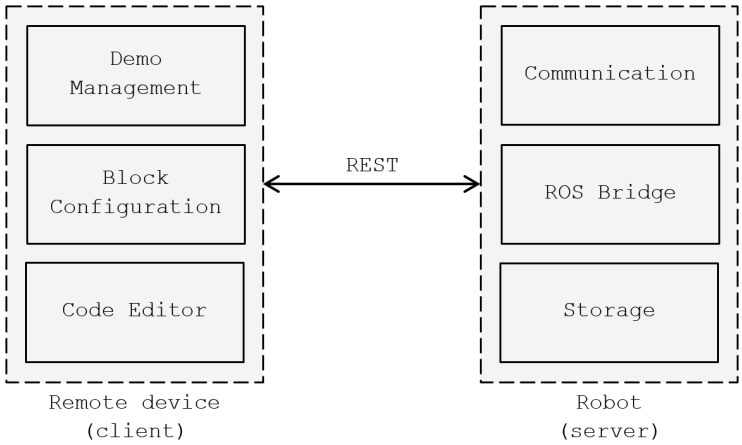


Figure 1.2: Architecture Overview and essential components

In order to not require the user to install any additional software a web-based application is proposed, meaning the source code is deployed at the robot. It operates as a server delivering all the front end functionality to the remote programming devices as well as taking responsibility of code execution and

communication tasks. On the client side the user is able to create programs using a graphical editor, which is built upon Blockly[4] - a library that adds a visual code editor to web applications. Programming then is done by simply connecting blocks. The block configurator enables the user to create new blocks, for example when using a new robot. Finally, a demo management provides a handy way to manage existing programs, such as directly running or deleting them. On the back end side the most integral part is the ROS bridge module, which should ensure ROS connectivity in order to directly run the code from the client side and provide functionality for each ROS communication pattern. This module is also responsible for enabling the tool to be transferred to any other ROS-based robot. The server and client communication is done via Representational State Transfer (REST)[1].

Blockly provides code generation for Python, meaning it is not necessary to create blocks for basic programming functionality (e.g loops, variables, functions and conditional statements). The implementation of generic connectivity is within the scope of this thesis. For convenient usage and scalability reasons this functionality is wrapped into a Python module, which can be exported and used in any other suitable way. In order to enable HOBBIT to the above mentioned tasks, blocks for each of them are implmented. An exemplary workflow is shown in Figure 1.3. First, the program is created via connecting blocks using the graphical editor. Using Blockly the Python code is generated and send to the robot. Using the ROS bridge functionality the code is then executed.

Figure 1.3: Workflow: Graphical programming to code execution

The evaluation of the UI is achieved by conducting a study with people, who have at least moderate experience with ROS and the required programming languages (C++/Python). It should provide answers to the questions, whether such a UI is seen as an improvement compared to the traditional coding approach (in terms of time efficeiness, flexibility and scalability), the workflow is more streamlined and the designed UI is intuitive.

[4]https://developers.google.com/blockly/

1.4 Chapter Organization

This thesis is structured into the following six chapters. After this chapter, in Chapter 2 related work regarding the present problem is discussed. A brief overiew of the milestones and types of Visual Programming Languages (VPLs) is presented as well as some common frameworks and Integrated Development Environments (IDEs). Then the contribution of the tool development during this thesis is analysed. Chapter 3 explains different approaches and the architectural decision. Then the implementation is presented in Chapter 4, covering each component separately. The evaluation process is explained in Chapter 5 and presents the design of the conducted experiment. Then its results are presented and discussed. Finally, in Chapter 6 summarizes the thesis and gives directions for future work.

2 Related Work

This chapter examines the related work of fields this work is linked to. It starts with a brief summary of the fundamental development of VPLs, then presents a classification of them and provides some representative examples. In the next section UIs for programming robots are summarized. Then an overview of already existing graphical environments for ROS-based robots is given. Finally, a comparison between matured interfaces with the one presented in this work is presented.

2.1 Visual Programming Languages

There were basically two milestones in the development of a VPL as we know it nowadays. The first milestone was Sketchpad, presented by Ivan Sutherland ([2]). It was designed in 1963 on the TX-2 computer at MIT (Massachusetts Institute of Technology) and has been called the first computer graphics application. The system allowed users to work with a lightpen to create 2D graphics by creating simple primitives, like lines and circles, and then applying operations, such as copy, and constraints on the geometry of the shapes. Its graphical interface and support for user-specifiable constraints stand out as Sketchpad's most important contributions to visual programming languages. By defining appropriate constraints, users could develop structures such as complicated mechanical linkages and then move them about in real time.[3] David Canfield Smith achieved the next major, groundbreaking step in the history of VPLs. In his PhD dissertation [4] he introduced both the use of small pictorial objects, called icons, and the notion of programming by demonstration.

The field of VPLs has grown rapidly in recent years and therefore more and more interest has been focused on creating a robust, standardized classification for work in this area. A lot of frameworks and environments are established in different fields of use, including education, multimedia, video games, simulation and automation processes. Since there are different focus points across these fields, the usage and design varies in a broad spectrum and it is reasonable to classify them. As [3] outlines it is possible to cluster them according to the way users interact with them - e.g. differing *purely visual languages* and *hybrid text*

and visual systems. Another appropiate classification scheme is the following, which focuses more on data- and input processing:

- Block-based languages
- Flowcharts
- Dataflow programming languages

2.1.1 Block-based languages

Block-based VPLs are especially used in tools for reducing barriers for non-programmers or for teaching children programming. The core of these languages is a set of blocks, which can be connected with each other to form an executable program. Each block represents a specific paradigm of the proper programming language. A typical editor consists of a toolbox which contains the code blocks and a workspace, where the blocks can be placed - usually via a drag-and-drop, which is a key feature and make these VPLs considered most intuitive and user-friendly.

The field of application of block-based VPLs covers a wide range: Ardublock[1] is a graphical programming add-on to the default Arduino IDE - an open-source electronics platform based on easy-to-use hardware and software, which has a broad community. The MIT App Inventor is a drag-and-drop visual programming tool for designing and building fully functional mobile apps for Android.[5] In the fields of education, Scratch[2], developed in 2007, plays an important role, since it is designed to be highly interactive. The name highlights the idea of tinkering, as it comes from the scratching technique used by hip-hop disc jockeys. In Scratch programming, the activity is similar, mixing graphics, animations, photos, music, and sound. The scripting area in the Scratch interface is intended to be used like a physical desktop (see Figure 2.1)[6].

2.1.2 Flowcharts

These VPLs are able to translate an algorithm's logic given as flowchart into executable machine instructions. According to [7] a flowchart is the graphical representation of a process or the step-by-step solution of a problem, using suitably annotated geometric figures connected by flowlines for the purpose of designing or documenting a process or program. Flowcharts are typically used in the fields of automation. Grafcet ([8]) is a tool, drawing its inspiration from

[1] http://blog.ardublock.com/
[2] https://scratch.mit.edu/

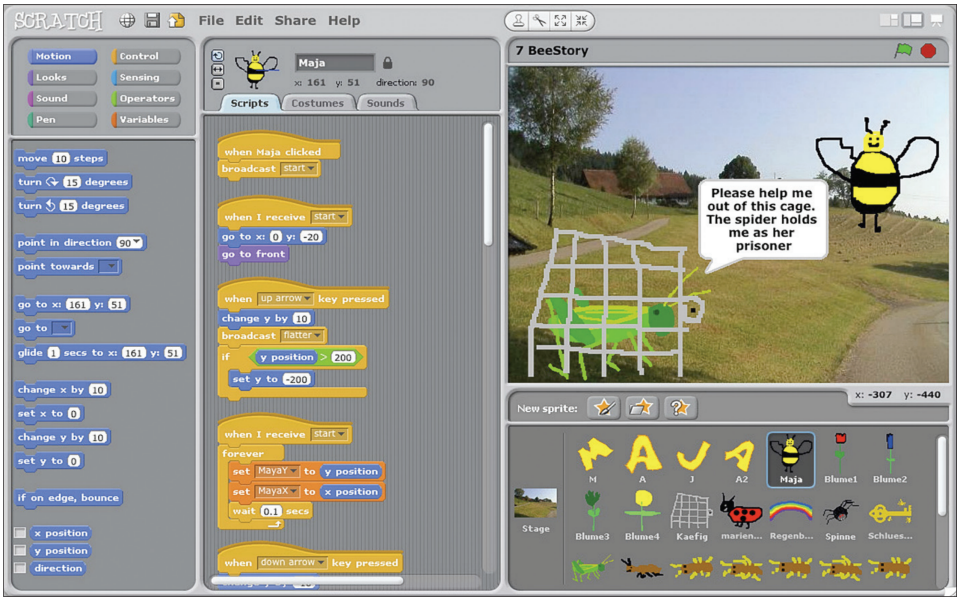


Figure 2.1: Scratch user interface (Source:[6])

Petri nets (a general purpose mathematical tool allowing various discrete-event systems to be described), whose aim is the specification of Programmable Logic Controllers (PLCs). It is the basis of the Sequential Function Chart (SFC), an International Standard in 1987. An implementation of the Grafcet norm can be found in S7-GRAPH, developed by Siemens AG and part of their STEP 7 software for programming PLCs (see Figure 2.2).

Another application using a flowchart VPL is KTechLab[3], which is an IDE for microcontrollers and electronics. It supports circuit simulation, program development for microcontrollers and simulating the programmed microcontroller together with its application circuit.

RoboFlow is a flow-based VPL which allows programming of generalizable mobile manipulation tasks [9]. The authors applied a technique called texitProgramming by Demonstration from the field of End-User Programming (EUP) to robot programming. They also presented an implementation on the PR2[4] robot platform using ROS communication.

[3]https://sourceforge.net/projects/ktechlab/
[4]http://www.willowgarage.com/pages/pr2/overview

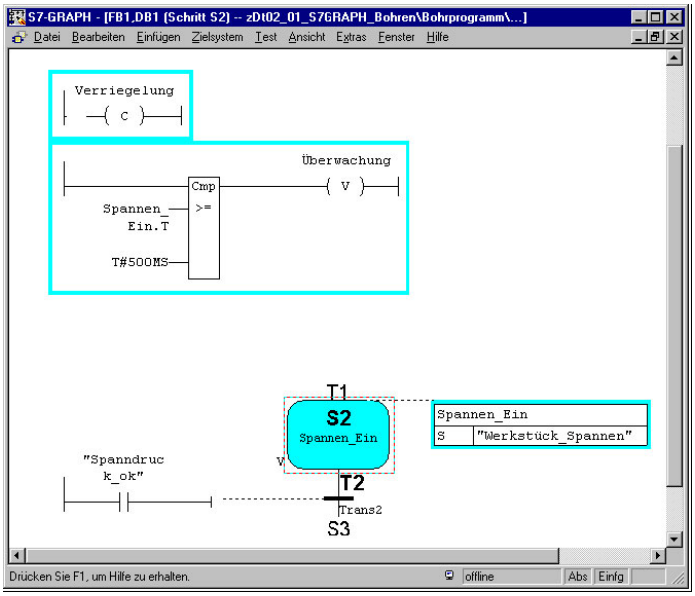


Figure 2.2: S7-GRAPH interface of STEP 7 (Source [5])

2.1.3 Dataflow programming languages

Similar to flowcharts dataflow programming is used for professional applications primarily, aimed at designers rather than end users or coding newbies. Dataflow programming is a programming paradigm whose execution model can be represented by a directed graph, representing the flow of data between nodes, similarly to a dataflow diagram. Considering this comparison, each node is an executable block that has data inputs, performs transformations over it and then forwards it to the next block. A dataflow application is then a composition of processing blocks, with one or more initial source blocks and one or more ending blocks, linked by a directed edge.[10] National Instruments LabVIEW [11] and Simulink (Figure 2.3) can be mentioned as the representatives of this big group.

2.2 Graphical Robot Programming Environments

The driving force of VPLs in terms of graphical programming of robots is education. Teachers rely on simple educational robots and intuitive programming environments. Therefore, graphical programming environments have become a frequent starting point for young students. The used environments mostly

[5] https://w3.siemens.com/mcms/simatic-controller-software/en/step7/simatic-s7-graph/pages/default.aspx

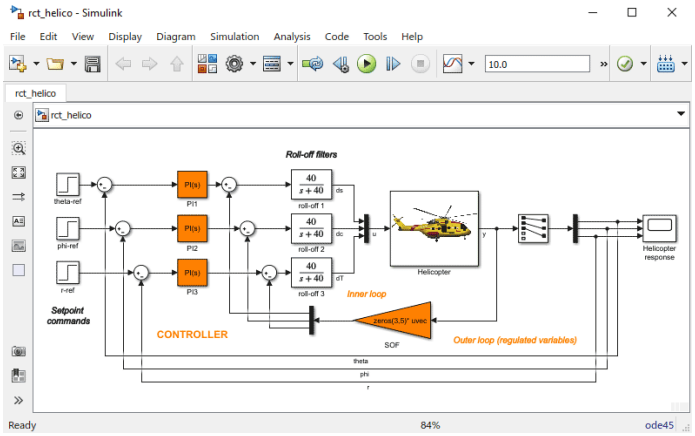


Figure 2.3: User interface of Simulink to use simulation models (Source[6])

depends on the robot the corresponding class is programming and therefore there are many offerings.

A famous environment is the *Lego Mindstorms* system[7] - a hardware and software platform for the development of programmable robots based on Lego building bricks. It comes up with an IDE that is available on Windows PC or Mac. Its programming software is based on LabVIEW and provides the ability to downloading programs to the programmable brick, which can be connected to different sensors and actuators. Besides of the default editor the platform also supports the use of third-party environments as outlined in [12].

Dataflow oriented VPLs, like used in the mentioned Lego Mindstorms system, may be not suitable for absolute beginners, as described in [13]. The author mentioned, that the procedural approach where a program is firstly considered as a sequence of statements is much easier to learn than the data flow or functional approach. Therefore a new graphical programming environment, called *Grape*, was developed. With its help a flowchart can be built and the meaning of the individual elements of the flowchart are defined. It comes up with a list of predefined classes for robot programming and provides the feature to extend the list by own classes using a simple Extensible Markup Language (XML) syntax. The code generation itself also uses XML representation: first the graphical structure of the program is converted into a XML tree, which then is translated into C++ code via a mapping schema.

One of the most powerful graphical robot programming environment is the

[6]https://www.mathworks.com/products/simulink.html
[7]https://www.lego.com/en-us/mindstorms

Open Roberta platform [14]. The connection to the user is called *Open Roberta Lab*, a cloud-based application, which enables children and adolescents to visually program real robot hardware directly from the web browser or by using a build in online robot simulator. It also provides platform features like user login, program saving/sharing and easy hardware pairing over Wi-Fi as well as USB and Bluetooth connection.[15] Its programming language is called *NEPO* and is built up on Google's Blockly, which also plays a fundamental part in this work (see Section 3.2.3 and Section 4.4).

When it comes to humanoid robots, by now, there are three major representatives: Nao[16], Pepper[8] and Romeo[9]. All are developed by SoftBank Robotics and come up with a powerful Software Development Kit (SDK) called *NaoQi*. Besides of SDKs for Python, C++, Java, JavaScript and ROS, the framework also provides a graphical programming tool: Choregraphe[17]. Actually, Choregraphe is a specific module of NaoQi and more or less just a graphical representation of NaoQi's functions. The Choregraphe module produces an XML file describing the program (i.e. the application, the boxes, the connections between them, the included scripts etc.). For execution, the file is interpreted by the XML module of NaoQi. Since the architecture of humanoid robots is very complex, even experienced programmers, at least partly, rely on interfaces like Choregraphe and therefore it is not exclusively used for education.

Another hybrid robot programming environment is provided by Aseba Studio[10] IDE of the Thymio II, a small and low-priced educational robot. It provides pure visual programming, two block based VPLs (Scratch, Blockly) as well as an editor and an API for text programming. It is also possible to connect the robot and directly run and debug the program via the user interface.

2.3 Environments for ROS-based robots

As stated in the beginning, ROS is considered as the de-facto standard framework for robot software development. Nonetheless the environments explained in Section 2.2, as well as other frameworks too, do not provide ROS connectivity out-of-the-box. This might be reasonable, considering purchasing a robot is not cheap. If a user needs to control just one robot, there is no necessity to search for a generic tool - especially because such a tool most likley would not

[8] https://www.softbankrobotics.com/emea/en/pepper
[9] https://projetromeo.com/
[10] https://www.thymio.org/en:asebastudio

provide as powerful capabilities as the default programming interface would do. Assuming the case of a lab, where several people might work with different robots or just quick showcases are desired, it is reasonable to design such an interface. Right now, there a only a few frameworks available for such purposes.

Erle Robotics developed a web-based visualization and block programming tool, called *robot_blockly*, which supports their own robots and drones [18]. It comes pre-installed with their Linux-based artificial robotic brain *Erle-Brain*[11] and also provides a rudimentary interface for all other robots. It uses the standard block creation process of Blockly (Section 3.2.3). Therefore the user needs to update different files of the source code, must follow some naming rules, recompile Blockly and the package and also needs - apart from ROS - basic JavaScript knowledge. The ROS connectivity and all execution routines can be implemented in a Python file, which is read from a specific directory and then included into the Blockly source code.

Another Blockly-based tool for controlling ROS-based robots is the *evablockly_ros* package, developed by Inovasyon Muhendislik[19]. The provided blocks are mainly created specifically for operations that can be performed by using evarobot, a mobile robot platform built by the author. Apart from them the package also provides the following basic ROS connectivity features:

- set up server connection
- create a publisher to send data
- send data via created publisher
- create a subscriber to receive data
- perform operations when data is received from determined subscriber

2.4 Comparison of visual programming tools

This section provides a comparison of this work with some visual programming tools presented in the previous sections in respect of their features. The following systems were considered: robot_blockly[20], Choregraphe[17], Open Roberta Lab[15], Grape[13] and EV3[21]. The specified features target two important fields of the tool decision process - platform independency and usablity. Those two main features are broken down to several minor criteria, as Table 2.1

[11] http://docs.erlerobotics.com/brains/erle-brain-3

shows. All of the mentioned tools come with an IDE for multiple computer platforms and operation systems.

Most of the tools provide a block-based programming interface, which tends to be the simplest VPL type in terms of usablity. Conclusively, the two most popular educational programming platforms, EV3 - the newest generation of the Lego Mindstorms system - Open Roberta Lab, are included. Since EV3 also provides the ability to use third-party environments, all VPLs are supported as well as pure coding itself. When looking at the provided programming options of Choregraphe, it is highlighted that it is possible to create complex programs with it, but getting there requires more technical expertise. The tool presented in this work, Rockly, trys to reach both audiences (i.e. novice and experienced users).

Open Roberta Lab supports seven different robot platforms out of the box, which is significantly more than the other tools. Besides its own programming brick, EV3 supports its preceding robot system NXT partially at the moment, but allows the integration of third-party sensors and motors. Choregraphe Suite supports platforms providing the NaoQi SDK, which are currently three physical robots. All other tools, including Rockly, only provide out-of-the-box connectivity of one robot, though, but are able to be upgraded via different interfaces. Only Rockly comes up with a graphical assistant, which guides the user through the upgrade process, and does not require any rebuild process. The other tools require the user to touch or recompile the source code in order to connect further robots.

Besides the number of robots, which can be connected to a system, platform independency also means, that different operating systems are supported. Web-based applications are the most independent ones, since users are not required to install software on their computers. Running the infrastructure on a server allows users to connect to the robot remotely and use it on mobile devices too. EV3 currently supports mobile devices with the following minimum versions: Android 4.2, iOS 8.0. Of the evaluated tools Rockly, robot_blockly and Open Roberta Lab are fully web-based. All of them are based on Google's Blockly framework.

Development support interfaces such as a robot simulator and a debugging tool are only provided by three tools. When comparing this to the programming types, a pattern could be obtained. One of the advantages using a block-based VPL - such as Rockly - is that the translated code is always syntactically correct, so a major reason using such support during development, is obsolete.

When it comes to ROS connectivity the classification outlined in Section 2.2 and Section 2.3 can be applied. It is possible to create ROS nodes with each tool - except EV3. It is worth a remark that Choregraphe's NaoQi SDK provides ROS connectivity only via code represention, i.e. it is not possible to use ROS communication within the graphical programming interface. Of the tools enabling updates for both ROS connectivity and robot platforms, all provide a manual, but only the workflow of Rockly is tool assisted. This means that upgrading is more of a configuration than programming process - depending on how complex the desired block should be. Therefore less programming knowledge is required - especially in terms of the communication patterns of ROS. Not touching the source code also means no recompiling, which therefore leads to the fact, that - once deployed - Rockly runs autonomously without any internet connection required.

Feature		Rockly	robot_blockly	Choregraphe	Open Roberta Lab	Grape	EV3
Programming	block-based	✓	✓		✓		✓
	flowchart					✓	✓
	dataflow			✓			✓
	code	✓		✓			✓
Robot platforms	out-of-the-box	1	1	1	7	1	2[a]
	upgradeable	✓	✓		✓	✓	
User interface	upgrade assistant	✓					
	web-based	✓	✓		✓		
	simulator			✓	✓		
	debugging			✓			✓
ROS connectivity	out-of-the-box	✓	✓				
	upgradeable			(✓)[b]	✓	✓	
Upgrade workflow	tool assisted	✓		-			-
	manual	✓	✓	-	✓	✓	-
	programming languages	1	2	-	2	2	-
	recompiling		✓	-	✓		-

Table 2.1: Comparison of visual programming platforms

[a] third-party sensors and motors are supported
[b] not for visual programming environment

3 Architecture

This chapter describes the architectural design of Rockly (portmanteau of ROS and Blockly). First the requirements are presented, then the purpose and need of the tool are explained. Based on these constraints the options implemtenting the tool are presented as well as an explanation of the final design. Then an overview of the architecture is given, followed by a short description of all supporting frameworks and dependencies, which are: the robot operating system ROS with its concepts and some JavaScript frameworks, which eased the implementation effort.

3.1 Requirements

In the field of software engineering constraints are the basic design parameters. Therefore, it is necessary to provide them as detailed as possible. In the given case the basic constraints are given by the purpose of the tool and the architecture of the robot.

3.1.1 HOBBIT - The Mutual Care Robot

The HOBBIT PT2 (prototype 2) platform was developed within the EU project of the same name. The robot was developed to enable independent living for older adults in their own homes instead of a care facility. The main focus is on fall prevention and detection. PT2 is based on a mobile platform provided by Metralabs. It has an arm to enable picking up objects and learning objects. The head, developed by Blue Danube Robotics, combines the sensor set-up for detecting objects, gestures, and obstacles during navigation. Moreover, the head serves as emotional display and attention center for the user. Human-robot interaction with Hobbit can be done via three input modalities: Speech, gesture and a touchscreen. [22][23][24]

In terms of technology HOBBIT is based on the robot operating system ROS (Section 3.4.1), which allows easy communication between all components. The system is set up to be used on Ubuntu 16.04 together with the ROS distribution *Kinetic*. All ROS nodes are implemented in either Python or C++.

Name	Type	Message type	Description
/cmd_vel	Topic	geometry_msgs/Twist	move HOBBIT
/head/move	Topic	std_msgs/String	move HOBBIT's head
/head/emo	Topic	std_msgs/String	control HOBBIT's eyes
/MMUI	Service	hobbit_msgs/Request	control UI interface
hobbit_arm	Action	hobbit_msgs/ArmServer	control HOBBIT's arm
move_base_simple	Action	geometry_msgs/PoseStamped	navigate HOBBIT

Table 3.1: Common commands used by HOBBIT

In order to provide a fast and simple way to implement new behaviours several commands should be pre-implemented. These commands are performed either by publishing messages to topics or services, or executing callbacks defined in the corresponding action's client. The common commands and their description are listed in Table 3.1. A detailed list of possible messages for each command is presented in Appendix E.

