TECHNISCHE UNIVERSITÄT WIEN

ACIN
AUTOMATION & CONTROL INSTITUTE
INSTITUT FÜR AUTOMATISIERUNGS- & REGELUNGSTECHNIK

Rockly: A graphical tool 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

<Please write an abstract here.>

Summary

Contents

List of Figures

List of Tables

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

3 Architecture

This chapter describes the architecture and implementation of Rockly (portmanteau of ROS and Blockly). First the purpose and need of it are explained, followed by a architectural overview of HOBBIT, the used robot. Based on this constraints the options implemtenting the tool are presented as well as an explanation of the decision. Then a short description of the robot operating system ROS and the fundamental JavaScript frameworks, Node.js and Express, are given and finally the necessary details of the implementation are documented - for both, the frontend and the backend.

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 3.3.1), 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

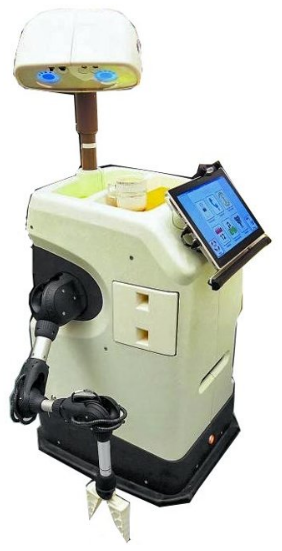

Figure 3.1: HOBBIT - The Mutal Care Robot

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 3.3.1 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.

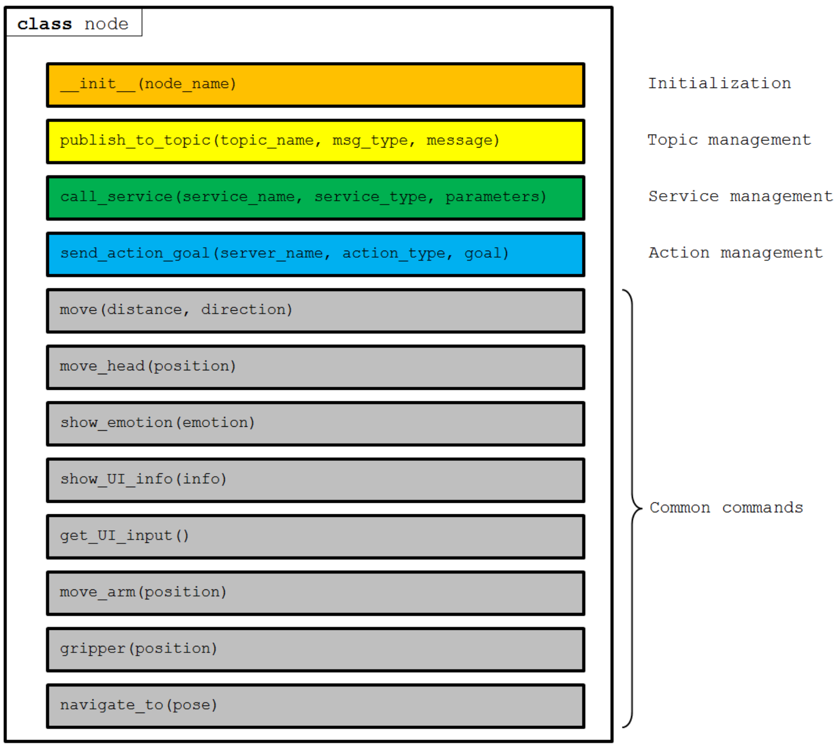

Figure 3.2: Exemplary design of API for Python

```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 3.3.1) 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

```python
from smach import StateMachine
from smach_ros import SimpleActionState
from hobbit_msgs.msg import ArmServerGoal,
   ArmServerAction
from rospy import loginfo

# Instance of SMACH state machine
sm = StateMachine(['finished','aborted','preempted'])

with sm:
    # Definition of action goals
    goal_floor = ArmServerGoal(data='
       MoveToPreGraspFloor', velocity=0.0, joints=[])
    goal_table = ArmServerGoal(data='
       MoveToPreGraspTable', velocity=0.0, joints=[])
    goal_OpGrip = ArmServerGoal(data='OpenGripper',
       velocity=0.0, joints=[])
    goal_ClGrip = ArmServerGoal(data='CloseGripper',
       velocity=0.0, joints=[])

