TECHNISCHE UNIVERSITÄT WIEN

ACIN AUTOMATION & CONTROL INSTITUTE
INSTITUT FÜR AUTOMATISIERUNGS- & REGELUNGSTECHNIK

Rockly: 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
Georg Humbhandl Gasse 1
2362 Biedermannsdorf
Österreich

Wien, im September 2018

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

Preamble

<Hier bitte Vorwort schreiben.>
Wien, im September 2018

Abstract

This work presents the development of an graphical user interface for programming ROS based robots. ROS (Robot Operating System) 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. Using these frameworks means that the tool can be easily deployed and run on many different platforms. Rockly - the name of the tool and 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. Therefore the presented tool tackles two key points of the field of robotics: teaching robot programming methods and abstracting the complex implementation to a level, where very little technical expertise is required.

Summary

Contents

List of Figures

List of Tables

Listings

1 Introduction

2 Related Work

2.1 Choregraphe

2.2 evablockly_ros - Inovasyon Muhendislik

2.3 robot_blockly - erlerobotics

2.4 Open Roberta Lab

2.5 RobotC

2.6 Ardublock

2.7 Grape â€“ Graphical Robot Programming for Beginners

3 Architecture

This chapter describes the architectural design of Rockly (portmanteau of ROS and Blockly). First the requirements are presented. The purpose and need of the tool are explained. Based on this constraints the options implemtenting the tool are presented as well as an explanation of the decision. Then a overview of the architecture is given, followed by a short description of all supporting frameworks 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 Mutal 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. [1]

In terms of technology HOBBIT (see Figure 3.1) 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

Figure 3.1: HOBBIT - The Mutal Care Robot

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

Python or C++. 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 will be presented in Section 3.4.1 after furher explanation of the ROS architecture.

3.1.2 Purpose of the tool

HOBBIT became very popular since the above mentioned EU project and demos of it's behaviours has been presenting at large number of fairs. Unfortunally only the following show cases are currently implemented on the robot:

- HOBBIT follows the user
- HOBBIT learns object
- HOBBIT starts an emergency call
- HOBBIT picks up an object

All of the demos can be started via the UI 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 is part of the Educational Robotics for STEM (ER4STEM) project, which aims to turn curious young children into young adults passionate about science and technology with hands-on workshops on robotics. The ER4STEM framework will coherently offer 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. [2] Providing an intuitive programming tool would allow the integration of HOBBIT into the project, which would be an extra input evaluation parameter.

At last the framework should be implement 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 an 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 application programming interface (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 a API could consists 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 doesn't re-implement ROS functionality, but extend it and prvovide a simpler use of it. Depending of how generic the API is implemented it is possible that the user can control the robot without any detailed knowledge of the technical setup of it. 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 3.2. 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 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.

Figure 3.2: Exemplary design of API for Python

```python
import HobbitRosModule  # Import of API module

node = HobbitRosModule.node('demo') # Create ROS node
node.gripper('open')    # Open gripper
node.move_arm('floor')  # Move arm to floor pick up
    position
node.gripper('close')   # Close gripper = pick object
node.move_arm('table')  # Move to table position
node.gripper('open')    # Open gripper = drop object
```

Listing 3.1: Example Python code using the API shown in Figure 3.2

3.2.2 SMACH

SMACH is a task-level architecture for rapidly creating complex robot behavior. At its core, SMACH is a ROS-independent Python library to build hierarchical state machines. [3]. Since that, 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 doesn't require a lot of the provided SMACH functionality, this section only covers the needed ones to fulfill the requirements. For a detail description on how to use SMACH refer to [3].

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 broadcast 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 3.3.