3.1.2 Purpose of the tool

The primary purpose of Rockly is to simplify the development of demonstrations for the HOBBIT robot platform. It became very popular since the above mentioned EU project and demos of it's behaviours have been presented at a large number of fairs. Currently, there are several application use cases implemented on the robot, including the following[24]:

- Call HOBBIT: summon the robot to a position linked to battery-less call buttons.
- Emergency: call relatives or an ambulance service.
- Safety check: guide the user through a list of common risk sources and provide information on how to reduce them.
- Pick up objects: objects lying on the floor are picked up by the robot.
- Learn and bring objects: visual learning of user's objects to enable the robot to search and fnd them
- Reminders: deliver reminders for drinking water and appointments directly to the user

- Transport objects: placing objects on to the robot and letting it transport them to a commanded location
- Go recharging: autonomously, or by a user command, move to the charging station for recharging
- Break: put the robot on break when the user leaves the fat or when the user takes a nap
- Fitness: guided exercises that increase the overall ftness of the user

All of the demos can be started via the user interface running on HOBBIT's tablet, but re-writing new demos would assume a detailed knowledge of the robot's setup. In order to implement new behaviours and demos more easily it is necessary to provide a programming interface, which provides a powerful, generic base to cover a wide range of HOBBIT's features as well as an intuitive handling.

Furthermore the Automation and Control Institute of the TU Wien was part of the Educational Robotics for STEM (ER4STEM) project, which aimed to turn curious young children into young adults passionate about science and technology with hands-on workshops on robotics. The ER4STEM framework coherently offered students aged 7 to 18 as well as their educators different perspectives and approaches to find their interests and strengths in robotics to pursue STEM careers through robotics and semi-autonomous smart devices. [25] Providing an intuitive programming tool would allow the integration of HOBBIT into such projects, which would be an extra input evaluation parameter.

Finally, the framework should be implemented to be re-used for other ROS based robots. This means that it should not only provide an interface to the mentioned commands for HOBBIT, but an open, adpatable framework. It should be able to allow a flexible configuration and assembly of the provided functions.

3.2 Options

There are several approaches to fulfill the mentioned requirements. In the following subsections three different options are presented by a simple example: the implementation of picking up an object from the floor and putting it on the table. This should give a rough overview in terms of complexity of the usability

as well as the implementation of the corresponding approach. For reasons of simplicity tasks like searching and detecting the object or gripper positioning are excluded.

3.2.1 Custom API

The most obvious way to fulfill the requirements is to provide an API for the desired programming languages (Python, C++). An API is a set of commands, functions, protocols and objects that programmers can use to create software or interact with an external system. It provides developers with standard commands for performing common operations so they do not have to write the code from scratch. In the present case such an API could consist of the following components:

- Initialization: setting up communication and intial states - e.g. creating ROS nodes, starting the arm referencing or undocking from charger
- Topic management: managing the messages published to ROS topics and creating subscriber nodes if applicable
- Service management: managing the ROS services of HOBBIT - e.g. the tablet user interface
- Action management: creating ROS action clients for e.g. navigation or arm movement
- Common commands: providing common commands (see Table 3.1)

It should be noted that the components do not re-implement ROS functionality, but extend it and prvovide a simpler use of it. Depending on how generically the API is implemented it is possible that the user can control the robot without any detailed knowledge of the technical setup. Nevertheless, this approach would assume the user to have knowledge of the programming language the API is desigend for. Refering to the required commands in Table 3.1 an API for Python could be designed as shown in Figure 3.1. The highest usability would be reached, if all input parameters are from common variable types such as interger and string. Indeed, this would increase the implementation effort, especially in terms of error handling, as well as the extent of the documentation, which are huge disadvantages of writing an API.

Assuming the API would be implemented as explained before and the Python module would be named *HobbitRosModule*, Listing 3.1 shows a sample code for

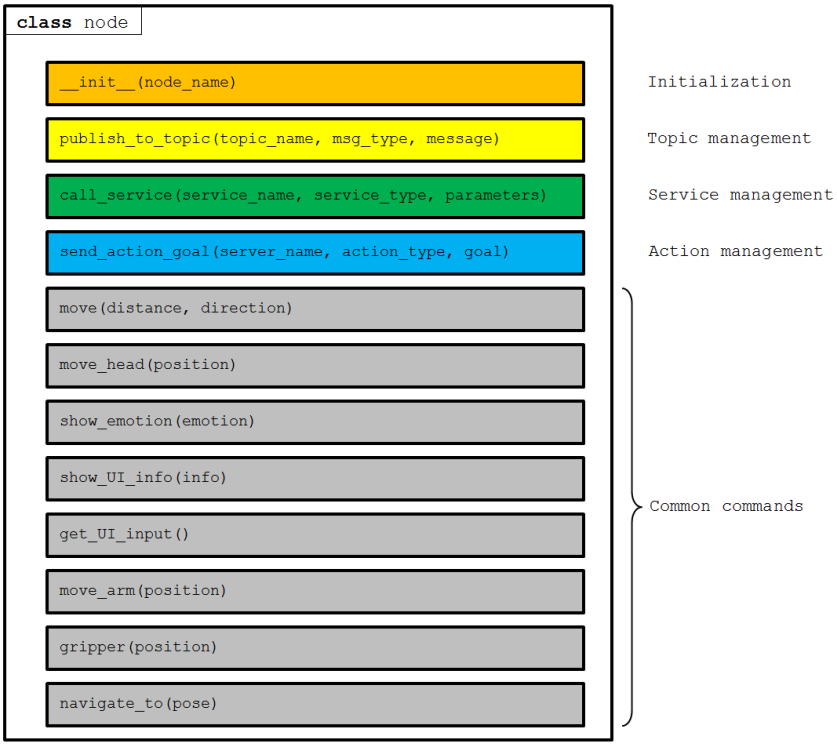


Figure 3.1: Exemplary design of an API for Python

```
1 import HobbitRosModule  # Import of API module
2
3 node = HobbitRosModule.node('demo') # Create ROS node
4 node.gripper('open')    # Open gripper
5 node.move_arm('floor')  # Move arm to floor pick up
     position
6 node.gripper('close')   # Close gripper = pick object
7 node.move_arm('table')  # Move to table position
8 node.gripper('open')    # Open gripper = drop object
```

Listing 3.1: Example Python code using the API shown in Figure 3.1

the mentioned use case to pick up an object. Note that the code is very short and easy to read, which - on the other hand - means that the implementation of the API must cover a broad technical range, such as error handling for unsupported inputs and communication errors.

3.2.2 SMACH

SMACH is a task-level architecture for rapidly creating complex robot behaviour. At its core, SMACH is a ROS-independent Python library to build hierarchical state machines. [26]. Therefore, this approach would also end up in providing an API for the user, but allows to create more complex demos with less effort than the one described in Section 3.2.1. SMACH also provides a powerful graphical viewer to visualize and introspect state machines as well as an integration with ROS, of course. Since the aforementioned example is a very simple one and does not require a lot of the provided SMACH functionality, this section only covers the needed ones to fulfill the requirements. For a detailed description on how to use SMACH refer to [26].

The arm of HOBBIT is controlled via the hobbit_arm action. SMACH supports calling ROS action interfaces with it's so called *SimpleActionState*, a state class that acts as a proxy to an *actionlib* (see Section 3.4.1) action. The instantiation of the state takes a topic name, action type, and some policy for generating a goal. When a state finishes, it returns a so called *outcome* - a string that describes how the state finishes. The transition to the next state will be specified based on the outcome of the previous state. Listing 3.2 shows a possible implementation of picking up a object an placing it on the table. After the imports of the necessary modules (lines 1 to 4), the state machine is instanced (line 7), to which the required states are added (lines 17-22). The parameters passed to *SimpleActionState* are

- the name of the action as it will be broadcasted over ROS (e.g. *hobbit_arm*)
- the type of action to which the client will connect (e.g. *ArmServerAction*) and
- the goal message.

For reasons of readability the goals are declared at a seperate code block (lines 11 to 14). The equivalent visualization of the state machine is shown in Figure 3.2.

Providing just the SMACH interface has some disadvantages and would not be practicable. First, the user would need an advanced knowledge of Python. Depending on the design of the self-implemented API (Section 3.2.1) the knowledge has to be at least at the same level. Next the user would have to understand the API and needs to find a design to fit for the corresponding

```
1  from smach import StateMachine
2  from smach_ros import SimpleActionState
3  from hobbit_msgs.msg import ArmServerGoal,
       ArmServerAction
4  from rospy import loginfo
5
6  # Instance of SMACH state machine
7  sm = StateMachine(['finished','aborted','preempted'])
8
9  with sm:
10     # Definition of action goals
11     goal_floor = ArmServerGoal(data='
           MoveToPreGraspFloor', velocity=0.0, joints=[])
12     goal_table = ArmServerGoal(data='
           MoveToPreGraspTable', velocity=0.0, joints=[])
13     goal_OpGrip = ArmServerGoal(data='OpenGripper',
           velocity=0.0, joints=[])
14     goal_ClGrip = ArmServerGoal(data='CloseGripper',
           velocity=0.0, joints=[])
15
16     # Assambly of the full state machine
17     StateMachine.add('INITIAL_POS', SimpleActionState('
           hobbit_arm',ArmServerAction,goal=goal_OpGrip),
           transitions={'succeeded':'FLOOR_POS','aborted':'
           LOG_ABORT', 'preempted':'LOG_ABORT'})
18     StateMachine.add('FLOOR_POS', SimpleActionState('
           hobbit_arm',ArmServerAction,goal=goal_floor),
           transitions={'succeeded':'CLOSE_GRIPPER','
           aborted':'LOG_ABORT', 'preempted':'LOG_ABORT'})
19     StateMachine.add('GRIPPER_CLOSED',
           SimpleActionState('hobbit_arm',ArmServerAction,
           goal=goal_ClGrip),transitions={'succeeded':'
           TABLE_POS','aborted':'LOG_ABORT', 'preempted':'
           LOG_ABORT'})
20     StateMachine.add('TABLE_POS', SimpleActionState('
           hobbit_arm',ArmServerAction,goal=goal_table),
           transitions={'succeeded':'OPEN_GRIPPER','aborted
           ':'LOG_ABORT', 'preempted':'LOG_ABORT'})
21     StateMachine.add('GRIPPER_OPEN', SimpleActionState(
           'hobbit_arm',ArmServerAction,goal=goal_OpGrip),
           transitions={'succeeded':'finished','aborted':'
           LOG_ABORT', 'preempted':'LOG_ABORT'})
22     StateMachine.add('LOG_ABORT', loginfo('Demo aborted
           !'), transitions={'succeeded': 'aborted'})
```

Listing 3.2: Using SMACH to generate a state machine

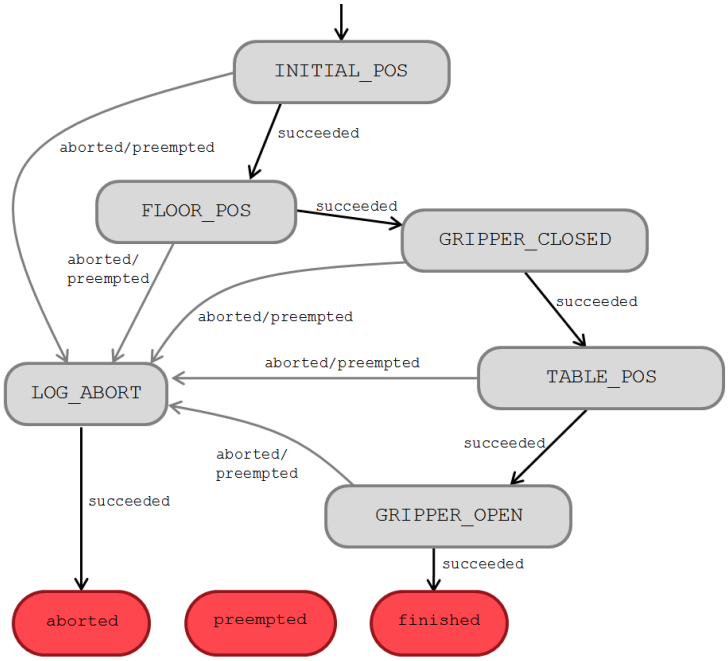


Figure 3.2: State machine generated via Listing 3.2

demo case. Furthermore, it requires the user also to exactly know the ROS specification of the robot. So, if SMACH would be choosen as the underlying framework, it would also be necessary to provide a more abstract API - basically with the same interfaces as shown in Figure 3.1.

3.2.3 Blockly

Blockly is a library that adds a visual code editor to web and Android apps. The Blockly editor uses interlocking, graphical blocks to represent code concepts like variables logical expressions, loops, and more. It allows users to apply programming principles without having to worry about syntax or the intimidation of a blinking cursor on the command line. [27] So for the present case, in contrast to the other approaches the user would not need to have any technical knowledge of HOBBIT, its components and interfaces. Furthermore, such an editor would not require the user to master any programming language. On the other hand implementing this approach would require additional knowledge of web applications (i.e. JavaScript, HTML and CSS).

Figure 3.3 shows an exemplary injection and use as well as the basic structure of Blockly applications. It consists of a toolbox, from where the progamming blocks can be dragged to the workspace where they are connected. The Blockly

Figure 3.3: A short Blockly demo showing it's structure and use

API [28] provides a function to generate a code for all blocks in the workspace to several languages: JavaScript, Python, PHP, Lua, Dart and XML. In the shown example for each of them a tab is avaible to show the generated code. The blocks dragged to the workspace in Figure 3.3 are already customized blocks, with whom an object can be picked up and be placed on the table. There are basically four steps requierd in order to create and use a custom block, which are briefly described in the following paragraphs. A detailed documentation is given in [29].

Defining the block

Blocks are defined in the `/blocks/` directory of the source code by adding either JSON objects or JavaScript functions to the `Blockly.Blocks` mapping. It includes the specification of the shape, fields, tooltip and connection points. An example definition using a JavaScript function of the *gripper* block is shown in Listing 3.3. Attention should be payed to lines 6 to 21, where the input fields are defined (line 8). Here a dropdown field with two options ("Open" and "Close") is created. The name ("gripper_position") is used to refer to it later.

Providing the code

Similar to the definition of a block, the code, which is generated out of them, is stored in a mapping variable inside the Blockly library. Since different languages are supported, the code definition has to be in the right directory. Note that it is not necessary to provide code for each language. The code generation is handled in the `/generator/` directory of the library. Each language has it's own helper functions file (e.g. `python.js`) and subdirectory, where the code for each

```
Blockly.Blocks['hobbit_arm_gripper'] = {
    init: function () {
        this.jsonInit({
            "type": "hobbit_arm_gripper",
            "message0": "%1 Gripper",
            "args0": [
                {
                    "type": "field_dropdown",
                    "name": "gripper_position",
                    "options": [
                        [
                            "Open",
                            "open"
                        ],
                        [
                            "Close",
                            "close"
                        ]
                    ]
                }
            ],
            "previousStatement": null,
            "nextStatement": null,
            "colour": 360,
            "tooltip": "Control HOBBIT's gripper",
            "helpUrl": ""
        });
    }
};
```

Listing 3.3: Block initialization using a JavaScript function

block is defined. There are several interfaces functions provided by Blockly to manage interaction with a block - such as collecting arguments of the block. A short example to control HOBBIT's gripper is shown Listing 3.4. In line 2 the `block.getFieldValue()` function is used to get the user's selection of the dropdown field. Note that the generator always returns a string variable including the code in the desired language (line 4). So the shown example requires to use a custom Python API (such as Figure 3.1), because `node.gripper()` is not a built-in function of Python.

```
1 Blockly.Python['hobbit_arm_gripper'] = function(block)
    {
2   var dropdown_movement = block.getFieldValue('
      gripper_position');
3
4   return 'node.gripper(\''+dropdown_movement+'\')\n';
5 };
```

Listing 3.4: Defintion of a code generator in Blockly for Python

Building

After the customized block and it's code generator are defined, the whole Blockly project has to be rebuilt by running `python build.py`. Building means that the source code, which is usually spread to several files (in the given case over a hundred), is converted into a stand-alone form, which can be easily integrated. The Blockly build process uses Google's online Closure Compile and outputs compressed JavaScript files for core functionalites, blocks, block generators for each progamming language and a folder including JavaScript files for messages in several lingual languages. In our case the following four files needs to be included:

- `blockly_compressed.js`: Blockly core functionalites
- `blocks_compressed.js`: Defintion of all blocks
- `python_compressed.js`: Code generators for all blocks
- `/msg/js/en.js`: English messages for e.g. tooltips

Add it to the toolbox

Once the building is completed and the necessary files are included to the web application, the custom block needs to be included in the toolbox. The toolbox is specified in XML and passed to Blockly when it is injected. Assuming the blocks shown in Figure 3.3 are build as described in the previous paragraphs, they can be add to the toolbox as shwon in Listing 3.5.

```
1 <xml id="toolbox" style="display: none">
2   <block type="hobbit_arm_gripper"></block>
3   <block type="hobbit_arm_movement"></block>
4 </xml>
```

Listing 3.5: Minimal example of adding two blocks to a Blockly toolbox

3.2.4 Decision

In order to select the best fitting solution the requirements explained in Section 3.1 are summarized as follows:

- The user wants to build as complex demos as possible.
- The user wants an intuitive interface to create demos.
- The user should need as little knowledge of the robot as possible.
- The user should need as little technical knowledge as possible.
- The implementation of the tool should not exceed the usual effort of a master thesis.
- The tool should be maintable, meaning it should have clear interfaces and as less dependencies as possible.
- The tool should provide the ability to add new functionalites.

The selection is based on subjective rating each of the approaches in regards of each mentioned requirement with a score ranging from 1 (lowest) to 5 (highest). The approach with the highest overall score is considered as the best fitting one and will be implemented. The scales used to rate each criteria are listed in Table 3.2 and an explanation of the scores is prestend in the following.

Criteria	1	2	3	4	5
Complexity of demos	v. low	low	neutral	high	v. high
User Interface	not int.	little int.	neutral	int.	v. int.
Robot specific know-how	v. high	high	moderate	low	v. low
Technical know-how	v. high	high	moderate	low	v. low
Implementation effort	v. high	high	moderate	low	v. low
Maintainability	v. low	low	moderate	high	v. high
Scalability	v. low	low	neutral	high	v. high

Table 3.2: Scales used for evaluation scores. Abbreviations: very (v); intuitive (int)

Because of its high abstraction and encapsulation of the libary complex demos are hardly possible to implement using a block-based VPL like Blockly. It would require lots of blocks, which would make the workspace very confusing. An additional factor hard to implement would be the usage of describer nodes and callbacks, because Blockly is designed for procedural programming. Using a custom API would allow more or less the same level of demo complexity as using the standard ROS API, since there is little abstraction and additon. However, the powerful state machine functionality of SMACH enables users to create, debug and visualize programs and therefore apply it to complex applications.

The graphical UI of Blockly can be considered as very intuitive, because it has become the dominanting VPL framework used in a varity of applications (see Section 2.2). On the oter hand, the usability of an API mostly depends on its documentation and design since it does not provide a graphical interface. Therefore, both programming interfaces are considered as little intuitive.

When evaluating the required robot specific and technical know-how Blockly benefits from its high abstraction level. Once the blocks are set up, the user does not need to take care about which message and communication pattern should be used, and therefore needs very low technical and robot-specific expertise. Using APIs particularly requires advanced programming knowledge in order to use it correctly. As prestend by the example Section 3.2.2 additional robot specific know-how is required when using SMACH in order to add states to the state machine.

A big challenge when designing and implementing an open, customized API is the error handling in case of invalid usage in order to prevent programs from crashing. Therefore, the implementation effort for this apporach is considered to be very high. In the case of choosing SMACH the implementation effort decreases, because it already provides such features for its capabilites.

Implementing the Blockly approach also would require to create an API for scalability reasons, but the mentioned disadvantage must not be taken in account, because it would not be open for end users. However, the fact, that Blockly is a JavaScript libary would increase the implementation, because additional programming expertise is requierd.

The necessity to master a second programming language besides Python is also the reason that the Blockly approach is considered to provide a very low maintainability - especially, when taking into account, that the Blockly workflow itself (Section 3.2.3) requires good web development skills. APIs, including SMACH, does not need a lot of maintenance once fully set up. Assuming to be familiar with the corresponding programming language, adding new functionality can be done with moderate effort.

Evaluating possible scalability limits lead to similar conclusions as when discussing the complexity of demos. The VPL of Blockly does not provide a user friendly way to handle large programs, especially when thinking of running multiple ROS nodes. SMACH provides a powerful state machine functionality, but nothing beyond it, and overheads for large application use cases are possible. Since using a simple API provides more flexibility, it can be considered that a custom API would provide the highest scalability of the investigated approaches.

Table 3.3 shows the final result of the evaluation. Based on the assessment Blockly convinced with its very intuitive interface, which requires the user to have very little previous knowledge to build demos. Despite the fact that maintainability and scalability suffer from the high abstraction and encapsulation of the libary, the effort in terms of implementation is not worth mentioning compared to the other approaches. This results from the fact that each one would require the design of a custom API. Instead of just abstracting the given ROS functionality, Blockly adds a much more user friendly option to create demos - in contrast to the others.

3.3 Design

Based on the mentioned framework decision a design for the tool was developed. Figure 3.4 gives an overview of Rockly's final architecture and it's components. The front end consists of the following five components:

- a **Demo Management** which allows the user to create, edit, delete and run demos easily,
- a **Code Editor** to allow further editing of the generate code before running it on the robot,

Criteria	Custom API	SMACH	Blockly
Complexity of demos	3	5	2
User Interface	2	2	5
Robot specific know-how	3	1	5
Technical know-how	1	1	5
Implementation effort	1	2	2
Maintainability	3	3	1
Scalability	4	3	2
Σ	17	17	22

Table 3.3: Evaluation of possible approaches. Scores range from 1 (lowest) to 5 (highest)

- the **Code Generation** with the help of Blockly - the heart of the tool,
- an interface for creating and managing custom blocks (**Block Configuration**, which then can be used for code generation,
- sending and receiving data from the robot via the **Back end Communication**

and the back end is made up of the following four:

- sending and receiving data from the user interface via the **Front end Communication**,
- a **Storage Management**, which provides the demos and custom blocks to the user,
- a **Python Module** containing all necessary functionalites to provide an ROS executable code,
- an interface to finally execute the generated code on the robot (**Code Execution**).

3.4 Supporting frameworks & dependencies

As described the components can be clustered into front end and back end, but they can be seen also in respect of their functional interfaces. On the one hand there is a connection to the robots sensors and actuators, on the other

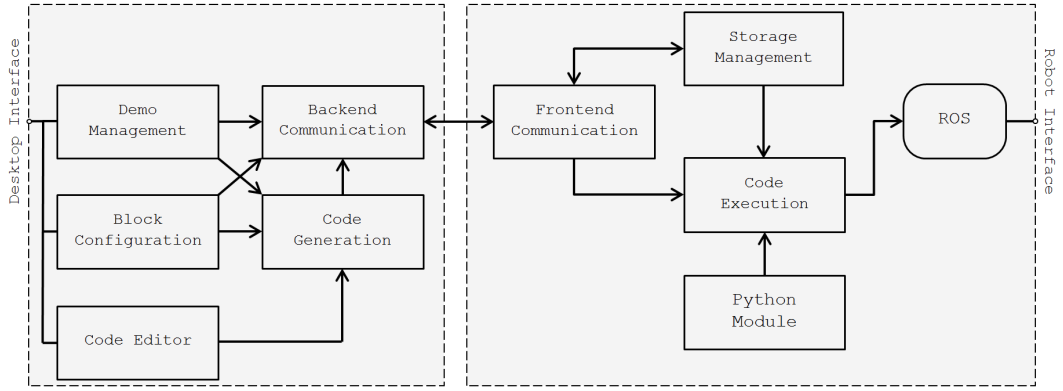


Figure 3.4: Front end and back end architecture

hand there is an interface to the human using the tool. Both functionalites are build up with the support of software frameworks, which are described in the following sections.

3.4.1 ROS

ROS provides libraries and tools to help software developers create robot applications. It provides hardware abstraction, device drivers, libraries, visualizers, message-passing, package management, and more. Since ROS is licensed under an open source, BSD license it used for a wide range of robots, sensors and motors. Covering all of its features would go beyond the scope of this thesis, so just the concepts, which are needed to implement Rockly, are summarized.[30]

Packages

Software in ROS is organized in packages. A package might contain ROS nodes, a ROS-independent library, a dataset, configuration files, a third-party piece of software, or anything else that logically constitutes a useful module. The goal of these packages is to provide this useful functionality in an easy-to-consume manner so that software can be easily reused.

Nodes

A node is a process that performs computation and are written with the use of a ROS client library, such as roscpp (for C++) or rospy (for Python). Nodes are combined together into a graph and communicate with one another using streaming topics, RPC (Remote Procedure Call) services, and the Parameter Server. A robot control system will usually comprise many nodes. For example,

one node controls a laser range-finder, one node controls the robot's wheel motors, one node performs localization, one node performs path planning, and so on.

Topics

Topics are named buses over which nodes exchange messages. Topics have anonymous publish/subscribe semantics, which decouples the production of information from its consumption. In general, nodes are not aware of who they are communicating with. Instead, nodes that are interested in data subscribe to the relevant topic; nodes that generate data publish to the relevant topic. There can be multiple publishers and subscribers to a topic. Each topic is strongly typed by the ROS message type used to publish to it and nodes can only receive messages with a matching type.

Messages

A message is a simple data structure, comprising typed fields. Standard primitive types (integer, floating point, boolean, etc.) are supported, as are arrays of primitive types. Messages can include arbitrarily nested structures and arrays. Nodes can also exchange a request and response message as part of a ROS service call. Message descriptions are stored in .msg files in the `/msg/` subdirectory of a ROS package.

Services

The publish-subscribe model is a very flexible communication paradigm, but its many-to-many one-way transport is not appropriate for RPC request-reply interactions, which are often required in a distributed system. Request-reply in ROS is done via a service, which is defined by a pair of messages: one for the request and one for the reply. Services are defined using .srv files, which are compiled into source code by a ROS client library.

Actions

In some cases, e.g. if a service takes a long time to execute, the user might want the ability to cancel the request during execution or get periodic feedback about how the request is progressing. The actionlib package provides tools to create servers that execute long-running goals that can be preempted. It also provides a client interface in order to send requests to the server. The action client and action server communicate via the ROS Actionlib Protocol[1], which

[1] http://wiki.ros.org/actionlib

is built on top of ROS messages. The client and server then provide a simple API for users to request goals (on the client side) or to execute goals (on the server side) via function calls and callbacks.

3.4.2 JavaScript frameworks

Since Rockly is a web application all of the functionality blocks do touch JavaScript in some way. And since there are a lot of JavaScript frameworks supporting the development of web applications, it is necessary to get an overview of the important ones - besides Blockly (see Section 3.2.3) - used to implement Rockly before diving into the implementation itself.

Node.js

Node.js is an open source platform that allows you to build fast and scalable network applications using JavaScript. Node.js is built on top of V8, a modern JavaScript virtual machine that powers Google's Chrome web browser. At its core, one of the most powerful features of Node.js is that it is event-driven. This means that almost all the code written in Node.js is going to be written in a way that is either responding to an event or is itself firing an event (which in turn will fire other code listening for that event). In an effort to make the code as modular and reusable as possible, Node.JS uses a module system - called "Node Package Manager" (NPM) - that allows to better organize the code and makes it easy to use third-party open source modules.

Express

Express is a minimal and flexible Node.js web application framework, providing a robust set of features for building single, multi-page, and hybrid web applications. In other words, it provides all the tools and basic building blocks one need to get a web server up and running by writing very little code. Listing 3.6 shows a minimal example of using Node.js and Express to implement the famous "Hello World" program. First the Express module is imported (Line 1) and an instance of the app is created (Line 2), which finally listens to the configured port (Line 6). Line 3 to 5 describes the mentioned event-driven behaviour: `app.get('/',...)` causes every request to trigger the given callback function, which in this case is just sending the "Hello World" string.

Ace

Ace, whose name is derived from "Ajax.org Cloud9 Editor", is a standalone code editor written in JavaScript. Ace is developed as the primary editor for Cloud9

```
1 var express = require('express');
2 var app = express();
3 app.get('/', function(req, res){
4    res.send('Hello World');
5 });
6 app.listen(3000);
```

Listing 3.6: "Hello World" program implemented using Node.js and Express

ide) - an online integrated development environment - and the successor of the Mozilla Skywriter (formerly Bespin) Project. It supports syntax highlighting and auto formatting of the code as well as customizable keyboard shortcuts.

jQuery

jQuery is a fast, small, and feature-rich JavaScript library. It makes things like HTML document traversal and manipulation, event handling, animation and ajax (Asynchronous JavaScript and XML) much simpler with an easy-to-use API that works across a multitude of browsers. Rockly basically uses its functionality to exchange data via RESTful (Representational State Transfer) web services and manipulating HTML elements.

4 Implementation

This chapter gives detailed insights into the implementation of Rockly. Each component presented in the architectural overview (Section 3.3) is explained together with the essential code snippets.

4.1 Back end & Front end Communication

The communication between front end (client) and back end (server) is done via a RESTful API. A RESTful API is based on representational state transfer (REST) technology and uses HTTP (Hypertext Transfer Protocol) request methods, which are defined in RFC 2616 [31], to exchange data between web services. All implemented API endpoints, which are used to manage demos and codes are listed in Table 4.1.

Type	Path	Description
GET	/demo/list	list all demos saved on the robot
GET	/demo/load/*{demoId}*	get the XML tree of a demo
POST	/demo/save	save a new demo on the robot
POST	/demo/run/*{demoId}*	run a demo
DELETE	/demo/delete*{demoId}*	delete a demo
GET	/demo/toolbox	load the toolbox of the Blockly interface
GET	/block/list	list all already configured custom blocks
POST	/block/create	create a new custom block
PUT	/block/update/*{blockId}*	update a custom block
DELETE	/block/delete/*{blockId}*	delete a custom block

Table 4.1: API endpoints for managing demos and custom blocks

4.2 Demo Management

The Demo Management (Figure 4.1) is one key feature of Rockly and the entry point of it. It gives an overview of all avaible demos, meaning demos, which

are saved on the robot in a specific directory. It provides a graphical interface and hence makes it a lot easier to manage and run demos - in contrast to the other mentioned options (Section 3.2), where e.g. the user needs to explicity locate the file and run it. Since usability and simplicity are main requirements for this tool, such a feature is a main advantage of it. Furthermore the main page provides the option to create new demos and a redirection to the block configuration component (Section 4.3).

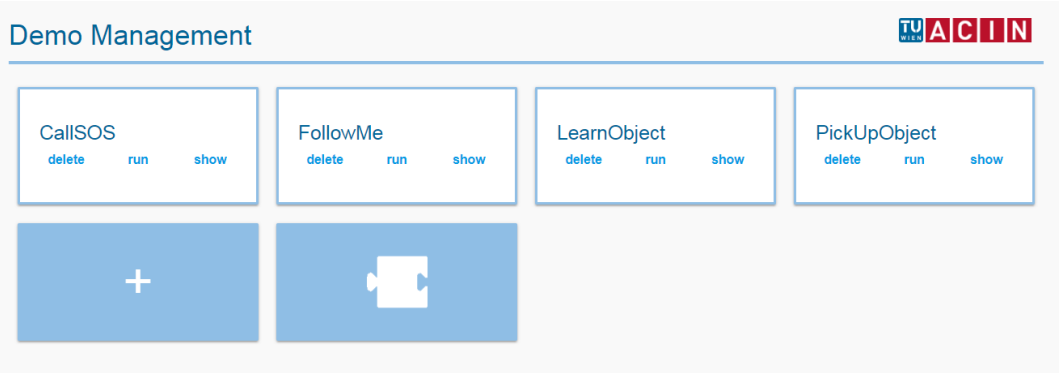


Figure 4.1: Demo Management Page

At the start of the tool the `/demos/` directory inside the tools folder on the robot is searched for saved demos. Since we talking about a web-based tool and demos are saved on the robot, this requieres a call to the back end, which then returns a list of all demos via the RESTful API. The structure of all demos are stored in `.xml` files, which are interpreted by Blockly.

4.3 Block Configuration

There are a lot of reasons a user might want to create a custom block. The pre-implemented blocks are designed for the ROS architecture of HOBBIT and any other robot won't show the desired behavior when receiving commands sent from these blocks. Even for HOBBIT itself the provided block set does not cover all of its functionalites. Using another robot means publishing different data to other topics, calling other services and using other action clients. Of course, the user can walk through the whole custom block creation process described in Section 3.2.3 to create new blocks. This requieres knowledge of progamming in JavaScript and the Blockly API. The user would then be able to use all of Blockly's broad range of features and advantages, especially dynamically changing of block features.

On the other hand, for a lot of the tasks such features are not necessary or a workaround with less effort can be found, respectively. For this reason Rockly provides a block configuration interface, which allows the user to create custom blocks with the two basic visual designs: an execution block and an input block (Figure 4.2). With this interface the user can configure blocks for publishing data to topics, calling services with parameters and creating simple action clients.

Figure 4.2: Basic visual designs: execution block (left) and input block (right)

The design of the interface is shown in Figure 4.3. It is devided into three sections:

I. An overview of already created custom blocks

II. A form to provide general information of the block

III. A form to provide detail information of the block (type-specific)

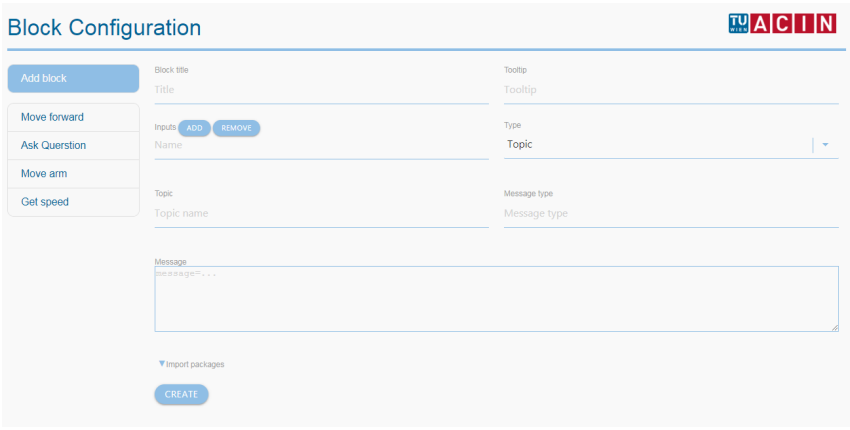


Figure 4.3: Design of the block configuration interface

The list of all already configured custom blocks is loaded via the GET request endpoint `/block/list` of the tool's internal RESTful API. The request delivers an object with the Blockly-conform block and code definitions, the unique ID and name of the respective block as well as the block's meta data. The latter is used to set the general and detailed forms in case the custom block is selected.

Identification which custom block is selected is handeld via the query string of the URL, which is set to the custom block's ID if it is selected. The ID of a block is an alphanumerical 12-character string, which is generated randomly.

The general information form is primarily used to set the visual design of the block. The user can configure the title and tooltip of the block as well as the number and names of the inputs, which then can be used in the detail section. The detail section varies depending on which type of the ROS communicating patterns is used. A topic must be specified by its name, the message type and the message itself. If the user wants to create a custom block for calling services, it is necessary to provide the name of the service, the message type and a list of all message type specific fields with their values. Furthermore, it is possible to choose whether the response of the service should be used as output - which will lead to an input block - or not - then an execution block is generated (Figure 4.2). The detail section for actions includes fields for setting the execution timeout (which is defined as the time to wait for the goal to complete), the server name, the message type and the goal. It is also possible to provide the following callback functions:

- done_cb: callback that gets called on transitions to *Done* state.
- active_cb: callback that gets called on transitions to *Active* state.
- feedback_cb: callback that gets called whenever feedback for the goal is received.

All detail sections also feature an input field for importing all Python packages that are needed to execute the code correctly, e.g. messages types that are used to generate the message. Some step-by-step examples for configuring custom blocks are presented in Appendix B.

4.4 Code Generation

Demos are created via the user interface shown in Figure 4.4. It can be devided into three sections:

I. the **navigation header** holds elements to directly manage the current demo,

II. the **toolbox** contains code blocks organized in categories,

III. the **workspace** where the blocks can be dragged to from the toolbox and be connected with each other.

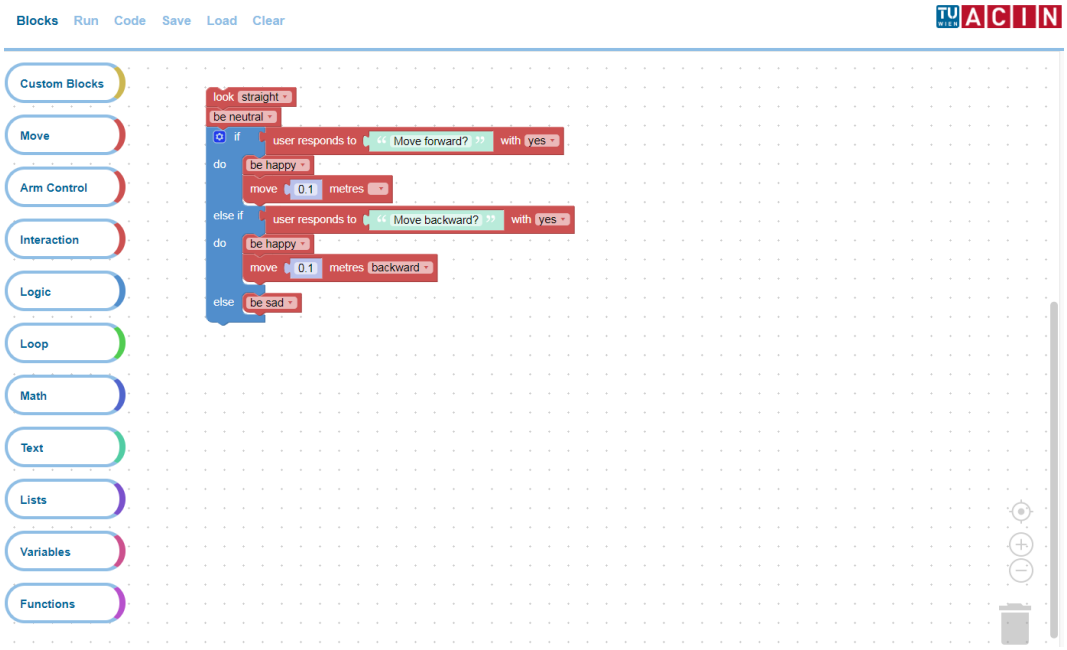


Figure 4.4: User interface for demo and code generation

Blocks which are dragged to and combined at the workspace are used to generate an executable code. This is done with the help of Blockly (see Section 3.2.3). Blockly came up with some predefined code blocks that allow some basic programming procedures. On top of them Rockly provides further blocks, which allows to connect to HOBBIT and perform several actions (Table 3.1). A list and description of these blocks can be found in Appendix C. All of them are using one of the mentioned ROS communicating pattern (topic, service, action) to call the interface of the robot and have a similar structure thanks to the design of a custom Python module (Section 4.6).

To get a clearer understanding of the general structure of these blocks and the Blockly custom block creation (Section 3.2.3) an explicit example is presented. The choosen example shows the block and code definition to create a block, which allows to move HOBBIT for a given distance into a given direction (Figure 4.5).

Figure 4.5: Example block to be created

The full block definition for the mentioned example is given in Listing 4.1.

```
Blockly.Blocks['hobbit_move'] = {
  init: function () {
    this.jsonInit({
      "type": "hobbit_move",
      "message0": "move %1 metres %2",
      "args0": [
        {
          "type": "input_value",
          "name": "distance",
          "check": "Number"
        },
        {
          "type": "field_dropdown",
          "name": "direction",
          "options": [
            [
              "forward",
              "+"
            ],
            [
              "backward",
              "-"
            ]
          ]
        }
      ],
      "previousStatement": null,
      "nextStatement": null,
      "colour": Blockly.Constants.hobbit.HUE,
      "tooltip": "Move HOBBIT",
      "helpUrl": ""
    });
  }
};
```

Listing 4.1: Full block initialization for moving HOBBIT forward and backward

Lines 2 and 3 indicates, that the block is initialized using a JavaScript function (same as in Listing 3.3). In lines 1 and 4 the name and type of the block are set, which is important to get the corresponding code object later on. The `message0` key is used to set the message displayed on the block, whereas the placeholders (%1,%2) are replaced by the arguments given by the objects passed to the `args0` key. In the presented case the first argument is an input field and must be a number (lines 8,10) and the second one (lines 13 to 24) a dropdown field with options "forward" and "backward" displayed. In order to get the passed values for code generation a name for each paramter is set (lines 9,14). Lines 27 and 28 indicates that the block has connections on the top and bottom, but there are no constraints for them. The last three lines are just used to set the color, tooltip and an optional help URL.

```
1 Blockly.Python['hobbit_move'] = function(block) {
2   var value_distance = Blockly.Python.valueToCode(block
      , 'distance', Blockly.Python.ORDER_ATOMIC);
3   var dropdown_direction = block.getFieldValue('
      direction');
4   Blockly.Python.InitROS();
5
6   var code = Blockly.Python.NodeName+'.move('+
      dropdown_direction+value_distance+')\n';
7   return code;
8 };
```

Listing 4.2: Full code initialization for moving HOBBIT forward and backward

The corresponding code initialization is shown in Listing 4.2. Line 1 again shows the internal design of Blockly, which is based on assigning functions and objects to classes. Blockly supports code generation for several programming languages, which all are implemented in seperate classes. Since the presented tool should convert blocks into Python code, the Python class is used. Within the code initialization the input values of the block are read through the provided API first (line 2,3). Then some ROS specific initialization is done (line 4), which basically handles the import of the rospy package and the custom Python module as well as creating a ROS node. The name of the node is defined by the constant string `Blockly.Python.NodeName` and an instance of a certain class of the Python module. `Blockly.Python.InitROS()` is a custom command and must be included in the code initialization of each block. Finally, the code string is assembled by just calling the corresponding control function of the ROS node instance with the input parameters (line 6) and then returned

(line 7).

The code initialization of each created block follows the mentioned steps, which can be summarized as follows:

1. Getting the values of the inputs.
2. Initialization of the ROS interface
3. Assembly of the code string by simply calling the responding functions of the Python module

This generic design is another decrease of the conditions - in terms of JavaScript knowledge - for using the Blockly framework, because it outsources the main and most complex task of assembling the executing code to an interface more robot programmers are familiar with. Futhermore Google provides a visual interface - using the Blockly framework itself - to easily create the block configuration object.[1]

4.5 Code Editor

The generated code can be accessed by clicking on the *Code* tab on the user interface (Figure 4.4). It provides the opportunity to add further functionality to the code, which is not already covered by the tool, before executing the demo or saving it. This could be very basic modification, e.g. just adding comments to the code or inserting debugging messages, or more advanced ones, e.g. adding the functionality to subscribe to a topic.

This makes Rockly not only to be a stand-alone solution for basic use cases, but a supporting tool for more experienced users, who do not want to build their code from scratch.

The implementation of the provided editor basically involve just embedding Ace via its API (see Section 3.4.2). The basic embedding code is shown in Listing 4.3. It also shows how connected blocks on the workspace are translated into an executable Python code using the Blockly API (Line 2) and to display the generated code (Line 10). The variable *workspace* is an instance of the static class *Blockly.Workspace*[2].

4.6 Python Module

As described above each block of the HOBBIT block set calls a corresponding method of the ROS node class inside the custom Python module, which then

[1] https://blockly-demo.appspot.com/static/demos/blockfactory/index.html
[2] https://developers.google.com/blockly/reference/js/Blockly

```
1  // Generate code from workspace
2  var code = Blockly.Python.workspaceToCode(workspace);
3  // Create Ace instance and set preferences
4  var editor = ace.edit("editor");
5  editor.setTheme("ace/theme/chrome");
6  editor.getSession().setMode("ace/mode/python");
7  editor.getSession().setUseWrapMode(true);
8  editor.setShowPrintMargin(false);
9  // Display code
10 editor.setValue(code);
```

Listing 4.3: Embedding Ace, setting preferences and displaying generated code

calls generic communication methods. This section describes how the module is set up and the generic methods to initiate the ROS communication for each communication pattern are implemented. It is desigend as shown in Figure 3.1 with a class containing the necessary properties and methods. For all listings debugging messages are excluded.

Initialization

For any form of communication between the nodes, they have to register to the so called ROS Master - the master node, which provides naming and registration services to the rest of the nodes in the ROS system. It has to be started as the first process, which in the case of HOBBIT is done during the booting process. Therefore, the initialization process of the custom Python module only needs to register a new client node. This is done by simply calling the corresponding `init_node` method[3] of the rospy package, which takes the node's name as parameter. Duplicate calls to `init_node` are forbidden, for which reason the `Blockly.Python.InitROS()` method inside every block was introduced as mentioned in Section 4.4. It ensures that `init_node` is called from the main Python thread.

Publishing to a topic

For publishing to a topic the `rospy.Publisher` class[4] of the rospy package is used. It takes two mandatory initialization paramters: the resource name of topic as a string and the message class. The publishing itself is executed by calling the `publish` method of the class. It can either be called with the message instance

[3] http://docs.ros.org/jade/api/rospy/html/rospy-module.html#init_node
[4] http://docs.ros.org/melodic/api/rospy/html/rospy.topics.Publisher-class.html

to publish or with the constructor arguments for a new message instance. The full implementation of the generic publishing method is shown in Listing 4.4. There are a few important things to be considered. First, an instance of the `rospy.Rate` class[5] is created to ensure the publisher is instanced and the message is published. Second, the `exec` Python built-in function is used, so that any manipulation by the user should be prevented. This is ensured by encapsulating the generic function as mentioned in the beginning of this section. Last, all message classes for the implemented commands (Table 3.1) are imported at the beginning in order to successfully execute the `exec` statement.

```
1 def publishTopic(self, topic, message_type, message):
2     rate = rospy.Rate(1)
3     exec('pub = rospy.Publisher(\''+topic+'\', '+
          message_type+', queue_size=10)')
4     rate.sleep()
5     pub.publish(message)
6     rate.sleep()
```

Listing 4.4: Implementation of a generic method to publish to a ROS topic

Calling a service

To call ROS services it is first necessary to create an instance of the `rospy.ServiceProxy` class[6] with the name and class of the desired service. Then it is recommended to wait until the service is available - which is done by calling the `rospy.wait_for_service` method - before finally calling the instance. This basic steps are included in the generic service call method of the custom Python module (Listing 4.5). It can be structured into three parts: the two just mentioned - creating (lines 9 to 14) and calling the instance (lines 16 to 22) - and a parameter preparation part (lines 2 to 7).