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 demo case. Furthermore it requires the user also to exactly know the ROS

```
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

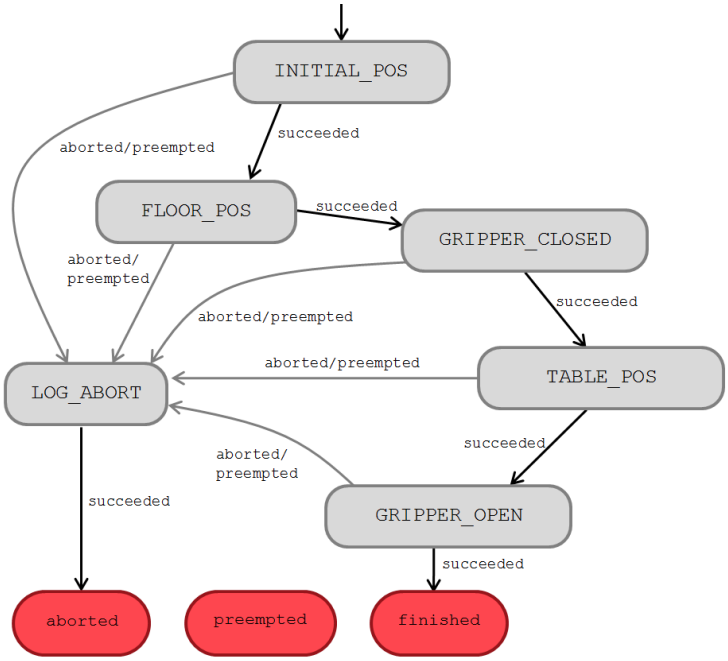


Figure 3.3: State machine generated via Listing 3.2

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 3.2.

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. [4] 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 a editor would not require the user to master any programming language. On the other hand implementing this approach would require knowledge of web applications (i.e. JavaScript, HTML and CSS) additionally.

Figure 3.4 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 dragged to the workspace, where they are connected. The Blockly

Figure 3.4: A short Blockly demo showing it's structure and use

API [5] provides a function to generate 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 3.4 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 described briefly in the following paragraphs. A detailed documentation is given in [6].

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 exampe 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 provided 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 3.2), 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 - in the given case over a hundred - files, is converted into a stand-alone form, that 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 3.4 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 provide the ability to add new functionalites.
- The tool should be maintable, meaning it should have clear interfaces and as less dependencies as possible.

The selection is based on rating each of the approaches in regards of each single requirement mentioned with a score ranging from 0 (lowest) to 5 (highest). The approach with the highest overall score is seen as the best fitting one and will be implemented. Table 3.2 shows the result of the evaluation. In the end the 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 circumstance 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.

	Custom API	SMACH	Blockly
Complexity of demos	3	4	2
User Interface	2	2	5
Robot specific know-how	3	1	5
Technical know-how	1	1	5
Implementation	2	3	3
Scalability	4	3	2
Maintainability	3	3	1
Σ	18	17	23

Table 3.2: Evaluation of possible approaches

3.3 Design

Based on the mentioned framework decision a design for the tool was developed. Figure 3.5 gives an overview of Rockly's final architecture and it's components. While the frontend 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,
- 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 **Backend Communication**

the backend is made up of the following four:

- sending and receiving data from the user interface via the **Frontend 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**).

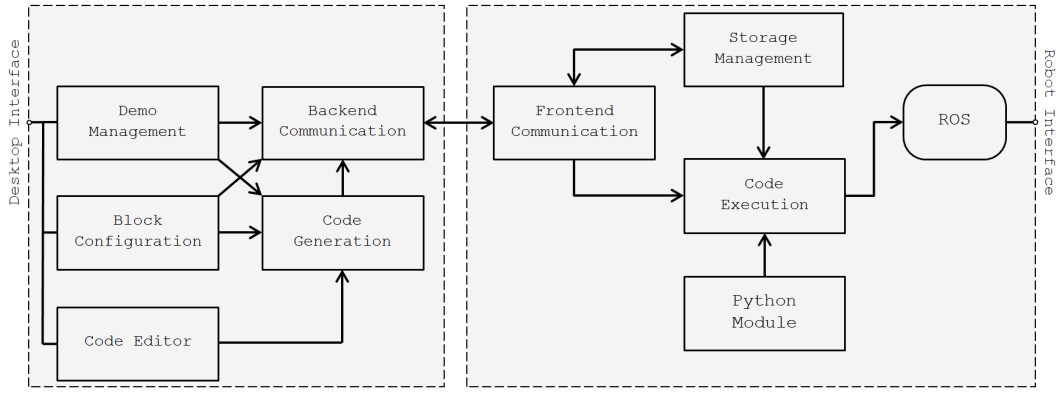


Figure 3.5: Frontend and backend architecture

3.4 Supporting frameworks & dependencies

As described the components can be clustered into frontend and backend, but they also can be seen in respect of their functional interfaces. On the one hand there is a connection to the robots sensors and actuators, on the other 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 (Robot Operating System) 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. [7]

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 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 any way. And since there are a lot of JavaScript frameworks supporting the development of web applications, it's 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

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

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 behavior: `app.get('/',...)` causes every request to trigger the given callback function, which in this case is just sending the "Hello World" string.