The latter is introduced to create a simple and understandable execution statement when calling the service. Service parameters, which are passed as list to `callService` method, are splitted and assigned to temporary variables `par0,par1,...,parN`, where $N = k - 1$ and k being the number of service parameters. These temporary variables are then combined to a string, which seperates the parameters with a comma. This string then is passed to the execution statement. Additionally the basic error handling is outlined by excepting typical ROS provided exceptions.

[5] http://docs.ros.org/jade/api/rospy/html/rospy.timer.Rate-class.html
[6] http://docs.ros.org/api/rospy/html/rospy.impl.tcpros_service.ServiceProxy-class.html

```
1  def callService(self,ServiceName,ServiceType,args):
2      ParameterList = []
3      if args:
4          for i,arg in enumerate(args):
5              exec('par'+str(i)+'=arg')
6              ParameterList.append('par'+str(i))
7      parameters = ','.join(ParameterList)
8
9      try:
10         rospy.wait_for_service(ServiceName)
11         exec('servicecall = rospy.ServiceProxy(\''+
               ServiceName+'\',' +ServiceType+')')
12
13     except rospy.ROSException:
14         return None
15
16     try:
17         exec('req = '+ServiceType+'Request('+parameters
               +')')
18         resp = servicecall(req)
19         return resp
20
21     except rospy.ServiceException:
22         return None
```

Listing 4.5: Implementation of a generic method to call a ROS service

Sending a goal to an action server

Although the ROS actionlib is a very powerful component, its usage is very simple, as can be seen in Listing 4.6. The action client and server communicate over a set of topics. The action name describes the namespace containing these topics, and the action specification message describes what messages should be passed along these topics. This infos are necessary to construct a `SimpleActionClient` and open connections to an `ActionServer`[7] (line 2). Before sending the goal to the server it is requiered to wait until the connection to the server is established. Afterwards an optional argument (`timeout`) decides how long the client should wait for the result before continuing its code execution and returning the result.

[7] http://docs.ros.org/jade/api/actionlib/html/classactionlib_1_1ActionServer.html

```
1 def sendActionGoal(self, namespace, action_type, goal,
    timeout = rospy.Duration()):
2     exec('client = actionlib.SimpleActionClient(\''+
          namespace+'\', '+action_type+')')
3     client.wait_for_server()
4     client.send_goal(goal)
5     client.wait_for_result(timeout)
6     return client.get_result()
```

Listing 4.6: Implementation of a generic method to use the actionlib

4.7 Storage Management

For reasons of simplicity all necessary data - such as demos, executable codes or custom blocks - are stored in raw files on the robots. They are managed by a own service within the back end. It uses the file system module (*fs*)[8] of Node.js. All of its file system operations have synchronous and asynchronous forms. Both are used in the implementation. The following functionalites are implemented using the storage management service:

- Listing all demos
- Saving, deleting, showing and running a specific demo
- Listing all custom blocks
- Creating, editing and deleting a custom block
- Providing the toolbox of Blockly's workspace

Listing resources - i.e. demos and blocks - following a simple read-and-send algorithm. The management of demos and blocks are slightly different, because informations of blocks are stored in a single file, while informations of demos are spread to multiple files and directories. A more detailed explanation of the latter is given in Section 4.8. The toolbox is assembled in two steps. The structure of the basic toolbox - including the HOBBIT block set and the predefined code blocks of Blockly - is stored in an `.xml` file. This allows the user to reorganize the toolbox to his preferences just following the description mentioned in Section 3.2.3 and without any necessity to touch the code. After reading the basic toolbox information, the storage management services checks whether there are any custom blocks created already, and, if so, appends them to the toolbox.

[8] https://nodejs.org/api/fs.html

4.8 Code Execution

The code generated through the blockly framework (Section 4.4) and optionally edited by the user (Section 4.5) is saved in a `.py` file seperatley from the demo `.xml` file. Although this leads to slightly more effort in terms of maintaining the tool, it also has an important advantage: it provides the ability to reuse the code indepently from the raw block data and the tool itself, e.g. for building more complex demos and ROS nodes, running it from the command line or simply sharing it. In other words, it is not even necessary to run the tool on the robot itself. Of course, there is also the option to run the code directly from Rockly via the integrated *Run* button of the user interface (Figure 4.4). In this case the corresponding request to the back end (`/demo/run/`, see Table 4.1) is send, including the code inside the request body. The back end then executes the code using the *child_process* module[9] of Node.js.

[9] https://nodejs.org/api/child_process.html

5 Evaluation

The following chapter describes how the implemented tool was evaluated. A user study was conducted in order to find out if the developed tool is seen as an improvement compared to the traditional coding approach. The following sections describe the study desgin, the data acquisition, the data analysis and the findings.

5.1 Experiment Setup

The implemented tool was evaluated within the scope of the course "Selected Topics - Robotics and Computer Vision" held by the V4R group of the automation and control insitute at the TU Wien. This section describes the goals and design of the experiment as well as the questions the participants were asked.

5.1.1 Goals of the Experiment

The following goal questions (GQ) should be answerd by the experiment:

- *GQ1*: Is the tool seen as an improvement compared to the traditional coding approach?
- *GQ2*: Do participants make less pauses when they use the tool compared to coding?
- *GQ3*: Does it take less time to find solutions when using the tool?
- *GQ4*: Is the usage of the tool intuitive?

All of the presented goal have in common that they can be answered positively, negatively and none of both. The third option means that furher research must be done, e.g. conducting further studies to get an answer for this question.

The primary goals of this evaluation is to find out if the developed tool is seen as an improvement compared to the traditional coding approach. This question is described by *GQ1* and should be answerd by analysing the feedback of participants, who have experience in programming robotics - especially using ROS.

Also, the answers to workload questions can be taken in account. (Section 5.1.4)

GQ2 should give an answer to the question whether using the tool affects the workflow of the participants. Do they make less pauses when using the tool? Do they change the code more often when using the traditional approach? When do they struggle? To quantify this the measurements presented Section 5.1.2 are analyzed as well as the answers to questions regarding the workload.

GQ3 simply should examine if using the tool saves time compared to the coding approach. Saving time when creating programs means more time for other work, which especially for users of the primary target group (see Section 3.1.2) is important, because it may not is the major content of their work or research.

Finally, *GQ4* should provide insights into the usability of the tool. This can be derived by asking feedback questions to the participants as well as by analysing the workflow during the experiment (see Section 5.1.2).

5.1.2 Study Design and Procedure

A cross over design was used for the experiment splitting the participants into two groups. Therefore, two different use cases were designed, which are described in Section 5.1.3. For the first use case, one group worked with the tool and the other group was asked to implement a code for the same use case. For the second use case roles were switched and participants which had worked with the tool had to write a code and vice versa.

For the coding task the group was supported by material covering explanation of the necessary ROS concepts including examples. Furthermore a list of the necessary ROS specifications (topics, services, actions, messages etc.) was provided. The participants were asked to use a customized code editor, which was able to perform basic measurements. They were free to choose whether implementing in Python or C++. This instrumentation ensures that the results and measurements of both tasks are compareable in terms of exploring the impact of programming knowledge.

All materials, also including the task description of the use cases and the questionnaires used to collect data, were reviewed by a colleague not participating in the experiment in advance to guarantee its understandability and soundness. All the study materials can be found in Appendix E. At the beginning of the experiment the tool was introduced using a short live demo showing all concepts and how the tool and the code editor, respectivley, should be used. The participants worked on each task for a total of 30 minutes.

5.1.3 Use Cases

This section presents the use cases the participants were asked to work on. It starts with the description of each use case, then the required ROS specifications are explained and finally flowcharts for better understanding are presented. The use cases were designed to be compareable in terms of complexitivity, which in this context is quantifed by the amount, type and distribution of different ROS communication patterns as well as the total amount of ROS message calls.

Learning a new object

The first task the participants were asked to work on was a behaviour, which is already implemented on HOBBIT: learning a new object. The use case can be described as follows: First HOBBIT should grab the turntable from its storing position. Then a message on its tablet should be shown to ask the user to put an object on the table. After the user confirmed the placement, HOBBIT should look at the object on the turntable and tell the user "I'm learning a new object" via its tablet interface. The table should turn clockwise first, before the user should be asked to place the the object upside down on the table. Again, the robot should wait for confirmation, then telling "I'm learning a new object" wihle rotating the table counterclockwise. After that, the user should be asked to remove the object and confirm this. Then HOBBIT should look straight, store the table and ask for the name of the object. Finally HOBBIT should show a happy emotion and tell "Thank you, now I know what X is", where X is the name of the object. The whole desired workflow is visualed in Figure 5.1. Furhter, Table 5.1 breaks down the complexitivity of the first use case. It is necessary to publish to two different topics, implementing one action client and calling one service.

Action	Type	Specification	Calls
Move head	Topic	/head/move	2
Show emotion	Topic	/head/emo	1
Move arm	Action	hobbit_arm	4
User interaction	Service	/MMUI	7

Table 5.1: ROS patterns used for implementing first use case

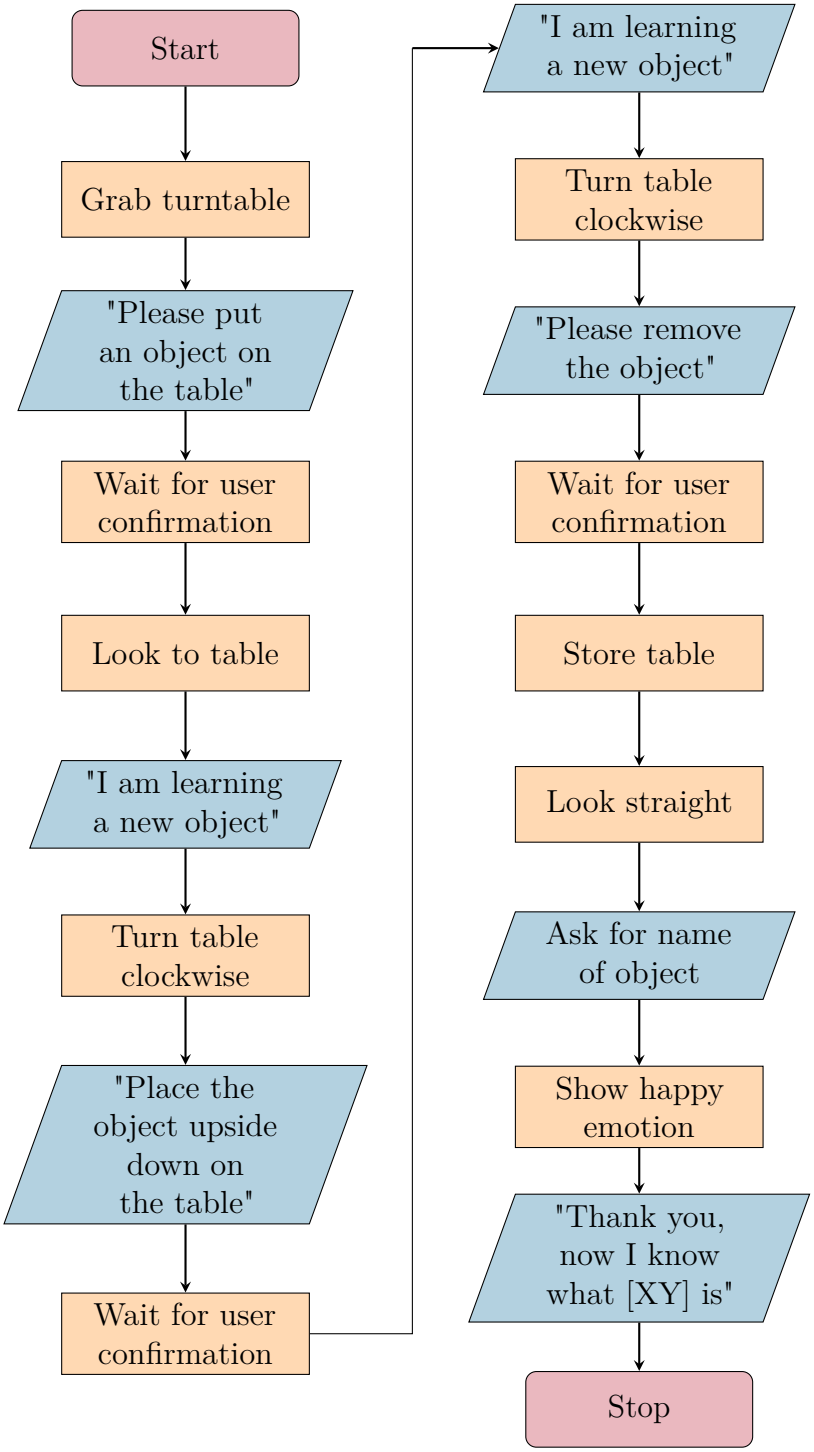


Figure 5.1: Flowchart of first use case

Bringing objects from another person

The second task was to implement a programm to ask the user repetitively if HOBBIT should bring an object from another person, which is located at another place. First, the user should be asked which objects should be picked up (e.g. "Which object do you want?"). The user then should use the robot's tablet to enter the name of the requested object. After that, HOBBIT should navigate to the second person. The exact location, specified by the coordinates and pose, was provided in advance to the participants. The second user then should be asked to handover the desired object. If it is answerd positively, the object should be placed on HOBBIT tray and the robot navigates back to its previous location telling the user "Here you are" and placing the object on the table. If the object has not been handed over, an appropriate message should be displayed on the tablet (e.g. "I'm sorry, your partner couldn't handover the object") after navigating back. Afterwards the user should be asked, if HOBBIT should bring another item. The whole procedure should be performed as long as the user does not request another object. After the final decline HOBBIT should show a happy emotion again. For a better understanding Figure 5.2 provides the flowchart of this use case.

Table 5.2 breaks down the complexitivity of this use case. It is necessary to publish to one topic, make calls from two different action clients and use one service. So the number of different ROS communication types is equal to the first use case. The total number of necessary calls differ slightly (14 in the first use case, 13 in the second). This should be compensated by asking participants to implementing just one action server for the first use case and publishing to two topics (compared to two actions and one topic for the second use case), since this actions are seen as the most difficult ROS pattern and topics the most simple ones.

Action	Type	Specification	Calls
Show emotion	Topic	/head/emo	1
Move arm	Action	hobbit_arm	4
Navigation	Action	move_base_simple	2
User interaction	Service	/MMUI	6

Table 5.2: ROS patterns used for implementing first second case

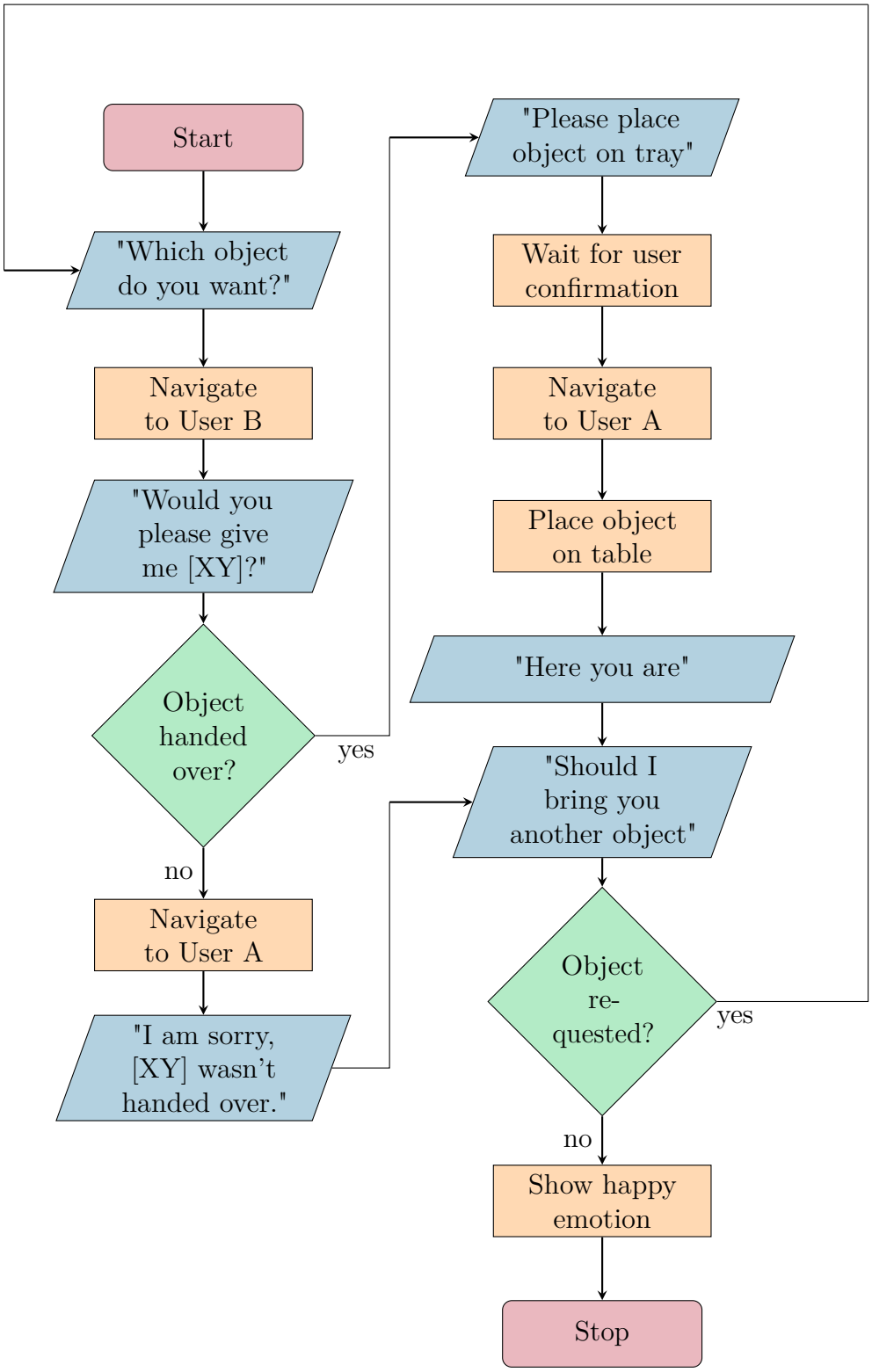


Figure 5.2: Flowchart of second use case

5.1.4 Measurements

Two types of measurements were included into the evaluation process. First, quantitative measurements were performed while the partcipants were working. Then they were asked to answer a questionnaire after they completed both tasks. The questions can be classified into four categories: demographic factors (DF), experience questions (EQ), feedback questions (FQ) and workload questions (WQ). Each of them are discussed in the following.

Quantitative Measurements

While working on the use cases two measurements are performed for each participant: total time (the time to accomplish tasks) and pauses. A duration is considered as *pause* if the programm does not change within two seconds. The measurements are performed for both tasks, working with Rockly and coding. For this purpose a routine using the Blockly API was implemented, which listens to changes in the tool's workspace - i.e. creating, updating or deleting blocks. The provided editor was also customized with a feature, which recorded the timestamps of changes. At the start of each task the participants had to click on a *Start* button, which started the timer. It was stopped, when the participant clicked the *Stop* button. The implemented function then submitted the following information:

- Participant ID: a randomly generated alphanumeric ID to assign submissions
- Total time taken to accomplish task
- Timestamps of pauses
- Type of submission (tool or code)
- Content of submission (e.g. full code)

Demographic Factors

Demographic questions are designed to help survey researchers determine what factors may influence a respondent's answers, interests, and opinions. Participants of this experiment were asked the following questions:

- *DF1*: What is your age?
- *DF2*: What is your highest qualification?

- *DF3*: What is your current employment status?
- *DF4*: What is your current field of work/study?

DF1 was asked to roughly determine the participant's knowledge and experience. The possible answers ranged from *under 18* to *above 44*. A more detailed information regarding the general knowledge of the participant is delivered by question *DF2*. The possible answers were: less than high school, high school diploma or equivalent degree, bachelor's degree, master's degree, higher than master's degree, no degree. Questions *DF3* and *DF4* were asked to determine whether a participant has experience in any related field and, if so, how much. Additionally, the answers to the demographic questions are used to check whether the participants have met the assumed requirements (Section 5.1.5).

Experience Questions

The main motivation for having experience questions is that they enable the correlation between results of the experiment and the experience for each participant. The following questions were asked to collect this information:

- *EQ1*: How much experience do you have with ROS?
- *EQ2*: How much experience do you have with programming in C++/Python?
- *EQ3*: How much experience do you have with programming in general?
- *EQ4*: How much experience do you have with block-based VPLs (e.g. Blockly)?

A scale including the answers *none*, *moderate* and *expert* was used. The answers can easily be mapped onto a numeric scale. The set of answers was also chosen to be that limited to minimize the variability of the answers. The questions are structured according to the level of abstraction. In particular *EQ1* was chosen since the implemented tool provides an environment which should support people without detailed knowledge of ROS. Experience with ROS therefore could influence the outcome of the experiment. Another influence could be the programming experience in C++ and Python, which are the main languages for programming ROS nodes (see *EQ2*). Especially when comparing the results of the coding task, this could explain differences. General programming experience may not influence the coding task, but could decrease the effort when working with the tool - e.g. when looking at the participants' pauses (see *EQ3*). Participants with experience in block-based languages are expected to find their way through the Rockly workspace more easily than novices (see *EQ4*).

Feedback Questions

The motivation for asking feedback questions is to get subjective feedback from each participant which should help, together with the analysis of the other mentioned measurements, to answer all the goal questions presented in Section 5.1.1. The following feedback questions were asked:

- *FQ1*: Do you think such a tool saves time compared to your current approach?
- *FQ2*: Do you think such a tool allows more flexibility compared to your current approach?
- *FQ3*: Do you think such a tool provides scalable solutions for tasks you are facing in your work?
- *FQ4*: Do you think the usage of the tool is intuitive?
- *FQ5*: Do you think the clustering of the blocks into categories is reasonable?
- *FQ6*: Do you think the the tool provides a pleasant way to maintain programs?

The scale used for the questions included the following possbile answers: *Strongly disagree*, *Disagree*, *Neutral*, *Agree*, *Strongly agree*. Again, the motivation for choosing this set of answers was to minimize the variability of answers. Especially when looking on *GQ1*, the feedback questions play a crucial part in this experiment.

FQ1 should find out if such a tool is seen as an assistance for topics the participants are facing during their work. As mentioned, programming a robot is not necessarily the main topic in research projects. Having a tool, which speeds up subtasks, allows to spend more time on more crucial tasks. The subjective answers to this question can also be compared to the result of the measurements.

FQ2 targets the flexibility of the tool, e.g. when thinking of creating different demos. It basically gives insights on how much complexitivity can be put into demos when implementing them with the tool.

Similar to that question, *FQ3* would also affect the judgement if the tool is seen as an improvement or not. If solutions can not be implemented flexible and

scalable enough, more research is required to further improve the presented tool.

Finally, *FQ4*, *FQ5* and *FQ6* should find out how intuitive the usage of the tool is regarding a participant's subjective feeling. Having an intuitive tool and workflow helps a visual programming tool to be accepted by potential users. Experienced users probably would tend to use it over self-implemented code because it saves time and unexperienced users may faster understand programming concepts.

Workload Questions

Besides getting direct feedback from participants another approach, judging how useful and intuitive a tool is, can be measuring the workload a participant felt during working with it. Because of that, another set of questions was included in the evaluation process. It is based on the NASA Task Load Index (NASA-TLX)[32], which consists of six subscales that represent somewhat independent clusters of variables: mental demand, physical demand, temporal demand, frustration, effort and performance. The assumption is that some combination of these dimensions are likely to represent the workload experienced by most people performing most tasks. The following four workload questions were asked for both tasks, working with the tool and writing a code:

- *WQ1*: How mentally demanding was the task?
- *WQ2*: How much time pressure did you feel during the task?
- *WQ3*: How hard did you have to work to accomplish your level of performance?
- *WQ4*: How insecure, discouraged, irritated, stressed and annoyed were you?