```
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

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 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 detail 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 Backend & Frontend Communication

The communication between frontend (client) and backend (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 [8], 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 for 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).

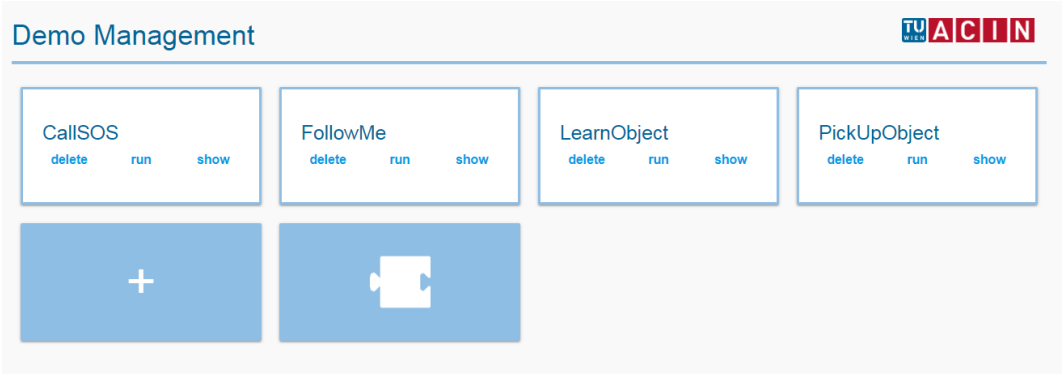


Figure 4.1: Demo Management Page

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 backend, 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 needs a user 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 doesn't 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 can 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 aren't 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)

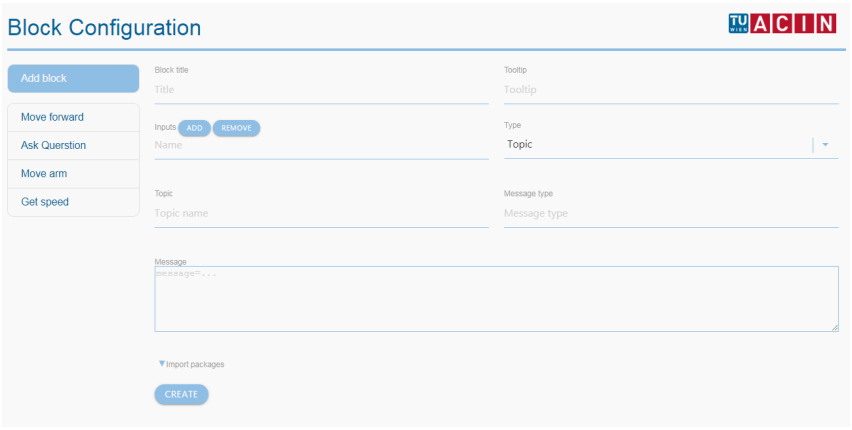


Figure 4.3: Design of the block configuration interface

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 detail 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 a 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 features 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 A.

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 connected with each other.

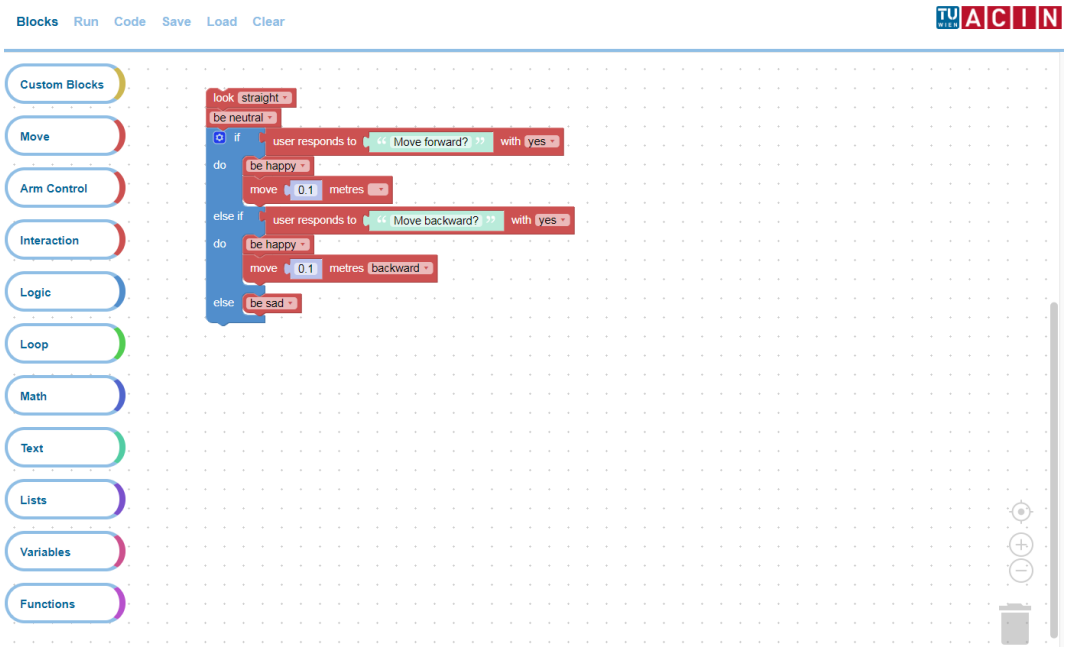


Figure 4.4: User interface for demo and code generation

Blocks which are dragged to and combined at the workspace creates 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 B. 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 examaple shows the block and code definition to create a block, which allows to move HOBBIT a given distance into a given direction (Figure 4.5).

The full block definition for the mentioned example is given in Listing 4.1. Lines 2 and 3 indicates, that the block is initialized using a JavaScript function

Figure 4.5: Example block to be created

```
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

(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 assign 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. 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 first the input values of the block are read through the provided API (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 execute the demo or save it. This could be very basic modification, e.g. just adding comments to 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 don't want to build their code from the scratch.

```
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

[1] https://blockly-demo.appspot.com/static/demos/blockfactory/index.html

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 usind 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 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 3.2 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 bes started as the first process, which in the case of HOBBIT is done at its 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 Section 4.4. It ensures that `init_node` is called from the main Python thread.

Publishing to a topic

For publishing to 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

[2]https://developers.google.com/blockly/reference/js/Blockly
[3]http://docs.ros.org/jade/api/rospy/html/rospy-module.htmlinit_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. First, a 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, so 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 first it is 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. This temporary variables then are 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 send a goal to a action

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 backend. 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 detail 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 backend (`/demo/run/`, see Table 4.1) is send, including the code inside the request body. The backend then executes the code using the *child_process* module [9] of Node.js.

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

5 Evaluation

5.1 Methods

5.2 Results

5.3 Discussion

6 Conclusion

Appendices

A 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

A.1 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 A.1, the final design is shown in Figure A.2.