The number of questions was limited to four, because two clusterd variables - namely, physical demand and performance - are not seen to be required in the scope of the chosen experiment setup. First, it can be expected that working with a software tool is not considered to be physically demanding. Second, the participants could not rate their performance properly because debugging and testing is not supported by the experiment setup. Further adaption was done in respect of the number of possible answers. In the official NASA-TLX paper and pencil version[1] increments of high, medium and low estimates for each point result in 21 gradations on the scales. To minimize the variability of

[1] https://humansystems.arc.nasa.gov/groups/TLX/downloads/TLXScale.pdf

answers only a scale with five gradations (*Very low*, *Low*, *Medium*, *High*, *Very high*) was used.

5.1.5 Participants

Since the developed tool targets people with basic to moderate programming skills, this was a major requirement to be included in the study in order to get reasonable results. Furthermore, partcipants had to have at least basic knowledge of ROS and its communication patterns in order to compare their efforts when working on the different tasks. To fulfill this conditions the study was conducted within the scope of the course "Selected Topics - Robotics and Computer Vision" course held by the V4R group of the automation and control institute at the TU Wien. Participants were all undergraduate students and received course credits for their participation. Ten participants (all males) completed the study. Eight of them were between 18 and 24 ages old, two were between 25 and 34. The detailed results of the demographic factors are listed in Appendix D.

5.2 Results

This section presents the results of the experiment. At first a quantitative analysis is performed comparing the results of the Rockly tool against the traditional coding approach. After that follows the examination of the questionnaire, including the experience, feedback and workload questions as well as the textual feedback.

5.2.1 Quantitative Analysis

As described in Section 5.1.2 the participants were provided with a customized editor for each task, which collected several data, including pauses, task execution time and the solution itself. This data is used to perform a quantitative analysis on the results. First, the solutions are analyzed in order to evaluate whether using the developed tool increases the sucess rate. And if so, if solutions can be found even faster by using it. Then a deeper look is provided by analysing the workflow of the participants. This is achieved by analysing the pauses they made during working on the tasks.

Success rate

At the first step the overall success rate for each use case is examined. This simply means that the submitted solution of each participant is checked for

semantic correctness as well as for feasibility. Table 5.3 and Table 5.4 show the outcomes of both use cases for each participant's solutions with and without the Rockly tool. 90% of the participants submitted at least an executable code when using Rockly, which means all used blocks were connected syntactically correct within the workspace of blocky. Furthermore, seven out of the ten participants were able to produce a program which would lead to correct behaviour of the robot. The success rate of participants using the manual approach - meaning they have to manually implement the code - is 0. Moreover, nobody was able to provide an executable solution, i.e. the codes of all partcipants were syntactically wrong. Furthermore, none of the participants' coding solutions included all necessary ROS communication calls (as listed in Table 5.1 and Table 5.2). The tables also show the time each participant needed to come up with a solution along with the average task completion time for each task. If only the successful cases (executable and correct behaviour) are considered, the average task completion time for both use cases is 13:56 when using Rockly.

Participant	Executable	Correct	Time
A	✗	✗	20:37
B	✗	✗	30:51
C	✗	✗	19:48
D	✗	✗	30:54
E	✗	✗	28:13
	0%	0%	26:05

(a) Coding solutions

Participant	Executable	Correct	Time
F	✓	✓	10:52
G	✓	✓	10:55
H	✓	✓	08:16
I	✓	✓	14:44
J	✓	✓	06:37
	100%	100%	10:17

(b) Tool supported solutions

Table 5.3: Outcomes of participants solutions for first use case[2]

Pauses

As discribed in Section 5.1.2, both editors tracked changes during the participants were working. Pauses are identified as such if there was no change of the programm within at least two seconds. The distributions of the pauses, clustered into 5-seconds-interval-bins, are shown in Figure 5.3. For both tasks the majority of the pauses lasted for under 10 seconds - 68.7% when the participants were manually coding and 72.3% when they were supported by Rockly. But there is a big difference when considering longer pauses: 7 pauses with a duration of more than 60 seconds were recorded when participants worked with Rockly, which is equal to only 1%. This is significantly less than the

[1]Time format: MM:SS

Participant	Executable	Correct	Time
F	✗	✗	29:06
G	✗	✗	30:55
H	✗	✗	29:37
I	✗	✗	25:59
J	✗	✗	28:18
	0%	0%	28:47

(a) Coding solutions

Participant	Executable	Correct	Time
A	✓	✗	22:18
B	✓	✗	24:53
C	✗	✗	13:04
D	✓	✓	23:38
E	✓	✓	22:25
	80%	40%	21:16

(b) Tool supported solutions

Table 5.4: Outcomes of participants solutions for second use case[2]

number (59) and share (7.4%) when looking at the pauses for the same interval when participants were coding. To demonstrate this, optionally, a histogram using bigger sized 20-seconds-interval-bins is shown in Figure 5.4. This could be influenced by the fact, that the participants had to go through the ROS reference of the robot in order to figure out the required commands and messages. Furthermore, it is reasonable to assume, that they also needed time to get familiar with the implementation of the ROS communication patterns. Overall, the longest recorded pause was 117.5 seconds when finding a tool supported solution and 1093.1 when there was no support.

Finally, Figure 5.6 and Figure 5.5 present the workflow diagrams for both tasks. The pauses are normalized for all partcipants along the x-axis and the y-axis shows the corresponding cummulated share of the pauses' durations. For example, the workflow of User C (black line in Figure 5.6) indicates, that the participant made few pauses, while the duration of one pause contributes nearly a hundred percent to the total pause time. When comparing both graphs, it can be observed that the workflow follows a very similar, linear pattern across all participants. This implies that Rockly enabled the participants to work more straight forward compared when they were coding.

Figure 5.3: Distribution of pauses clustered into 5s intervals cummulated for both use cases (>200s not shown)

Figure 5.4: Distribution of pauses clustered into 20s intervals cummulated for both use cases (>200s not shown)

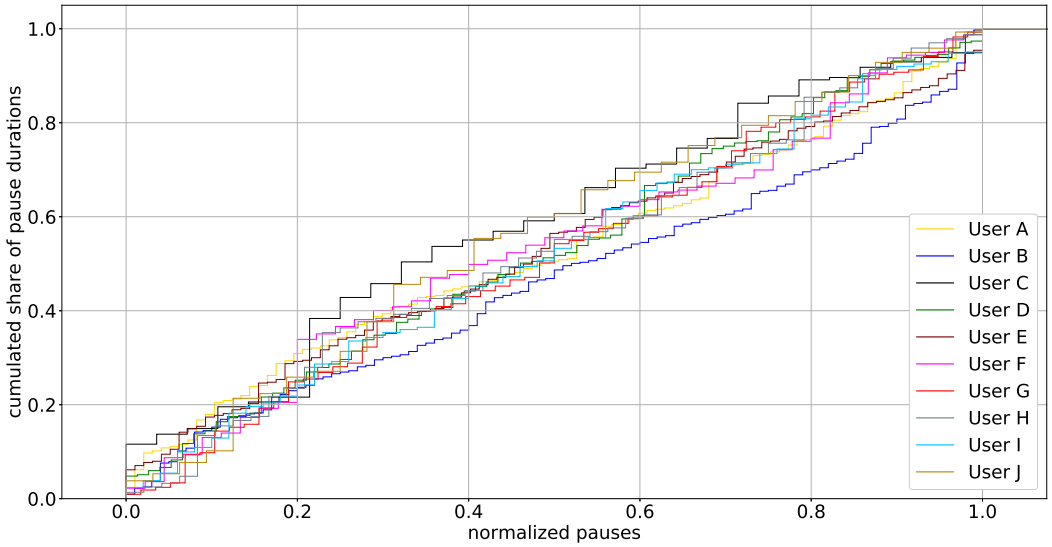


Figure 5.5: Workflow diagrams for participants' tool supported solutions (both use cases)

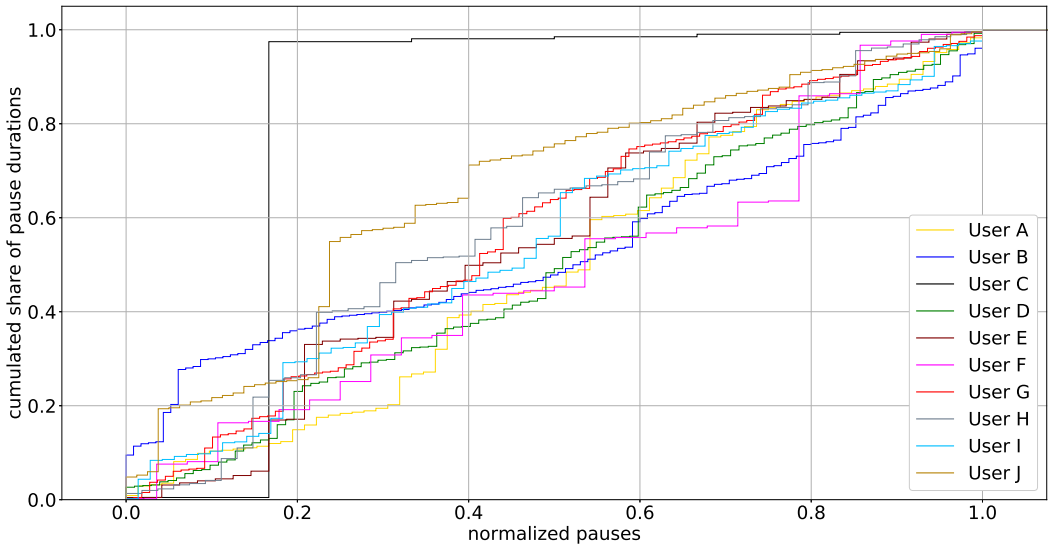


Figure 5.6: Workflow diagrams for participants' coding solutions (both use cases)

5.2.2 Self-report Questionnaire

After completing both tasks, the participants were asked to answer a questionnaire containing demographic, experience, feedback and and workload questions. The following sections presents the results for the most important categories. The results of the demographic factors are listed in Appendix D.

Experience

As described in Section 5.1.4 the experience levels of the participants are mapped onto a numeric scale, which can be seen in Table 5.5. The results of the questions are shown in Figure 5.7 and Table 5.6 lists the average experience and variance among all participants for each question. Almost all participants have *moderate* experience with ROS (*EQ1*) and everyone has a *moderate* experience with the relevant programming languages (*EQ2*). The experience level regarding programming in general (*EQ3*) is slightly higher, while only three have had experience with any VPL before attending the experiment (*EQ4*).

Experience	Value
None	1
Moderate	2
Expert	3

Table 5.5: Experience scale to numeric value mapping

Question	Average	Variance
EQ1	1.90	0.09
EQ2	2.00	0.00
EQ3	2.20	0.16
EQ4	1.30	0.21

Table 5.6: Mean and variance per experience question

Feedback

For analysis of the feedback questions a numeric mapping of the feedback scale defined in Section 5.1.4 was used and is listed in Table 5.7. Figure 5.8 shows the results of the feedback questions and Table 5.8 shows the question-wise

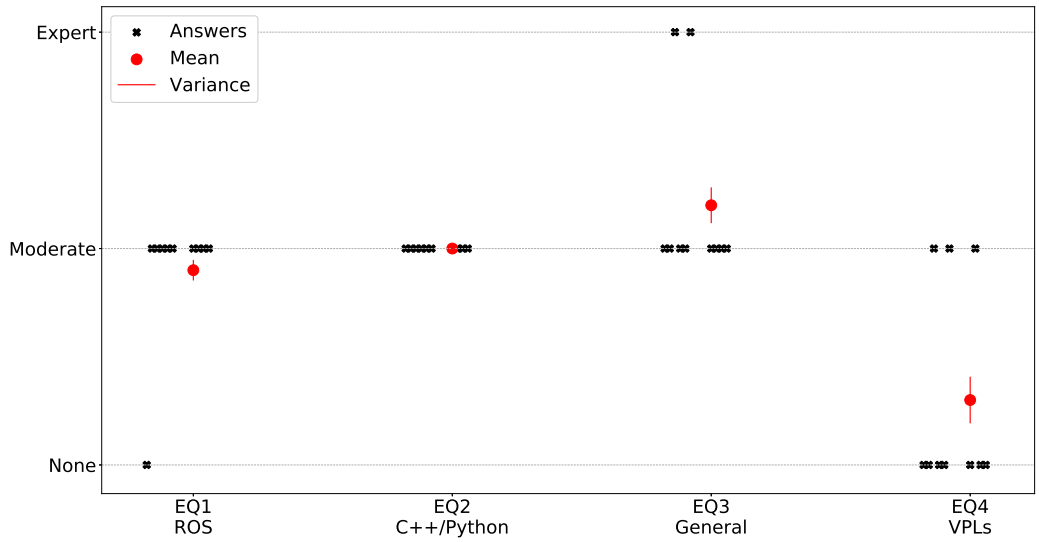


Figure 5.7: Results of the experience questions defined in Section 5.1.4

summary. The participants agree that using the tool saves time compared to their current approach (*FQ1*), think the usage of it is intuitive (*FQ4*) and the clustering of the blocks is reasonable (*FQ5*). Most of the participant also agree that the tool provides a pleasant way to maintain programs, but there is broad variance (*FQ6*). An even broader variance is found when asking, if the tool provides scalable solutions, so there is a *neutral* overall feedback for *FQ3*. The participants tend to *disagree* that such a tool allows more flexibility compared to their current approach (*FQ2*). The textual feedback goes along with this. One participant noted, that lacking the option to create multiple nodes is a big disadvantage.

Feedback	Value
Strongly agree	5
Agree	4
Neutral	3
Disagree	2
Strongly disagree	1

Table 5.7: Feedback scale to numeric value mapping

Question	Average	Variance
FQ1	3.9	0.49
FQ2	2.2	0.36
FQ3	2.9	0.89
FQ4	4.2	0.36
FQ5	4.0	0.2
FQ6	3.4	0.84

Table 5.8: Mean and variance per feedback question

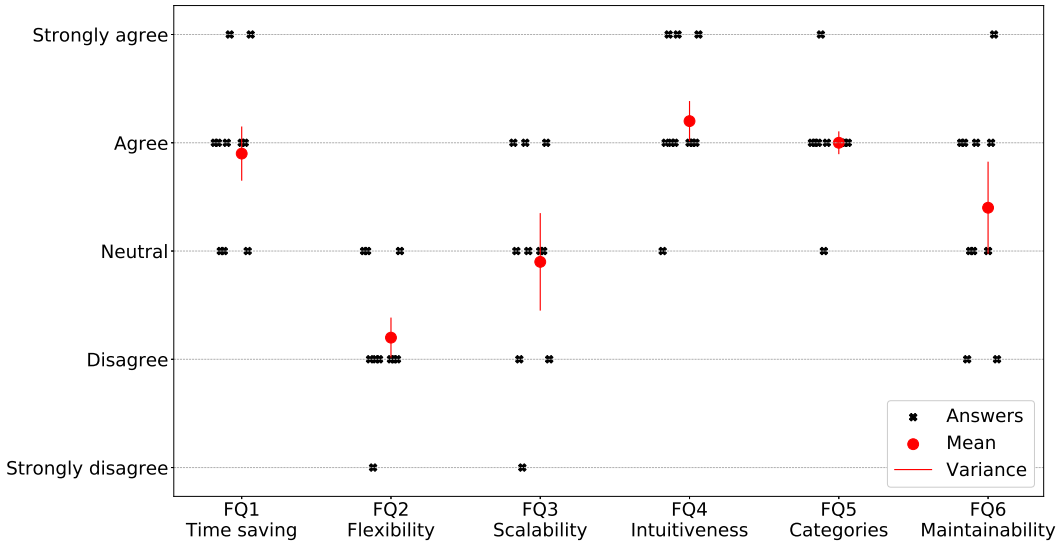


Figure 5.8: Results of the feedback questions defined in Section 5.1.4

Workload

The participants were asked four different questions regarding their workload (see Section 5.1.4) for each task they faced. The mapping for the possible answers are listed in Table 5.9. Table 5.10 shows the calculated mean and variance per question and Figure 5.9 a graphical summary for each participant's answers. It can be stated, that for each subscale chosen from the NASA-TLX template, the workload tends to be *low* when finding a solution with the developed tool and *high* without it.

Workload	Value
Very low	1
Low	2
Medium	3
High	4
Very high	5

Table 5.9: Workload scale to numeric value mapping

	Code		Tool supported	
Question	Average	Variance	Average	Variance
WQ1	3.8	0.56	2.4	0.84
WQ2	4.0	1.0	2.1	1.09
WQ3	3.6	0.44	2.2	0.96
WQ4	3.7	0.81	1.9	0.89

Table 5.10: Mean and variance per workload question

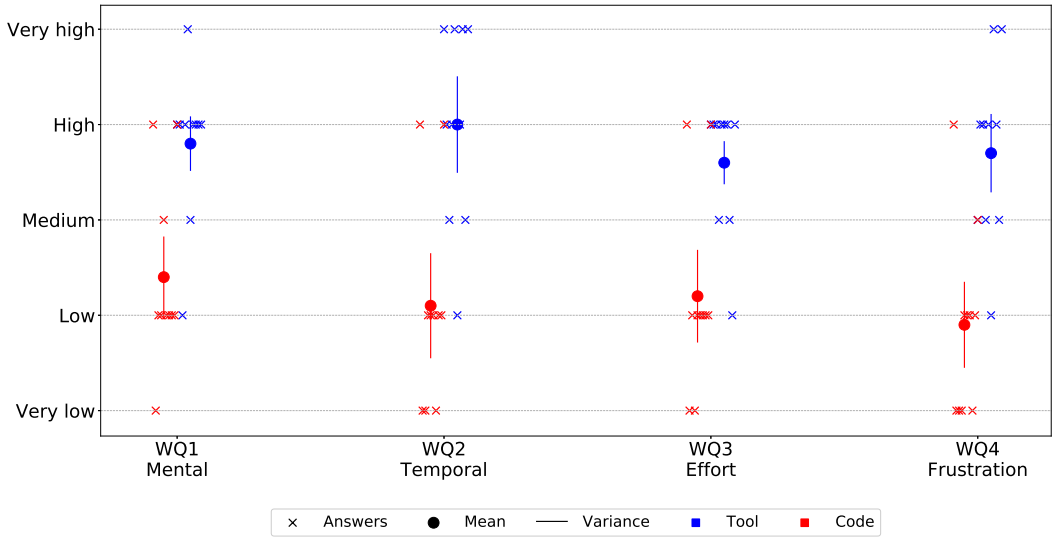


Figure 5.9: Results of the workload questions defined in Section 5.1.4

5.3 Discussion

In the following the goal questions of the experiment as stated in Section 5.1.1 are answerd and discussed. The primary goal of the evaluation was to find out if the developed tool is seen as an improvement compared to the traditional coding approach (*GQ1*). Then, the workflow should be analyzed, in order to evaluate if participants make less pauses when using the tool (*GQ2*), as well as a general comparison of the task completion times (*GQ3*). Finally, the usability of Rockly should be evaluated by asking, if the usage of it is intuitive (*GQ4*).

GQ1 - The results suggest that the tool is seen as a more effective solution (time-wise) compared the traditional coding approach

Analysing the answers of the feedback questions *GQ1* can be answerd positively considering time effectiveness. The lack of flexibility (*FQ2*) may lead to the conclusion that the tool is not suitable for complex applications. Moreover, if considering *FQ2* and *FQ3* as representing questions for this case, there is a negative correlation between experience and feedback (Figure 5.10). Because of the small sample size two data points occured multiple times. They are marked as *multiple votes* in the diagram. However, this disadvantage was already considered during the architectural design of the tool (see Section 3.2.4).

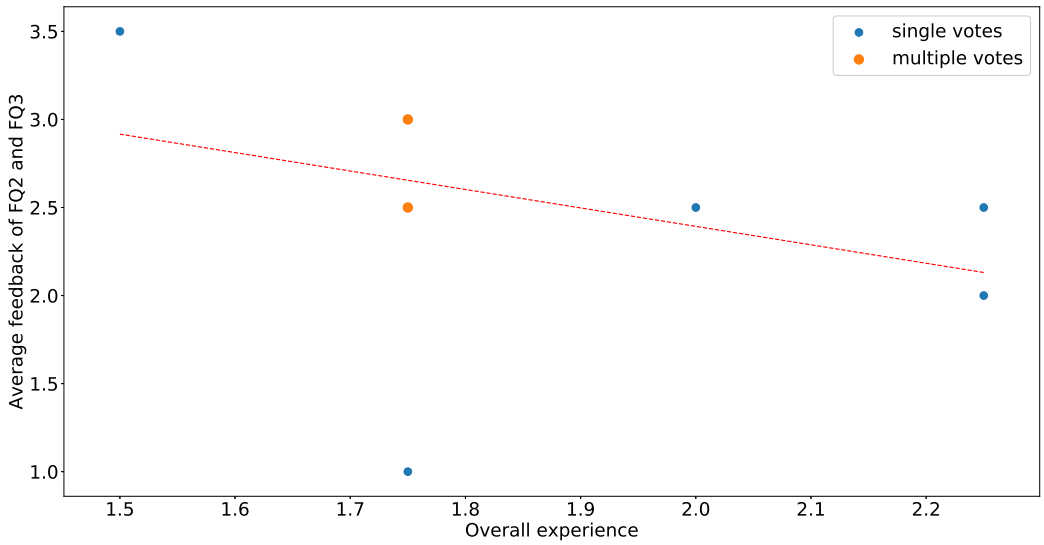


Figure 5.10: Experience to feedback correlation

GQ2 - Participants made less pauses when they used the tool compared to coding

When looking at the results of the quantitative analysis (Section 5.2.1) it can be stated, that working with the tool ensures a more fluent workflow than coding. This is supported by the results of the workload questions (Figure 5.9). Therefore, *GQ2* can be answerd positively.

GQ3 - It took partcipants significantly less time to find a solution when they were supported by the tool

The question, if using the tool saves time compared to the coding approach (*GQ3*), can be answerd by using the results of the total time measurements Section 5.1.4 along with the sucess rate of the participants for both tasks (Section 5.2.1). It took participants significantly less time to come up with a tool supported solution than a code based solution. The average task completion time cumulated for both use cases was 15:46 when using Rockly and 27:26 without it. Moreover, all of the tool supported solutions were executable while 100% (first use case) and 40% (second use case), respectively, are implement correctly. In contrast, 0% of the submitted code solutions were executable and correct. Therefore, *GQ3* can be answerd positively.

The fact, that the use cases were not hundred-percent identical, must be considered too. Although it was taken care of that during designing the study (Section 5.1.2), the results implies that there may be a bias. Participants working with the tool for the second use case needed more then twice the time to come up with a solution than the ones working with it during the first one. A possible reason for this is the task description, as this was claimed by a participant via the textual feedback. Additionally, the low number of partcipants (N=10) could have an influenced the results.

GQ4 - Partcipants consider the tool to be intuitive

GQ4 can also be answerd positively, based on the feedback questions *FQ4*, *FQ5* and *FQ6*. Most of the partcipants agreed that the tool is intuitive, the clustering of the blocks into categories is reasonable and the tool provides a pleasant way to maintain programs. The workflow analysis (Figure 5.5 and Figure 5.6) also indicates, that the participants were abble to work more streamlined compared when they were supported by Rockly.

Along with the other observations, it can be concluded, that Rockly enabled all partcipants to work more continuously to find a tool supported solution significantly faster than by coding, which in most cases was executable and

correct.

6 Conclusion and Future Work

This chapter summarizes the design as well as the implementation of Rockly, the tool presented by this work. Additionally, the most important results of the evaluation study are mentioned along with possible further improvements to enable more flexible and scalable applications.

6.1 Conclusion

This thesis presented the design, implementation and evaluation of a graphical user interface for programming ROS-based robots - called Rockly. Based on the requirements a suitable design and architecture was choosen according to the following points: complexity and sclability of applications, required robot and technical know-how, implementation affort, usability and maintainability. The first option was to provide a customized API abstracting ROS communication patterns for Python and/or C++. This approach would have given the opportunity to use it even for complex applications. A similar result was found when considering to create an API on-top of the SMACH package. Since the required technical skills are high for both of this approaches, developing a graphical user interface, build on a block-based VPL framework (Blockly), was then considerd to be the best fit.

A server-client architecture was chosen to provide remote control when programming the robot. The front end on the client side was implemented as a web-based application in order to not require the user to install any additional software. The back end of the tool, running on the robot, coordinates the communication and code execution. The user interface consits of a section for managing demos, an graphical editor for creating demos and a block configurator to manage and create customized blocks. Especially the last can be seen as the major improvement compared to commonly used graphical programming tools, since it enables the user to create blockls for any ROS-based robot without requiring any knowledge about Blockly and re-building it.

A user study, including the implementations of two use cases, was performed to evaluate the tool regarding its ability to support users while programming a robot, namley HOBBIT. First, the participants had to implement a solution,

which let HOBBIT learn a new object. Half of the group worked with the tool, the others had to find a solution by traditional coding. Then, the roles were switched, and participants were asked to implement a behaviour to let HOBBIT repeatedly ask a user to pick up an object. During the implementation the timestamps of pauses (indicated as such if there was no change within the workspace for at least two seconds) and the task completion time were measured. After completing both tasks, self-report questionnaires were filled out by the group.

Although the number of participants (N=10) may have been statistically not significant, the results indicate, that a tool supported solution was found much faster and with less errors than using the traditional coding approach. The overall success rate when using Rockly was 60%, compared to 0 when participants had to code. On average, when using the tool, participants were nearly 16 minutes faster than coding for the first use case and about 7,5 minutes faster for the second one. Moreover, it was observed that when using the tool the workflow was more streamlined, indicated by the number, durations and distribution of pauses they made during work.

Based on the evaluation it can be stated, that using Rockly basic demonstration programs - such as the presented use cases - can be implemented faster compared to the classic coding approach. Using the tool requires less programming knowledge as well as little robot specific expertise to build such demos. Furhtermore, the results also implies, that Rockly is an intuitive tool which could be used for educational purposes.

6.2 Future Work

All the presented functionalities and evaluation were implement on the HOBBIT PT2 robot only. Therefore, the first step to generlize the capabilities of Rockly for other robots, the source code must be transferred to them and tested there. This was out of scope of this thesis. Second, the evaluation study setup only included people with programming and ROS experience. In order to evaluate if and how programming and robotics novices profit from Rockly, further studies with proper setups are reasonable.

Using VPLs, especially block-based ones, allows users to become familiar with programming easily. But hence of a higher abstraction level flexibilty and scalablity may be restricted as well as the ability to use it for complex applications. Right now it may be mainly used for basic robot programming tasks, for example for educational purposes or quick demos. In order to enlarge

its field of application not only commands are necessary, meaning the tool should also be able to create subscriber nodes for listening to ROS topics. This, for example, can be achieved by extending the functionality of the block configurator by introducing subscriber blocks.

Thinking of more complex applications it also would be benefitial if multiple nodes can be created and managed by the tool. While maintaing the usability of the tool a possible approach to implement this feature, an additional module on top can be considerd instead of extend the functionalities of the existing modules. So nodes can be still created with the tool as is was developed during this thesis and only need to be run parallely. Also some asynchronous functionality can be considerd in order to create even more complex systems and establish VPLs also in more advanced fields of robotics.

Appendices

A List of Abbreviations

API Application Programming Interface

IDE Integrated Development Environment

NASA-TLX NASA Task Load Index

PLC Programmable Logic Controller

REST Representational State Transfer

ROS Robot Operating System

SDK Software Development Kit

SFC Sequential Function Chart

UI User Interface

VPL Visual Programming Language

XML Extensible Markup Language

B Block configuration manual

This appendix shows step-by-step examples for configuring custom blocks using Rockly's block configurator for each of the following ROS communication patterns:

- Topic
- Service
- Action

Custom block publishing to a topic

This example creates a custom block, which sending a message to a ROS based robot. The filled form is shown in Figure B.1, the final design is shown in Figure B.2.

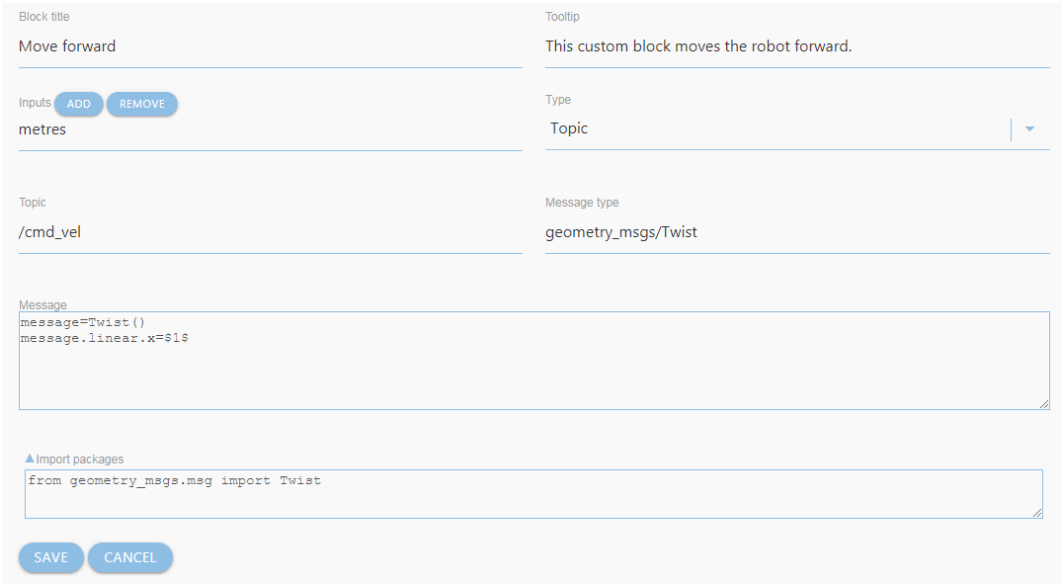


Figure B.1: Filled form to create a custom block publishing to a topic

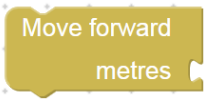


Figure B.2: Resulting custom block using the block configuration shown in Figure B.1

```
1  #!/usr/bin/env python
2  import HobbitLib
3  import rospy
4  from geometry_msgs.msg import Twist
5
6
7  if __name__ == '__main__':
8    try:
9      DemoNode = HobbitLib.node('DemoNode')
10
11     message=Twist()
12     message.linear.x=1
13     HobbitLib.importMsg('geometry_msgs.msg','Twist')
14     DemoNode.publishTopic('/cmd_vel', 'Twist', message)
15
16   except rospy.ROSInterruptException:
17     pass
```

Listing B.1: Examplary generated code using the block shown in Figure B.2

The given title and inputs of the block are can be observed, when looking at the block. The tooltip only appears on mouseover events. All other information is used to generate the code. The value of an input - which is passed by connecting the corresponding input - can be used for the message creation by putting a placeholder `$n$` (with n being the n-th input in the list - counting top down) to it. It is possible to put any code to the *message* field, but it is necessary that it includes an assignment of the `message` value. Assuming the value *1* is passed to the input, the code shown in Listing B.1 will be generated.

Custom block calling a service

Within this section the creation of an example block, which calls a service, is shown. The block can be used to ask a question to the the user and use the response as output, so that the block can be connected as input to another block. The configuration form for this block is shown in Figure B.3.

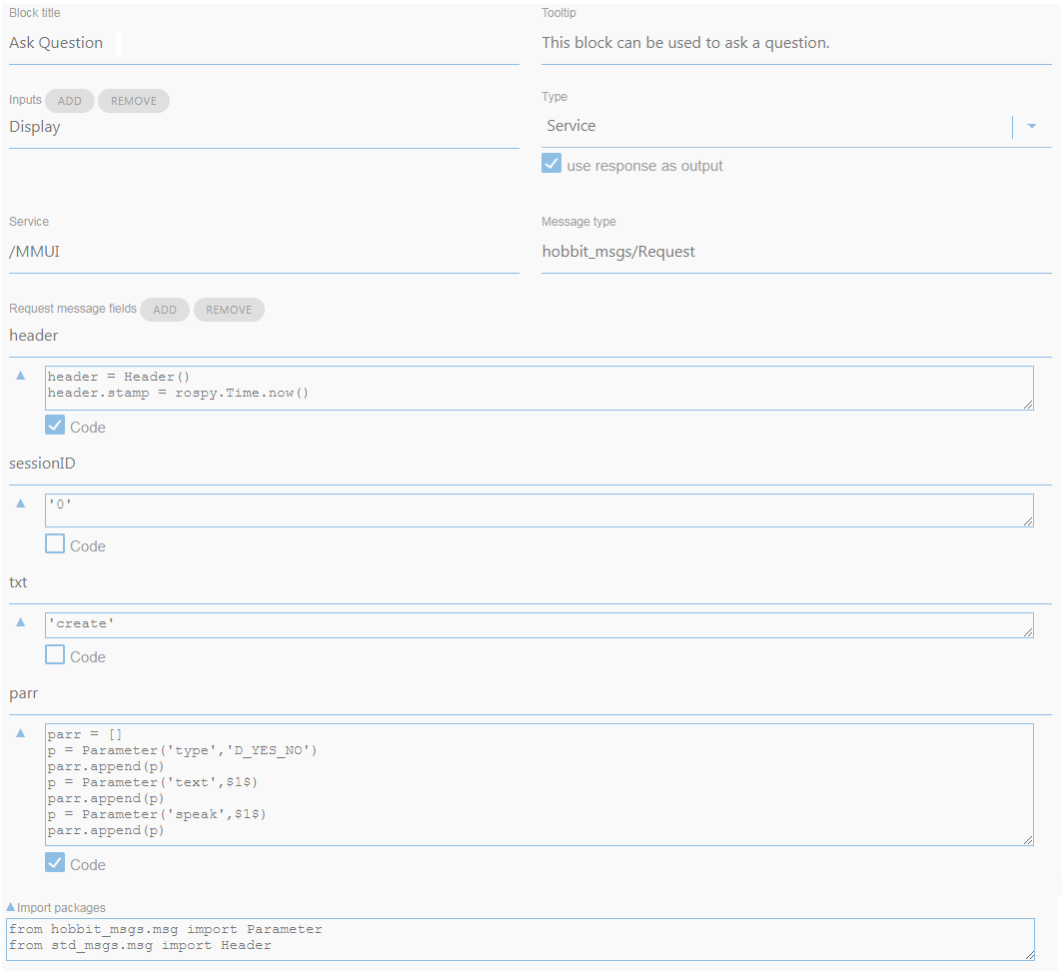


Figure B.3: Filled form to create a custom block calling a service

Besides the choice to use *Service* as communication pattern and pass the service's response, there's no noteworthy difference compared to Section B. In the service-specific section first the service's name (*/MMUI*) and message type (*hobbit_msgs/Request*) are set, then the request message fields are configured. There are two different ways of doing that:

- providing a key-value pair, or
- using a code block.

In the first case the key-value pair is just translated into a String containing the given info. If the latter one is used, it is necessary that the code includes a explicit assignment of the corresponding request message field. In the given

example `parr` is set via a code block. Note that the name of the field is identically the same as the variable's name. By the way, the same effect could be achieved using a key-value pair with the following value:
`[Parameter('type','D_YES_NO'),Parameter('text',$1$),Parameter('speak',$1$)]`.

Again, using the values of the blocks connected to the inputs can be included by putting the placeholder to the corresponding field, as explained in Section B. A exemplary use of the just created custom block is presented in Figure B.4 with Listing B.2 showing the generated code.

Figure B.4: Exemplary us of the custom block created using the block configuration shown in Figure B.3

Custom block using actionlib

Figure B.5 shows how a custom block can be defined, if it is desired that the block should send a goal to an action server. In this case a block for sending a command inputed by an another block, e.g. a text block, is created (Figure B.6). The generated code is shwon in Listing B.3.

The head configuration including the block's title, tooltip and definition of inputs is the same as in the previous sections. By selecting *Action* as a communication type the actionlib specific configuration fields appears including the following parameters:

- Timeout: Maximum time to block before returning. A zero timeout is interpreted as an infinite timeout
- Server: The node handle on top of which we want to namespace our action
- Action name: Defines the namespace in which the action communicates
- Goal: The desired goal message to be sent to action server
- Callback functions: Functions that get triggerd by the action server

As for the other communication patterns, it is possible to put any code to the *Gaol* field, but it is necessary that it includes an assignment of the `goal` value. The callback functions can also include any desired code. The parameters are passed and can be used as shown in Listing B.3.

```
#!/usr/bin/env python
import HobbitLib
import rospy
from hobbit_msgs.srv import Request
from hobbit_msgs.srv import RequestRequest
from hobbit_msgs.msg import Parameter
from std_msgs.msg import Header

def srv21zxtebqdd3i():
  header = Header()
  header.stamp = rospy.Time.now()
  sessionID='0'
  txt='create'
  parr = []
  p = Parameter('type','D_YES_NO')
  parr.append(p)
  p = Parameter('text','Are you happy?')
  parr.append(p)
  p = Parameter('speak','Are you happy?')
  parr.append(p)
  reqparams=(header,sessionID,txt,parr)
  return DemoNode.callService('/MMUI', 'Request',
     reqparams)

if __name__ == '__main__':
  try:
    DemoNode = HobbitLib.node('DemoNode')
    HobbitLib.importMsg('hobbit_msgs.srv','Request')

    print(srv21zxtebqdd3i())

  except rospy.ROSInterruptException:
    pass
```

Listing B.2: Generated code of the block connections shown in Figure B.4

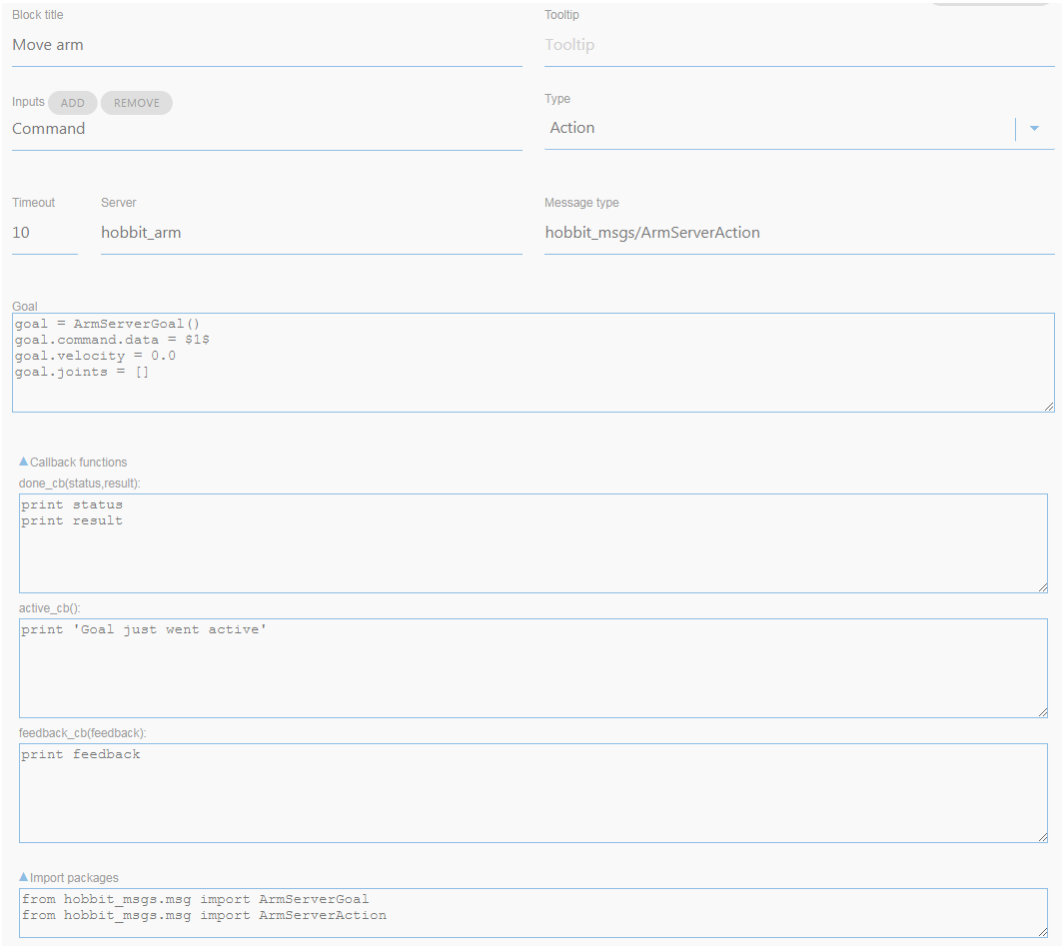


Figure B.5: Filled form to create a custom block using actionlib

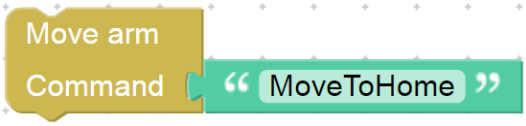


Figure B.6: Exemplary us of the custom block created using the block configuration shown in Figure B.5

```
#!/usr/bin/env python
import HobbitLib
import rospy
import actionlib
from hobbit_msgs.msg import ArmServerAction
from hobbit_msgs.msg import ArmServerGoal

def donecb_8epgqbnm9ho(status,result):
  print status
  print result

def activecb_8epgqbnm9ho():
  print 'Goal just went active'

def feedbackcb_8epgqbnm9ho(feedback):
  print feedback


if __name__ == '__main__':
  try:
    DemoNode = HobbitLib.node('DemoNode')

    goal = ArmServerGoal()
    goal.command.data = 'MoveToHome'
    goal.velocity = 0.0
    goal.joints = []
    HobbitLib.importMsg('hobbit_msgs.msg','
       ArmServerAction')
    client = actionlib.SimpleActionClient('hobbit_arm',
        ArmServerAction)
    client.wait_for_server()
    client.send_goal(goal,done_cb=donecb_8epgqbnm9ho,
       active_cb=activecb_8epgqbnm9ho,feedback_cb=
       feedbackcb_8epgqbnm9ho)
    client.wait_for_result(rospy.Duration.from_sec
       (10.0))

  except rospy.ROSInterruptException:
    pass
```

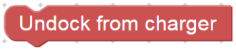
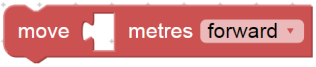
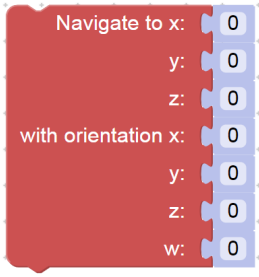
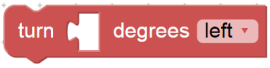
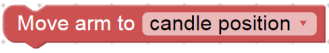
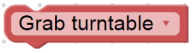
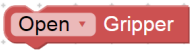
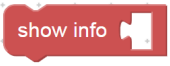
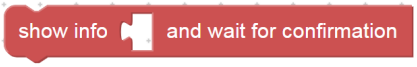
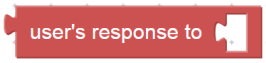
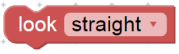
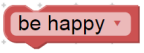
Listing B.3: Generated code of the block connections shown in Figure B.6

C HOBBIT block set overview

The blocks of Rockly' toolbox are divided in the following categories:

- Custom Blocks: all blocks which are created with the block configuraton interface (Section 4.3)
- Move: blocks to move HOBBIT
- Arm Control: blocks, which allows to move HOBBIT's arm
- Interaction: any form of blocks, which allows HOBBIT to communicate with the user
- Logic: collection of Blockly's predefined logic blocks
- Loop: collection of Blockly's predefined looping blocks
- Math: collection of Blockly's predefined math blocks
- Text: collection of Blockly's predefined text blocks
- Lists: collection of Blockly's predefined list blocks
- Variables: a category to create and use variables
- Functions: a category to create and use functions

This section contains an overview of all blocks created for HOBBIT's interface, a description of Blockly's predefined blocks can be found in [33].

Block	Description
	Undock HOBBIT from charger
	Move HOBBIT in the given direction
	Navigate to given pose
	Rotate HOBBIT in the given direction
	Move HOBBIT's arm to the given position
	Perform the given action with the turntable
	Open or close the gripper
	Show an info on HOBBIT's tablet
	Show an info on HOBBIT's tablet and wait for confirmation
	Get user's answer to a yes/no question
	Display the given question on HOBBIT's tablet and get user's answer
	Move HOBBIT's head to the given position
	Set HOBBIT's eyes according to the given emotion

D Results of Demographical Questions

The answers and results of the demographical questions asked during the experiment (Section 5.1.4) are shown in Table D.1.

User	DF1	DF2	DF3	DF4
A	18-24	High school diploma or equivalent degree	Part-time employment	Electrical engineering
B	18-24	Bachelor's degree	Student	Software engineering
C	25-34	High school diploma or equivalent degree	Student	Electrical engineering
D	18-24	High school diploma or equivalent degree	Student	Electrical engineering
E	18-24	High school diploma or equivalent degree	Student	Electrical engineering
F	25-34	High school diploma or equivalent degree	Part-time employment	Electrical engineering
G	18-24	High school diploma or equivalent degree	Student	Electrical engineering
H	18-24	High school diploma or equivalent degree	Student	Electrical engineering
I	18-24	High school diploma or equivalent degree	Student	Electrical engineering
J	18-24	High school diploma or equivalent degree	Student	Electrical engineering

Table D.1: Results of demographical questions for all participants

E Experiment materials

This sections presents all the materials used for the experiment including task descriptions, explanation of the necessary ROS concepts with examples and the necessary ROS specifications. The last two were provided by a webpage.