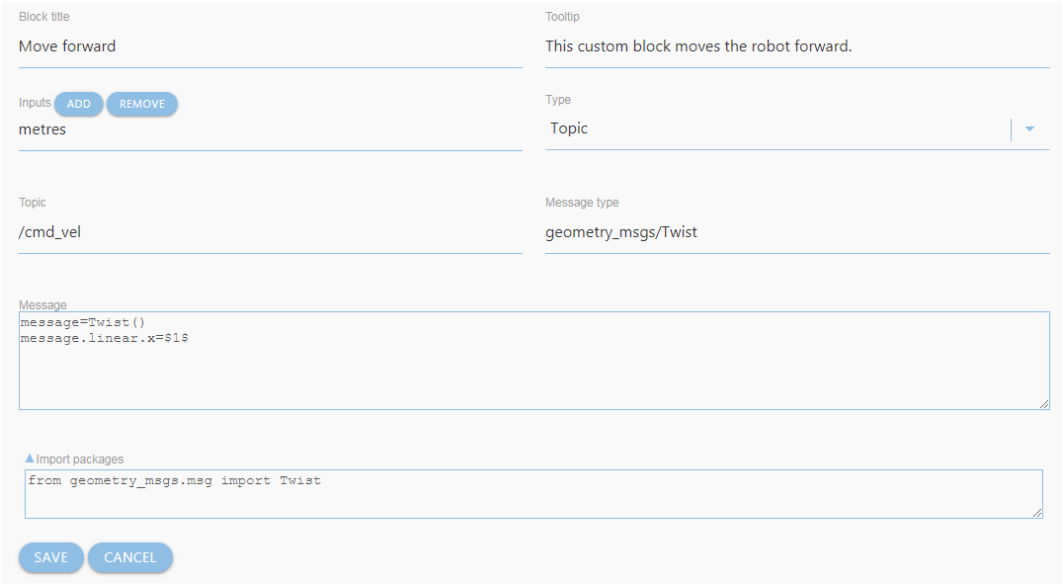


Figure A.1: Filled form to create a custom block publishing to a topic

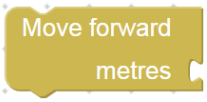


Figure A.2: Resulting custom block using the block configuration shown in Figure A.1

```
#!/usr/bin/env python
import HobbitLib
import rospy
from geometry_msgs.msg import Twist


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

    message=Twist()
    message.linear.x=1
    HobbitLib.importMsg('geometry_msgs.msg','Twist')
    DemoNode.publishTopic('/cmd_vel', 'Twist', message)

  except rospy.ROSInterruptException:
    pass
```

Listing A.1: Examplary generated code using the block shown in Figure A.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 A.1 will be generated.

A.2 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 A.3.

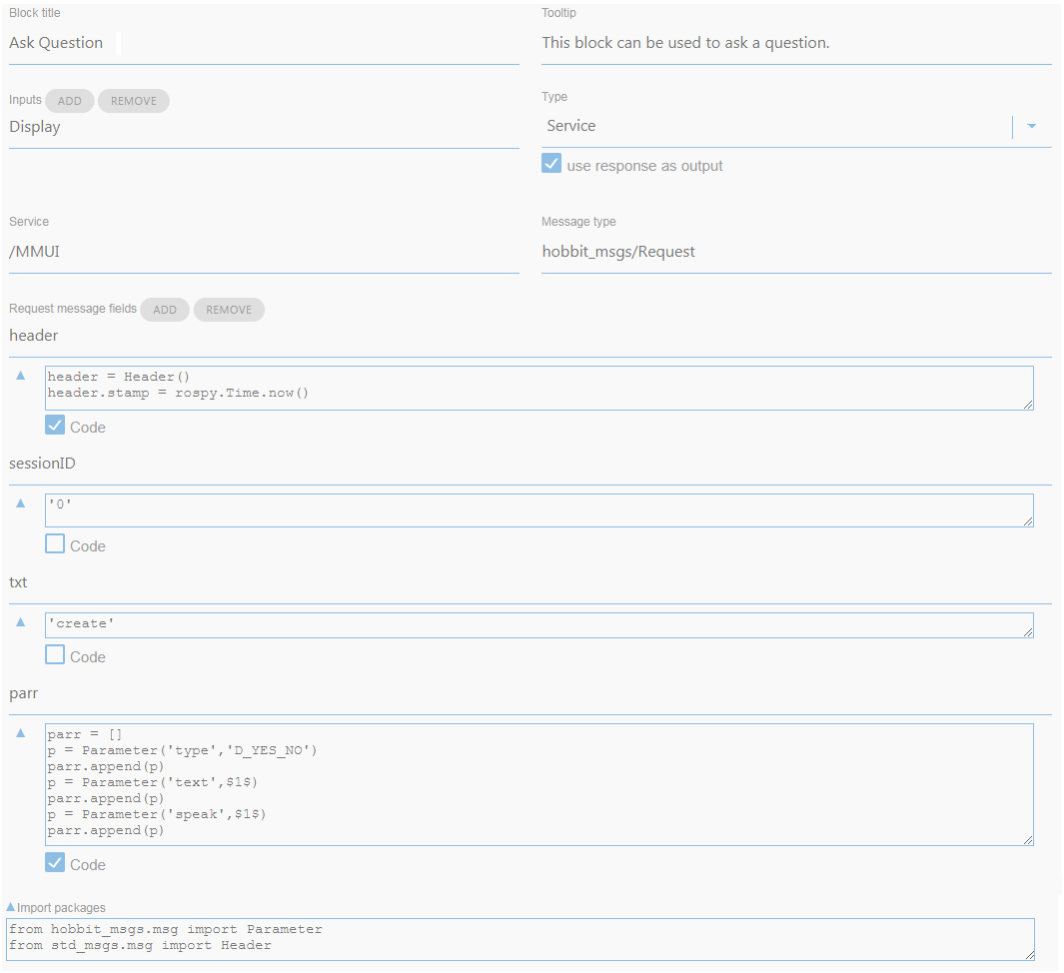


Figure A.3: Filled form to create a custom block calling a service

Besides the choice to use *Service* as communication pattern and pass the service's response, there's no noteworthy difference compared to Section A.1. 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 A.1. A exemplary use of the just created custom block is presented in Figure A.4 with Listing A.2 showing the generated code.