Task descriptions

First use case: Learning a new object - Coding

The use case which should be implemented can be described as follows: First Hobbit should grab the turntable from its storing position. Then a message on its tablet should be shown to ask the user to put an object on the table. After the user confirmed the placement, Hobbit should look at the object on the turntable and tell the user "I'm learning a new object" via its tablet interface. The table should turn clockwise first, before the user should be asked to place the the object upside down on the table. Again, the robot should wait for confirmation, then telling "I'm learning a new object" wihle rotating the table counterclockwise. After that, the user should be asked to remove the object and confirm the action. Then Hobbit should look straight, store the table and ask for the name of the object. Finally Hobbit should show a happy emotion and tell "Thank you, now I know what X is", where X is the name of the object. The desired workflow is visualed in

Please implement a solution using the provided code editor, which lets Hobbit show the desired behaviour, with respect to the following conditions:

- Do exclusively use the provided editor for implementing your solution
- Start working by clicking the "Start" button of the interface
- Do not close the graphical editor during your work
- Click the "Stop" button when you finished implementation
- Click "Submit" to submit your solution

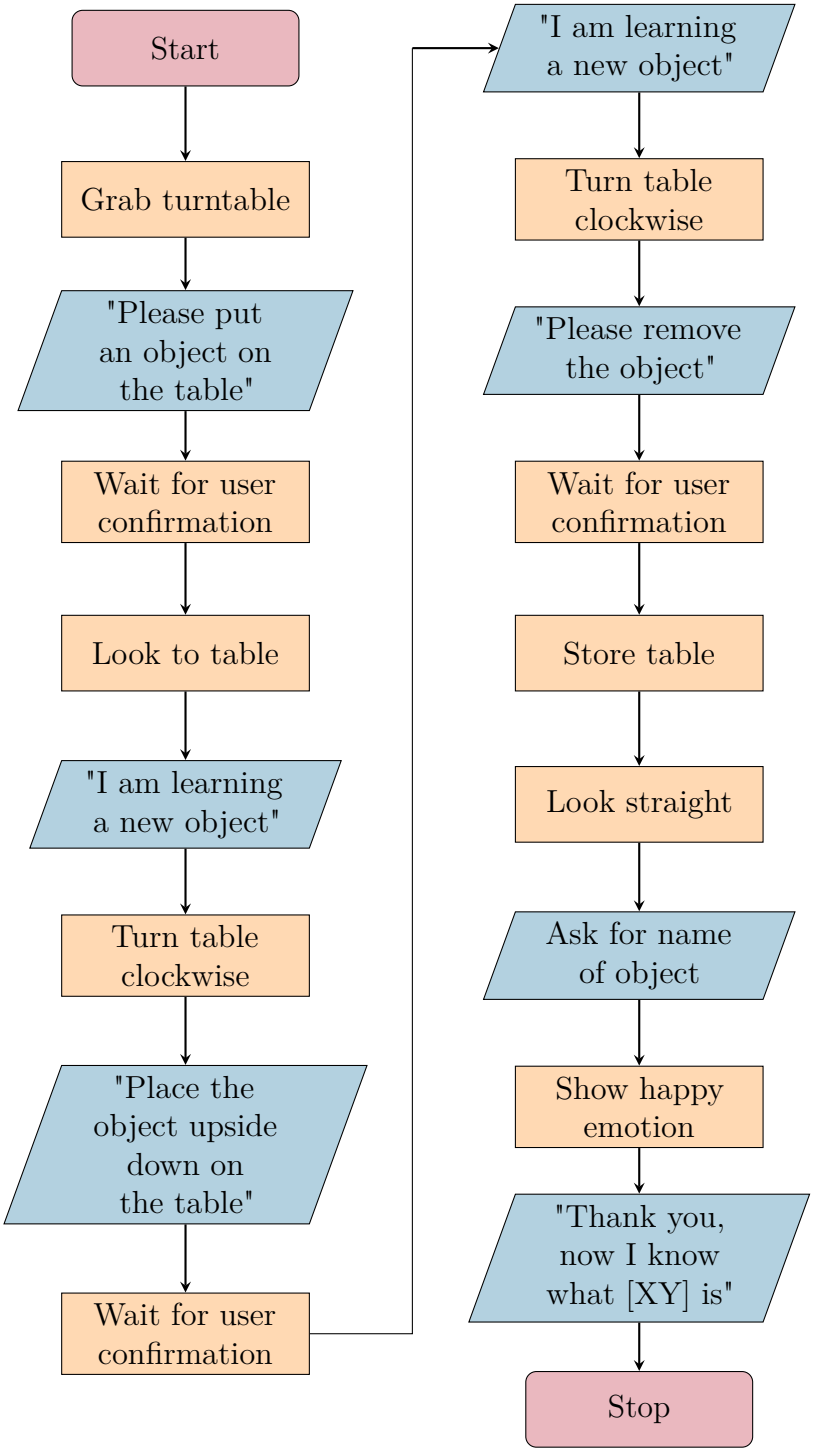


Figure E.1: Flowchart of first use case

First use case: Learning a new object - Tool supported

The use case which should be implemented can be described as follows: First Hobbit should grab the turntable from its storing position. Then a message on its tablet should be shown to ask the user to put an object on the table. After the user confirmed the placement, Hobbit should look at the object on the turntable and tell the user "I'm learning a new object" via its tablet interface. The table should turn clockwise first, before the user should be asked to place the the object upside down on the table. Again, the robot should wait for confirmation, then telling "I'm learning a new object" wihle rotating the table counterclockwise. After that, the user should be asked to remove the object and confirm the action. Then Hobbit should look straight, store the table and ask for the name of the object. Finally Hobbit should show a happy emotion and tell "Thank you, now I know what X is", where X is the name of the object. The desired workflow is visualed in Figure E.1.

Please implement a solution using the Blockly editor, which lets Hobbit show the desired behaviour, with respect to the following conditions:

- Start working by clicking the "Start" button of the interface
- Do not close the graphical editor during your work
- Click the "Stop" button when you finished implementation
- Click "Submit" to submit your solution
- Each block provides a help page - it is accessible via *right click* → *Help*

Second use case: Bringing objects from another person - Coding

The use case which should be implemented can be described as follows: Hobbit should be repetitively asking User A if it should bring an object from User B, which is located at another place. First the user should be asked which objects should be picked up (e.g. "Which object do you want?"). User A then should use the robot's tablet to enter the name of the requested object. After that, Hobbit should navigate to the User B. User B then should be asked to handover the desired object. If it is answerd positively, the object should be placed on Hobbit's tray and the robot navigates back to its previous location telling User A "Here you are" and placing the object on the table. If the object has not been handed over, an appropriate message should be displayed on the tablet

(e.g. "I'm sorry, your partner couldn't handover the object") after navigating back.

Afterwards User A should be asked, if Hobbit should bring another item. The whole procedure should be performed as long as User A does not request any other object. After the final decline Hobbit should show a happy emotion. For a better understanding Figure E.2 provides the flowchart of this use case. The locations of User A and B can be consired as the following poses:

- User A:
  - `position:{x:1.0,y:2.0,z:0.0}`
  - `orientation:{x:0.0,y:0.0,z:0.0,w:1.0}`
- User B:
  - `position:{x:14.0,y:-5.0,z:0.0}`
  - `orientation:{x:0.0,y:0.0,z:0.0,w:0.6}`

Please implement a solution using the provided code editor, which lets Hobbit show the desired behaviour, with respect to the following conditions:

- Do exclusively use the provided editor for implementing your solution
- Start working by clicking the "Start" button of the interface
- Do not close the graphical editor during your work
- Click the "Stop" button when you finished implementation
- Click "Submit" to submit your solution

Second use case: Bringing objects from another person - Tool supported

The use case which should be implemented can be described as follows: Hobbit should be repetitively asking User A if it should bring an object from User B, which is located at another place. First the user should be asked which objects should be picked up (e.g. "Which object do you want?"). User A then should use the robot's tablet to enter the name of the requested object. After that, Hobbit should navigate to the User B. User B then should be asked to handover the desired object. If it is answerd positively, the object should be placed on Hobbit's tray and the robot navigates back to its previous location telling User A "Here you are" and placing the object on the table. If the object has not

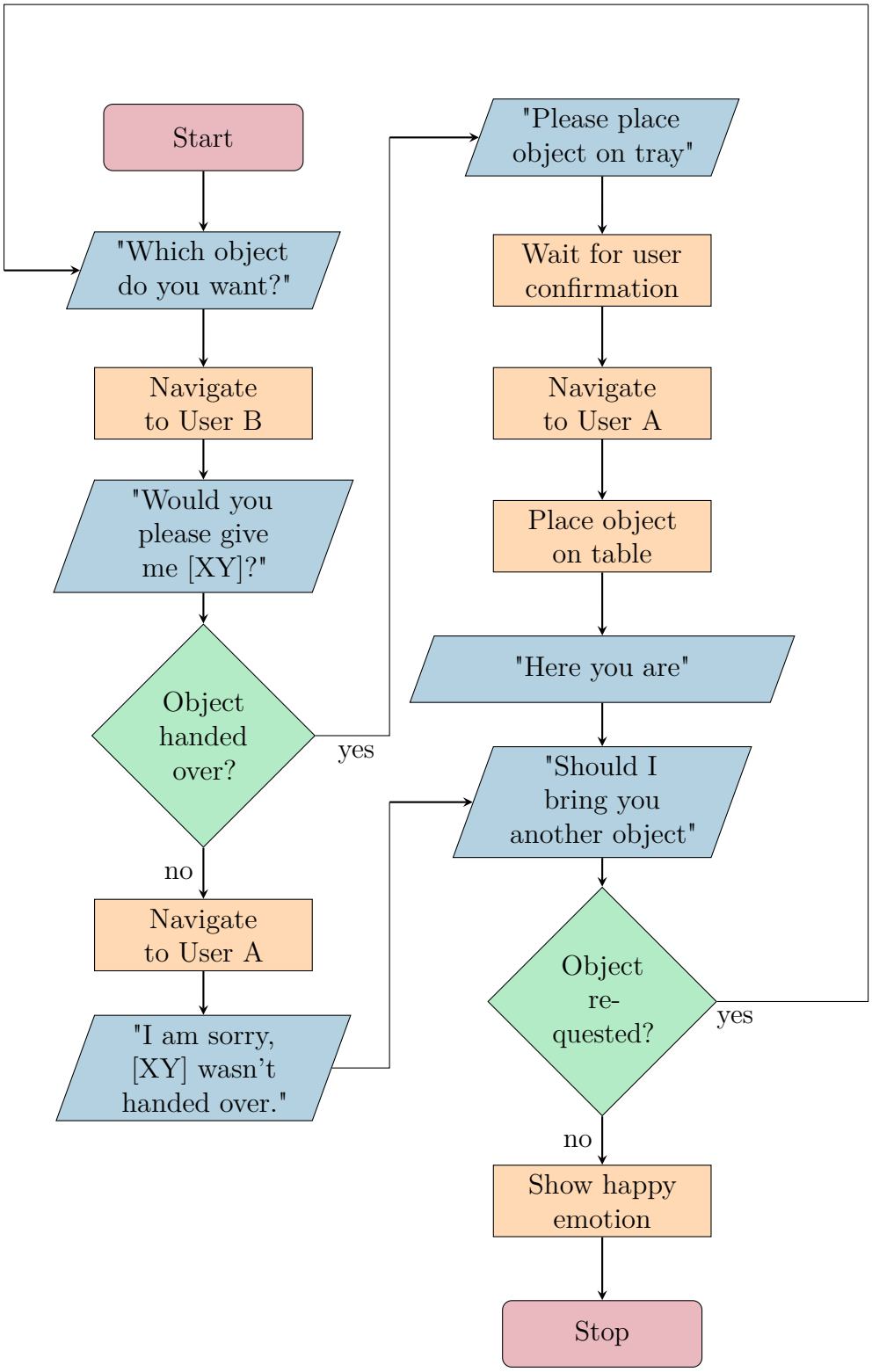


Figure E.2: Flowchart of second use case

been handed over, an appropriate message should be displayed on the tablet (e.g. "I'm sorry, your partner couldn't handover the object") after navigating back.

Afterwards User A should be asked, if Hobbit should bring another item. The whole procedure should be performed as long as User A does not request any other object. After the final decline Hobbit should show a happy emotion. For a better understanding Figure E.2 provides the flowchart of this use case. The locations of User A and B can be consired as the following poses:

- User A:
  - `position:{x:1.0,y:2.0,z:0.0}`
  - `orientation:{x:0.0,y:0.0,z:0.0,w:1.0}`
- User B:
  - `position:{x:14.0,y:-5.0,z:0.0}`
  - `orientation:{x:0.0,y:0.0,z:0.0,w:0.6}`

Please implement a solution using the Blockly editor, which lets Hobbit show the desired behaviour, with respect to the following conditions:

- Start working by clicking the "Start" button of the interface
- Do not close the graphical editor during your work
- Click the "Stop" button when you finished implementation
- Click "Submit" to submit your solution
- Each block provides a help page - it is accessible via *right click* → *Help*

Explanations and Examples

Introduction

roscpp is a C++ implementation of ROS. It provides a client library that enables C++ programmers to quickly interface with ROS Topics, Services, and Parameters. roscpp is the most widely used ROS client library and is designed to be the high-performance library for ROS.

Writing a Publisher Node

```
#include "ros/ros.h"
#include "std_msgs/String.h"

#include <sstream>

/**
 * This tutorial demonstrates simple sending of messages over the ROS system.
 */
int main(int argc, char **argv)
{
  /**
   * The ros::init() function needs to see argc and argv so that it can perform
   * any ROS arguments and name remapping that were provided at the command line.
   * For programmatic remappings you can use a different version of init() which takes
   * remappings directly, but for most command-line programs, passing argc and argv is
   * the easiest way to do it.  The third argument to init() is the name of the node.
   *
   * You must call one of the versions of ros::init() before using any other
   * part of the ROS system.
   */
  ros::init(argc, argv, "talker");

  /**
   * NodeHandle is the main access point to communications with the ROS system.
   * The first NodeHandle constructed will fully initialize this node, and the last
   * NodeHandle destructed will close down the node.
   */
  ros::NodeHandle n;

  /**
   * The advertise() function is how you tell ROS that you want to
   * publish on a given topic name. This invokes a call to the ROS
   * master node, which keeps a registry of who is publishing and who
   * is subscribing. After this advertise() call is made, the master
   * node will notify anyone who is trying to subscribe to this topic name,
   * and they will in turn negotiate a peer-to-peer connection with this
   * node.  advertise() returns a Publisher object which allows you to
   * publish messages on that topic through a call to publish().  Once
   * all copies of the returned Publisher object are destroyed, the topic
```

```
   * will be automatically unadvertised.
   *
   * The second parameter to advertise() is the size of the message queue
   * used for publishing messages.  If messages are published more quickly
   * than we can send them, the number here specifies how many messages to
   * buffer up before throwing some away.
   */
  ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);

  ros::Rate loop_rate(10);

  /**
   * A count of how many messages we have sent. This is used to create
   * a unique string for each message.
   */
  int count = 0;
  while (ros::ok())
  {
    /**
     * This is a message object. You stuff it with data, and then publish it.
     */
    std_msgs::String msg;

    std::stringstream ss;
    ss << "hello world " << count;
    msg.data = ss.str();

    ROS_INFO("%s", msg.data.c_str());

    /**
     * The publish() function is how you send messages. The parameter
     * is the message object. The type of this object must agree with the type
     * given as a template parameter to the advertise<>() call, as was done
     * in the constructor above.
     */
    chatter_pub.publish(msg);

    ros::spinOnce();

    loop_rate.sleep();
    ++count;
  }


  return 0;
}
```

The Code Explained

Now, let's break the code down.

```
#include "ros/ros.h"
```

ros/ros.h is a convenience include that includes all the headers necessary to use the most common public pieces of the ROS system.

```
#include "std_msgs/String.h"
```

This includes the std_msgs/String message, which resides in the std_msgs package. This is a header generated automatically from the String.msg file in that package. For more information on message definitions, see the msg page.

```
    ros::init(argc, argv, "talker");
```

Initialize ROS. This allows ROS to do name remapping through the command line -- not important for now. This is also where we specify the name of our node. Node names must be unique in a running system.

The name used here must be a base name, ie. it cannot have a / in it.

```
    ros::NodeHandle n;
```

Create a handle to this process' node. The first NodeHandle created will actually do the initialization of the node, and the last one destructed will cleanup any resources the node was using.

```
    ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
```

Tell the master that we are going to be publishing a message of type std_msgs/String on the topic chatter. This lets the master tell any nodes listening on chatter that we are going to publish data on that topic. The second argument is the size of our publishing queue. In this case if we are publishing too quickly it will buffer up a maximum of 1000 messages before beginning to throw away old ones.

NodeHandle::advertise() returns a ros::Publisher object, which serves two purposes: 1) it contains a publish() method that lets you publish messages onto the topic it was created with, and 2) when it goes out of scope, it will automatically unadvertise.

```
    ros::Rate loop_rate(10);
```

A ros::Rate object allows you to specify a frequency that you would like to loop at. It will keep track of how long it has been since the last call to Rate::sleep(), and sleep for the correct amount of time.

In this case we tell it we want to run at 10Hz.

```
int count = 0;
while (ros::ok())
{
```

By default roscpp will install a SIGINT handler which provides Ctrl-C handling which will cause ros::ok() to return false if that happens.

ros::ok() will return false if:

- a SIGINT is received (Ctrl-C)
- we have been kicked off the network by another node with the same name
- ros::shutdown() has been called by another part of the application.
- all ros::NodeHandles have been destroyed

Once ros::ok() returns false, all ROS calls will fail.

```
std_msgs::String msg;

std::stringstream ss;
ss << "hello world " << count;
msg.data = ss.str();
```

We broadcast a message on ROS using a message-adapted class, generally generated from a msg file. More complicated datatypes are possible, but for now we're going to use the standard String message, which has one member: "data".

```
chatter_pub.publish(msg);
```

Now we actually broadcast the message to anyone who is connected.

```
ROS_INFO("%s", msg.data.c_str());
```

ROS_INFO and friends are our replacement for printf/cout. See the rosconsole documentation for more information.

```
ros::spinOnce();
```

Calling ros::spinOnce() here is not necessary for this simple program, because we are not receiving any callbacks. However, if you were to add a subscription into this application, and did not have ros::spinOnce() here, your callbacks would never get called. So, add it for good measure.

```
    loop_rate.sleep();
```

Now we use the ros::Rate object to sleep for the time remaining to let us hit our 10Hz publish rate.

Writing a Simple Service

This tutorial covers how to write a service node in C++.

```
#include "ros/ros.h"
#include "beginner_tutorials/AddTwoInts.h"

bool add(beginner_tutorials::AddTwoInts::Request  &req,
         beginner_tutorials::AddTwoInts::Response &res)
{
  res.sum = req.a + req.b;
  ROS_INFO("request: x=%ld, y=%ld", (long int)req.a, (long int)req.b);
  ROS_INFO("sending back response: [%ld]", (long int)res.sum);
  return true;
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "add_two_ints_server");

  ros::NodeHandle n;

  ros::ServiceServer service = n.advertiseService("add_two_ints", add);
  ROS_INFO("Ready to add two ints.");
  ros::spin();

  return 0;
}
```

Now, let's break the code down.

```
#include "ros/ros.h"
#include "beginner_tutorials/AddTwoInts.h"
```

beginner_tutorials/AddTwoInts.h is the header file generated from the srv file that we created earlier.

```
bool add(beginner_tutorials::AddTwoInts::Request  &req,
         beginner_tutorials::AddTwoInts::Response &res)
```

This function provides the service for adding two ints, it takes in the request and response type defined in the srv file and returns a boolean.

```
{
  res.sum = req.a + req.b;
  ROS_INFO("request: x=%ld, y=%ld", (long int)req.a, (long int)req.b);
  ROS_INFO("sending back response: [%ld]", (long int)res.sum);
  return true;
}
```

Here the two ints are added and stored in the response. Then some information about the request and response are logged. Finally the service returns true when it is complete.

```
  ros::ServiceServer service = n.advertiseService("add_two_ints", add);
```

Here the service is created and advertised over ROS.

Writing a Simple Service Client

This tutorial covers how to write a service client node in C++.

```
#include "ros/ros.h"
#include "beginner_tutorials/AddTwoInts.h"
#include <cstdlib>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "add_two_ints_client");
  if (argc != 3)
  {
    ROS_INFO("usage: add_two_ints_client X Y");
    return 1;
  }

  ros::NodeHandle n;
  ros::ServiceClient client = n.serviceClient<beginner_tutorials::AddTwoInts>("add_two_int
  beginner_tutorials::AddTwoInts srv;
  srv.request.a = atoll(argv[1]);
  srv.request.b = atoll(argv[2]);
```

```
  if (client.call(srv))
  {
    ROS_INFO("Sum: %ld", (long int)srv.response.sum);
  }
  else
  {
    ROS_ERROR("Failed to call service add_two_ints");
    return 1;
  }

  return 0;
}
```

The Code Explained

Now, let's break the code down.

```
ros::ServiceClient client = n.serviceClient<beginner_tutorials::AddTwoInts>("add_two_int
```

This creates a client for the add_two_ints service. The ros::ServiceClient object is used to call the service later on.

```
beginner_tutorials::AddTwoInts srv;
  srv.request.a = atoll(argv[1]);
  srv.request.b = atoll(argv[2]);
```

Here we instantiate an autogenerated service class, and assign values into its request member. A service class contains two members, request and response. It also contains two class definitions, Request and Response.

```
if (client.call(srv))
```

This actually calls the service. Since service calls are blocking, it will return once the call is done. If the service call succeeded, call() will return true and the value in srv.response will be valid. If the call did not succeed, call() will return false and the value in srv.response will be invalid.

Writing a Simple Action Client

This tutorial covers using the simple_action_client library to create a Fibonacci action client. This example program creates an action client and sends a goal to the action server.

```
#include <ros/ros.h>
#include <actionlib/client/simple_action_client.h>
#include <actionlib/client/terminal_state.h>
#include <actionlib_tutorials/FibonacciAction.h>

int main (int argc, char **argv)
{
  ros::init(argc, argv, "test_fibonacci");

  // create the action client
  // true causes the client to spin its own thread
  actionlib::SimpleActionClient<actionlib_tutorials::FibonacciAction> ac("fibonacci", true

  ROS_INFO("Waiting for action server to start.");
  // wait for the action server to start
  ac.waitForServer(); //will wait for infinite time

  ROS_INFO("Action server started, sending goal.");
  // send a goal to the action
  actionlib_tutorials::FibonacciGoal goal;
  goal.order = 20;
  ac.sendGoal(goal);

  //wait for the action to return
  bool finished_before_timeout = ac.waitForResult(ros::Duration(30.0));

  if (finished_before_timeout)
  {
    actionlib::SimpleClientGoalState state = ac.getState();
    ROS_INFO("Action finished: %s",state.toString().c_str());
  }
  else
    ROS_INFO("Action did not finish before the time out.");

  //exit
  return 0;
}
```

The Code Explained

Now, let's break down the code piece by piece.

```
#include <ros/ros.h>
#include <actionlib/client/simple_action_client.h>
#include <actionlib/client/terminal_state.h>
```

- actionlib/client/simple_action_client.h is the action library used from implementing simple action clients.

- actionlib/client/terminal_state.h defines the possible goal states.

```
#include <actionlib_tutorials/FibonacciAction.h>
```

This includes action message generated from the Fibonacci.action file shown above. This is a header generated automatically from the FibonacciAction.msg file. For more information on message definitions, see the msg page.

```
int main (int argc, char **argv)
{
  ros::init(argc, argv, "test_fibonacci");

  // create the action client
  // true causes the client to spin its own thread
  actionlib::SimpleActionClient<actionlib_tutorials::FibonacciAction> ac("fibonacci", true
```

The action client is templated on the action definition, specifying what message types to communicate to the action server with. The action client constructor also takes two arguments, the server name to connect to and a boolean option to automatically spin a thread. If you prefer not to use threads (and you want actionlib to do the 'thread magic' behind the scenes), this is a good option for you. Here the action client is constructed with the server name and the auto spin option set to true.

```
  ROS_INFO("Waiting for action server to start.");
  // wait for the action server to start
  ac.waitForServer(); //will wait for infinite time
```

Since the action server may not be up and running, the action client will wait for the action server to start before continuing.

```
  ROS_INFO("Action server started, sending goal.");
  // send a goal to the action
  actionlib_tutorials::FibonacciGoal goal;
  goal.order = 20;
  ac.sendGoal(goal);
```

Here a goal message is created, the goal value is set and sent to the action server.

```
  //wait for the action to return
  bool finished_before_timeout = ac.waitForResult(ros::Duration(30.0));
```

The action client now waits for the goal to finish before continuing. The timeout on the wait is set to 30 seconds, this means after 30 seconds the function will return with false if the goal has not finished.

```
  if (finished_before_timeout)
  {
    actionlib::SimpleClientGoalState state = ac.getState();
    ROS_INFO("Action finished: %s",state.toString().c_str());
  }
  else
    ROS_INFO("Action did not finish before the time out.");

  //exit
  return 0;
}
```

If the goal finished before the time out the goal status is reported, else the user is notified that the goal did not finish in the allotted time.