A.3 Custom block using actionlib

Figure A.5 shows how a custom block can be defined, if it is desired that the block should send a goal to an action server.

```python
#!/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 A.2: Generated code of the block connections shown in Figure A.4

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

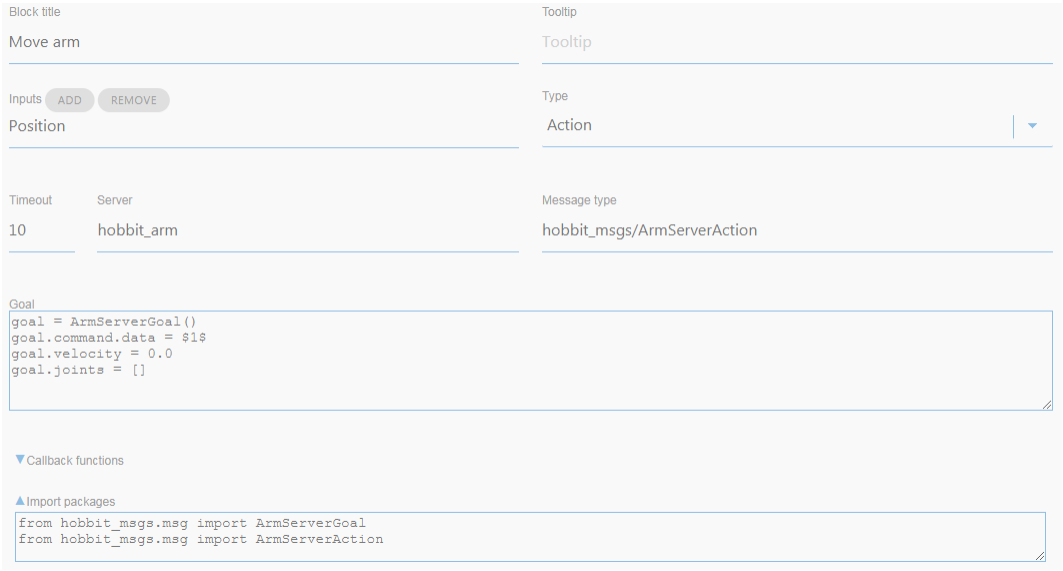


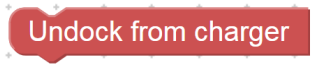
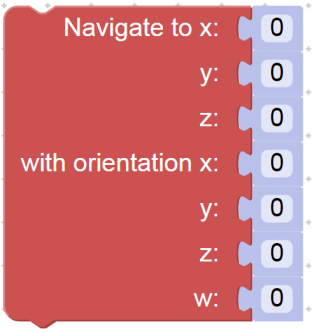
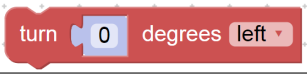
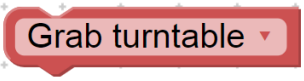
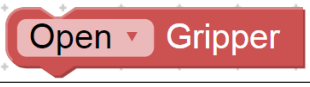
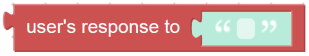
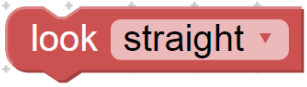
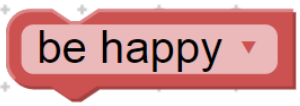
Figure A.5: Filled form to create a custom block using actionlib

B 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 [9].

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 yesâ€“no question
	Display the given question on HOBBIT's tablet and get user's answ
	Move HOBBIT's head to the given position
	Set HOBBIT's eyes according to the given emotion

Bibliography

[1] 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).

[2] ——, *ER4STEM Educational Robotics for STEM*. [Online]. Available: `https://www.acin.tuwien.ac.at/project/er4stem/` (visited on 08/17/2018).

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

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

[5] *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).

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

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

[8] 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).

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