Introduction

rospy is a pure Python client library for ROS. The rospy client API enables Python programmers to quickly interface with ROS Topics, Services, and Parameters.

Writing a Publisher Node

This demo will walk you through creating a simple ROS node (*"talker"*), which will broadcast a message on topic *"chatter"*.

```
#!/usr/bin/env python
# license removed for brevity
import rospy
from std_msgs.msg import String

def talker():
    pub = rospy.Publisher('chatter', String, queue_size=10)
    rospy.init_node('talker', anonymous=True)
    rate = rospy.Rate(10) # 10hz
    while not rospy.is_shutdown():
        hello_str = "hello world %s" % rospy.get_time()
        rospy.loginfo(hello_str)
        pub.publish(hello_str)
        rate.sleep()

if __name__ == '__main__':
    try:
        talker()
    except rospy.ROSInterruptException:
        pass
```

The Code Explained

Now, let's break the code down.

```
#!/usr/bin/env python
```

Every Python ROS Node will have this declaration at the top. The first line makes sure your script is executed as a Python script.

```
import rospy
from std_msgs.msg import String
```

You need to import rospy if you are writing a ROS Node. The std_msgs.msg import is so that we can reuse the std_msgs/String message type (a simple string container) for publishing.

```
pub = rospy.Publisher('chatter', String, queue_size=10)
rospy.init_node('talker', anonymous=True)
```

This section of code defines the talker's interface to the rest of ROS. `pub = rospy.Publisher("chatter", String, queue_size=10)` declares that your node is publishing to the chatter topic using the message type String. String here is actually the class `std_msgs.msg.String` . The queue_size argument is New in ROS hydro and limits the amount of queued messages if any subscriber is not receiving them fast enough. In older ROS distributions just omit the argument.

The next line, `rospy.init_node(NAME, ...)` , is very important as it tells rospy the name of your node -- until rospy has this information, it cannot start communicating with the ROS Master. In this case, your node will take on the name talker. NOTE: the name must be a base name, i.e. it cannot contain any slashes "/".

`anonymous = True` ensures that your node has a unique name by adding random numbers to the end of NAME. Refer to Initialization and Shutdown - Initializing your ROS Node in the rospy documentation for more information about node initialization options.

```
rate = rospy.Rate(10) # 10hz
```

This line creates a Rate object rate. With the help of its method sleep(), it offers a convenient way for looping at the desired rate. With its argument of 10, we should expect to go through the loop 10 times per second (as long as our processing time does not exceed 1/10th of a second!)

```
while not rospy.is_shutdown():
    hello_str = "hello world %s" % rospy.get_time()
    rospy.loginfo(hello_str)
    pub.publish(hello_str)
    rate.sleep()
```

This loop is a fairly standard rospy construct: checking the rospy.is_shutdown() flag and then doing work. You have to check is_shutdown() to check if your program should exit (e.g. if there is a Ctrl-C or otherwise). In this case, the "work" is a call to pub.publish(hello_str) that publishes a string to our chatter topic. The loop calls rate.sleep(), which sleeps just long enough to maintain the desired rate through the loop.

(You may also run across rospy.sleep() which is similar to time.sleep() except that it works with simulated time as well (see Clock).)

This loop also calls rospy.loginfo(str), which performs triple-duty: the messages get printed to screen, it gets written to the Node's log file, and it gets written to rosout. rosout is a handy for debugging: you can pull up messages using rqt_console instead of having to find the console window with your Node's output.

std_msgs.msg.String is a very simple message type, so you may be wondering what it looks like to publish more complicated types. The general rule of thumb is that constructor args are in the same order as in the .msg file. You can also pass in no arguments and initialize the fields directly, e.g.

```
msg = String()
msg.data = str
```

or you can initialize some of the fields and leave the rest with default values:

```
String(data=str)
```

You may be wondering about the last little bit:

```
try:
    talker()
except rospy.ROSInterruptException:
    pass
```

In addition to the standard Python `__main__` check, this catches a rospy.ROSInterruptException exception, which can be thrown by rospy.sleep() and rospy.Rate.sleep() methods when Ctrl-C is pressed or your Node is otherwise shutdown. The reason this exception is raised is so that you don't accidentally continue executing code after the sleep().

Writing a Service Node

Here we'll create the service ("add_two_ints_server") node which will receive two ints and return the sum.

```
#!/usr/bin/env python

from beginner_tutorials.srv import *
import rospy
```

```
def handle_add_two_ints(req):
    print "Returning [%s + %s = %s]"%(req.a, req.b, (req.a + req.b))
    return AddTwoIntsResponse(req.a + req.b)

def add_two_ints_server():
    rospy.init_node('add_two_ints_server')
    s = rospy.Service('add_two_ints', AddTwoInts, handle_add_two_ints)

    print "Ready to add two ints."
    rospy.spin()

if __name__ == "__main__":
    add_two_ints_server()
```

The Code Explained

Now, let's break the code down.

There's very little to writing a service using rospy. We declare our node using init_node() and then declare our service:

```
    s = rospy.Service('add_two_ints', AddTwoInts, handle_add_two_ints)
```

This declares a new service named add_two_ints with the AddTwoInts service type. All requests are passed to handle_add_two_ints function. handle_add_two_ints is called with instances of AddTwoIntsRequest and returns instances of AddTwoIntsResponse.

Just like with the subscriber example, rospy.spin() keeps your code from exiting until the service is shutdown.

Writing a Service Client Node

The following code calls the above created service.

```
#!/usr/bin/env python

import sys
import rospy
from beginner_tutorials.srv import *

def add_two_ints_client(x, y):
    rospy.wait_for_service('add_two_ints')
    try:
        add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
        resp1 = add_two_ints(x, y)
        return resp1.sum
```

```
    except rospy.ServiceException, e:
        print "Service call failed: %s"%e

def usage():
    return "%s [x y]"%sys.argv[0]

if __name__ == "__main__":
    if len(sys.argv) == 3:
        x = int(sys.argv[1])
        y = int(sys.argv[2])
    else:
        print usage()
        sys.exit(1)

    print "Requesting %s+%s"%(x, y)
    print "%s + %s = %s"%(x, y, add_two_ints_client(x, y))
```

The Code Explained

Now, let's break the code down.

The client code for calling services is also simple. For clients you don't have to call init_node(). We first call:

```
rospy.wait_for_service('add_two_ints')
```

This is a convenience method that blocks until the service named add_two_ints is available. Next we create a handle for calling the service:

```
add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
```

We can use this handle just like a normal function and call it:

```
resp1 = add_two_ints(x, y)
return resp1.sum
```

Because we've declared the type of the service to be AddTwoInts, it does the work of generating the AddTwoIntsRequest object for you (you're free to pass in your own instead). The return value is an AddTwoIntsResponse object. If the call fails, a rospy.ServiceException may be thrown, so you should setup the appropriate try/except block.

Writing a Simple Action Client

This tutorial covers using the action_client library to create a Fibonacci simple action client in Python.

```
#! /usr/bin/env python

import rospy
from __future__ import print_function

# Brings in the SimpleActionClient
import actionlib

# Brings in the messages used by the fibonacci action, including the
# goal message and the result message.
import actionlib_tutorials.msg

def fibonacci_client():
    # Creates the SimpleActionClient, passing the type of the action
    # (FibonacciAction) to the constructor.
    client = actionlib.SimpleActionClient('fibonacci', actionlib_tutorials.msg.FibonacciAc

    # Waits until the action server has started up and started
    # listening for goals.
    client.wait_for_server()

    # Creates a goal to send to the action server.
    goal = actionlib_tutorials.msg.FibonacciGoal(order=20)

    # Sends the goal to the action server.
    client.send_goal(goal)

    # Waits for the server to finish performing the action.
    client.wait_for_result()

    # Prints out the result of executing the action
    return client.get_result()  # A FibonacciResult

if __name__ == '__main__':
    try:
        # Initializes a rospy node so that the SimpleActionClient can
        # publish and subscribe over ROS.
        rospy.init_node('fibonacci_client_py')
        result = fibonacci_client()
        print("Result:", ', '.join([str(n) for n in result.sequence]))
    except rospy.ROSInterruptException:
        print("program interrupted before completion", file=sys.stderr)
```

The Code Explained

```
import actionlib_tutorials.msg
```

The action specification generates several messages for sending goals, receiving feedback, etc... This line imports the generated messages.

```
client = actionlib.SimpleActionClient('fibonacci', actionlib_tutorials.msg.FibonacciAc
```

The action client and server communicate over a set of topics, described in the actionlib protocol. The action name describes the namespace containing these topics, and the action specification message describes what messages should be passed along these topics.

```
client.wait_for_server()
```

Sending goals before the action server comes up would be useless. This line waits until we are connected to the action server.

```
# Creates a goal to send to the action server.
goal = actionlib_tutorials.msg.FibonacciGoal(order=20)

# Sends the goal to the action server.
client.send_goal(goal)
```

Creates a goal and sends it to the action server.

```
# Waits for the server to finish performing the action.
client.wait_for_result()

# Prints out the result of executing the action
return client.get_result()  # A FibonacciResult
```

The action server will process the goal and eventually terminate. We want the result from the termination, but we wait until the server has finished with the goal.

ROS Reference for HOBBIT

Topics

This sections lists some topics which can be used to control HOBBIT.

/head/move

Message Type: `std_msgs/String`

This topic is used to move HOBBIT's head.

Message	Description
"center_center"	look straight
"up_center"	look up
"down_center"	look down
"center_right"	look right
"center_left"	look left
"up_right"	look to upper right corner
"up_left"	look to upper left corner
"down_right"	look to lower right corner
"down_left"	look to lower left corner
"littledown_center"	look little down
"to_grasp"	look to grasp
"to_turntable"	look to turntable
"search_table"	look to table

/head/emo

Message Type: `std_msgs/String`

This topic is used to control HOBBIT's eyes/emotion

Message	Description

Message	Description
"HAPPY"	look happy
"VHAPPY"	look very happy
"LTIRED"	look little tired
"VTIRED"	look very tired
"CONCERNED"	look concerned
"SAD"	look sad
"WONDERING"	wonder
"NEUTRAL"	look neutral
"SLEEPING"	sleep

/cmd_vel

Message type: `geometry_msgs/Twist` This topic is used to move HOBBIT a certain distance in linear direction, i.e. it only uses the x coordinate of the linear part of the geometry_msgs/Twist Message. For example, the following message moves HOBBIT 2 metres forward:

```
'{linear:  {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 0.0}}'
```

Actions

This sections lists some action server namespaces which can be accessed to control HOBBIT.

hobbit_arm

Action goal message type: `hobbit_msgs/ArmServerGoal`

Action message type: `hobbit_msgs/ArmServerAction`

HOBBIT's arm can be controlled in three different ways.

Move arm along trajectory

To move the arm along a trajectory the joint values has to be passed to the goal, e.g.

```
command:
  data: 'MoveAlongTrajectory'
velocity: 0.0
joints: [
  joint_values: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
  joint_values: [0.1, 0.1, 0.0, 0.0, 0.0, 0.0],
  joint_values: [0.2, 0.2, 0.0, 0.0, 0.0, 0.0],
  joint_values: [0.3, 0.3, 0.0, 0.0, 0.0, 0.0],
  joint_values: [0.4, 0.4, 0.0, 0.0, 0.0, 0.0],
  joint_values: [0.5, 0.4, 0.0, 0.0, 0.0, 0.0],
  joint_values: [0.6, 0.4, 0.0, 0.0, 0.0, 0.0],
  joint_values: [0.7, 0.4, 0.0, 0.0, 0.0, 0.0],
  joint_values: [0.8, 0.4, 0.0, 0.0, 0.0, 0.0],
  joint_values: [0.9, 0.4, 0.0, 0.0, 0.0, 0.0],
  joint_values: [1.0, 0.4, 0.0, 0.0, 0.0, 0.0]
]
```

Move to joint values

To move the arm to a specific position it is necessary to provide it via the values of the joints and set the command.data property accordingly, e.g.

```
command:
  data: 'MoveToJointValues'
velocity: 0.0
joints: [
  joint_values: [1.0, 0.4, 0.0, 0.0, 0.0, 0.0]
]
```

Move to predefined position

It is also possible to move HOBBIT's arm to a couple of predefined position. The following message, for example, lets the arm move to its home position:

```
command:
  data: 'MoveToHome'
velocity: 0.0
joints: []
```

For all predefined position please refer to the following table.

Command	Description
"MoveToCandle"	Move to candle position
"MoveToHome"	Move to home position

Command	Description
"MoveToPreGraspFloor"	Prepare to grasp from floor
"MoveToPreGraspTable"	Move to table position
"MoveToTray"	Move arm to tray
"MoveToLearning"	Grab turntable
"StoreTurntable"	Store turntable
"TurnTurntableCW"	Turn turntable clockwise
"TurnTurntableCCW"	Turn turntable counterclockwise
"OpenGripper"	Open gripper
"CloseGripper"	Close gripper

move_base

Action goal message type: `geometry_msgs/PoseStamped`

Action message type: `move_base_msgs/MoveBase`

This namespace can be used to navigate HOBBIT to a specific point determinated by its geometric pose, for example

```
header:
  seq: 0
  stamp: now
  frame_id: "map"
pose:
  position:
    x: 1.0
    y: 2.0
    z: 0.0
  orientation:
    x: 0.0
    y: 0.0
    z: 0.0
    w: 1.0
```

> Note: Navigation is also possible by publishing a `geometry_msgs/PoseStamped` message to topic `/move_base_simple/goal` .

Services

This sections lists some services which can be used to control HOBBIT.

/MMUI

Service class: `hobbit_msgs/Request`

This service can be used to interact with the user via HOBBIT's tablet. The information is passed within the `params` Parameter array (type `hobbit_msgs/Parameter`) along some meta information. The following example shows how to prompt input from the user.

```
header:
  seq: 0
  stamp: now
  frame_id: ""
sessionID: "0"
requestText: ""
params: [
  {name: "type", value: "D_NAME"},
  {name: "text", value: "How are you?"},
  {name: "Timeout", value: "30"}
]
```

Basically, the desired action is defined by the "type" parameter of the `params` array. Please refer to the following table for some common functionality of the tablet UI.

params.type	Description
D_NAME	get user input from tablet keyboard*
D_PLAIN	show info
D_OK	show info and wait for confirmation
D_YES_NO	ask a yes-no-question*
F_CALLSOS	start SOS call
F_LOUDER	Set volume 10% higher
F_QUIETER	Set volume 10% lower

* Please consider the following structure for the response:

```
{
  params:[
    {
      ...
      value: <UserInput>
      ...
    },
    {
      ...
    }
    ...
  ],
  ...
}
```

For `params.type='D_YES_NO'` \<*UserInput*> is a string containing `'yes'` or `'no'` depending on which button the user has clicked. For `params.type='D_YES_NO'` \<*UserInput*> is a string containing the string the user enterd via the tablet interface.

For more detailed examples using `rospy` please refer to the this example.

Message Definitions

std_msgs/String

```
string data
```

geometry_msgs/Twist

```
geometry_msgs/Vector3 linear
geometry_msgs/Vector3 angular
```

geometry_msgs/Vector3

```
float64 x
float64 y
float64 z
```

geometry_msgs/PoseStamped

```
std_msgs/Header header
geometry_msgs/Pose pose
```

std_msgs/Header

```
uint32 seq
time stamp
string frame_id
```

geometry_msgs/Pose

```
geometry_msgs/Point position
geometry_msgs/Quaternion orientation
```

geometry_msgs/Point

```
float64 x
float64 y
float64 z
```

geometry_msgs/Quaternion

```
float64 x
float64 y
float64 z
float64 w
```

hobbit_msgs/ArmServerGoal

```
std_msgs/String command
float64 velocity
hobbit_msgs/arm_joints joints
```

hobbit_msgs/arm_joints

```
float32[6] joint_values
```

hobbit_msgs/Parameter

```
string name
string value
```

Service Classes

hobbit_msgs/Request

```
#Request Service. used for requesting input from user/mmui by modules or vice-versa
Header header
string sessionID
string requestText # please enter a name for this place, etc.

Parameter[] params  # Requestparams
---
Parameter[] params  # Responseparams
```

Bibliography

[1] R. T. Fielding, „Architectural Styles and the Design of Network-based Software Architectures," AAI9980887, PhD thesis, 2000, ISBN: 0-599-87118-0.

[2] I. E. Sutherland, „Sketchpad: A Man-machine Graphical Communication System," in *Proceedings of the May 21-23, 1963, Spring Joint Computer Conference*, ser. AFIPS '63 (Spring), Detroit, Michigan: ACM, 1963, pp. 329–346. [Online]. Available: `http://doi.acm.org/10.1145/1461551.1461591`.

[3] M. Boshernitsan and M. S. Downes, „Visual Programming Languages: a Survey," EECS Department, University of California, Berkeley, Tech. Rep. UCB/CSD-04-1368, 2004. [Online]. Available: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2004/6201.html`.

[4] D. C. Smith, „Pygmalion: A Creative Programming Environment.," AAI7525608, PhD thesis, Stanford, CA, USA, 1975.

[5] S. C. Pokress and J. J. D. Veiga, „MIT App Inventor: Enabling Personal Mobile Computing.," *PRoMoTo 2013 Proceedings*, 2013. [Online]. Available: `http://arxiv.org/abs/1310.2830`.

[6] M. Resnick, J. Maloney, A. Monroy-Hernandez, N. Rusk, E. Eastmond, and K. Brennan, „Scratch: Programming for all," *Communications of the ACM*, vol. 52, no. 11, pp. 60–67, 2009. [Online]. Available: `http://cacm.acm.org/magazines/2009/11/48421-scratch-programming-for-all/pdf`.

[7] „Information technology – Vocabulary," International Organization for Standardization, Geneva, CH, Standard, May 2015.

[8] R. David, „Grafcet: a powerful tool for specification of logic controllers," *IEEE Transactions on Control Systems Technology*, vol. 3, no. 3, pp. 253–268, 1995, ISSN: 1063-6536.

[9] S. Alexandrova, Z. Tatlock, and M. Cakmak, „RoboFlow: A flow-based visual programming language for mobile manipulation tasks," in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, 2015, pp. 5537–5544.

[10] T. B. Sousa, „Dataflow Programming Concept , Languages and Applications,“ 2012.

[11] J. Travis and J. Kring, *LabVIEW for Everyone: Graphical Programming Made Easy and Fun, 3rd Edition*. Prentice Hall Professional, 2007, ISBN: 0131856723.

[12] A. J. Hirst, J. Johnson, M. Petre, B. A. Price, and M. Richards, „What is the best programming environment/language for teaching robotics using Lego Mindstorms?“ *Artificial Life and Robotics*, vol. 7, no. 3, pp. 124–131, 2003, ISSN: 1614-7456. [Online]. Available: `https://doi.org/10.1007/BF02481160`.

[13] S. Enderle, „Grape – Graphical Robot Programming for Beginners,“ in *Research and Education in Robotics — EUROBOT 2008*, A. Gottscheber, S. Enderle, and D. Obdrzalek, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 180–192, ISBN: 978-3-642-03558-6.

[14] B. Jost, M. Ketterl, R. Budde, and T. Leimbach, „Graphical Programming Environments for Educational Robots: Open Roberta - Yet Another One?“ In *2014 IEEE International Symposium on Multimedia*, 2014, pp. 381–386.

[15] M. Ketterl, B. Jost, T. Leimbach, and R. Budde, „Tema 2: Open Roberta - A Web Based Approach to Visually Program Real Educational Robots,“ *Tidsskriftet LÃ¦ring og Medier (LOM)*, vol. 8, no. 14, 2015. [Online]. Available: `https://tidsskrift.dk/lom/article/view/22183`.

[16] D. Gouaillier, V. Hugel, P. Blazevic, C. Kilner, J. Monceaux, P. Lafourcade, B. Marnier, J. Serre, and B. Maisonnier, „Mechatronic design of NAO humanoid,“ in *2009 IEEE International Conference on Robotics and Automation*, 2009, pp. 769–774.

[17] E. Pot, J. Monceaux, R. Gelin, and B. Maisonnier, „Choregraphe: a graphical tool for humanoid robot programming,“ in *RO-MAN 2009 - The 18th IEEE International Symposium on Robot and Human Interactive Communication*, 2009, pp. 46–51.

[18] S. ERLE ROBOTICS, *Robot_Blockly documentation*. [Online]. Available: `http://docs.erlerobotics.com/robot_operating_system/ros/blockly/intro` (visited on 11/26/2018).

[19] I. Muhendislik, *Introduction to evablockly_ros*. [Online]. Available: `http://wiki.ros.org/evablockly_ros/Tutorials/indigo/Introduction` (visited on 11/30/2018).

[20] N. Gonzalez, A. H. Cordero, and V. M. Vilches, *robot_blockly ROS package documentation*, 2018. [Online]. Available: `http://wiki.ros.org/robot_blockly` (visited on 12/03/2018).

[21] T. L. Group, *EV3 Programmer App*. [Online]. Available: `https://www.lego.com/en-us/mindstorms/apps/ev3-programmer-app` (visited on 12/03/2018).

[22] Automation and T. W. Control Institute, *HOBBIT The Mutual Care Robot*. [Online]. Available: `https://www.acin.tuwien.ac.at/vision-for-robotics/roboter/hobbit/` (visited on 08/17/2018).

[23] D Fischinger, P. Einramhof, W. Wohlkinger, K. Papoutsakis, P. Mayer, P. Panek, T. Koertner, S Hofmann, A. Argyros, M. Vincze, *et al.*, „Hobbit-The Mutual Care Robot," Jan. 2013.

[24] S. Frennert, M. Bajones, M. Vincze, A. Weiss, D. Wolf, T. Koertner, M. Weninger, and H. Eftring, „Hobbit - Providing Fall Detection and Prevention for the Elderly in the Real World," *Journal of Robotics*, Mar. 2018.

[25] Automation and T. W. Control Institute, *ER4STEM Educational Robotics for STEM*. [Online]. Available: `https://www.acin.tuwien.ac.at/project/er4stem/` (visited on 08/17/2018).

[26] J. Bohren, *smach - ROS Wiki: Package Summary*. [Online]. Available: `http://wiki.ros.org/smach` (visited on 09/10/2018).

[27] *Introduction to Blockly*. [Online]. Available: `https://developers.google.com/blockly/guides/overview` (visited on 09/10/2018).

[28] *API documentation for the JavaScript library used to create webpages with Blockly*. [Online]. Available: `https://developers.google.com/blockly/reference/overview#javascript_library_apis` (visited on 09/13/2018).

[29] *Custom Blocks*. [Online]. Available: `https://developers.google.com/blockly/guides/create-custom-blocks/overview` (visited on 09/13/2018).

[30] *ROS Documentation*. [Online]. Available: `http://wiki.ros.org/` (visited on 09/28/2018).

[31] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, *Hypertext Transfer Protocol – HTTP/1.1*, 1999. [Online]. Available: `https://www.ietf.org/rfc/rfc2616.txt` (visited on 08/17/2018).

[32] S. G. Hart and L. E. Staveland, „Development of NASA-TLX (Task Load Index): Results of Empirical and Theoretical Research,“ in *Human Mental Workload*, ser. Advances in Psychology, P. A. Hancock and N. Meshkati, Eds., vol. 52, North-Holland, 1988, pp. 139 –183. [Online]. Available: `http://www.sciencedirect.com/science/article/pii/S0166411508623869`.

[33] N. Fraser, *Blockly Block Wiki*. [Online]. Available: `https://github.com/google/blockly/wiki` (visited on 11/07/2018).

Erklärung

Hiermit erkläre ich, dass die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt wurde. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Wien, Juli 2019

Alexander Semeliker