    # Assambly of the full state machine
    StateMachine.add('INITIAL_POS', SimpleActionState('
       hobbit_arm',ArmServerAction,goal=goal_OpGrip),
       transitions={'succeeded':'FLOOR_POS','aborted':'
       LOG_ABORT', 'preempted':'LOG_ABORT'})
    StateMachine.add('FLOOR_POS', SimpleActionState('
       hobbit_arm',ArmServerAction,goal=goal_floor),
       transitions={'succeeded':'CLOSE_GRIPPER','
       aborted':'LOG_ABORT', 'preempted':'LOG_ABORT'})
    StateMachine.add('GRIPPER_CLOSED',
       SimpleActionState('hobbit_arm',ArmServerAction,
       goal=goal_ClGrip),transitions={'succeeded':'
       TABLE_POS','aborted':'LOG_ABORT', 'preempted':'
       LOG_ABORT'})
    StateMachine.add('TABLE_POS', SimpleActionState('
       hobbit_arm',ArmServerAction,goal=goal_table),
       transitions={'succeeded':'OPEN_GRIPPER','aborted
       ':'LOG_ABORT', 'preempted':'LOG_ABORT'})
    StateMachine.add('GRIPPER_OPEN', SimpleActionState(
       'hobbit_arm',ArmServerAction,goal=goal_OpGrip),
       transitions={'succeeded':'finished','aborted':'
       LOG_ABORT', 'preempted':'LOG_ABORT'})
    StateMachine.add('LOG_ABORT', loginfo('Demo aborted
       !'), transitions={'succeeded': 'aborted'})
```

Listing 3.2: Using SMACH to generate a state machine

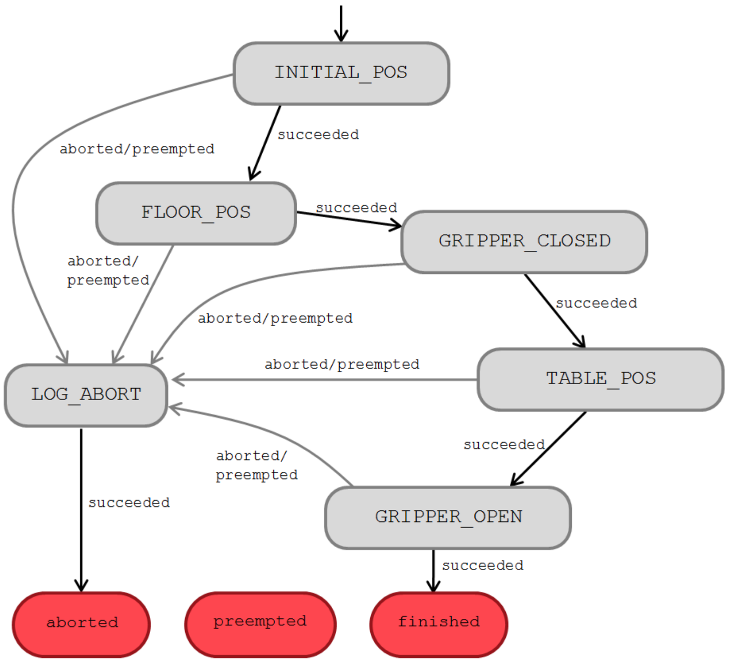

Figure 3.3: State machine generated via Listing 3.2

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, it's 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

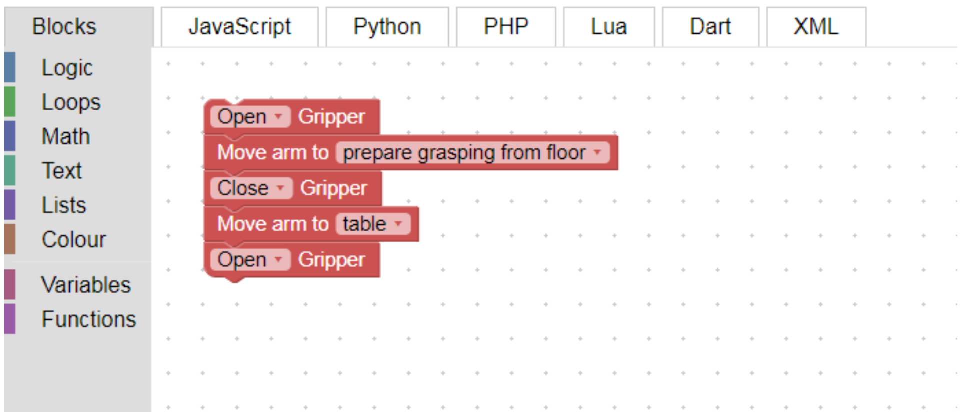

Figure 3.4: A short Blockly demo showing it's structure and use

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 drop-down 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.

```xml
<xml id="toolbox" style="display: none">
  <block type="hobbit_arm_gripper"></block>
  <block type="hobbit_arm_movement"></block>
</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 Framework

The fundamental component of Rockly's architecture (Figure 3.5) is a custom Python module, which abstracts ROS commands, but there are also several other basic frameworks used and it's necessary to have a brief overview of them.

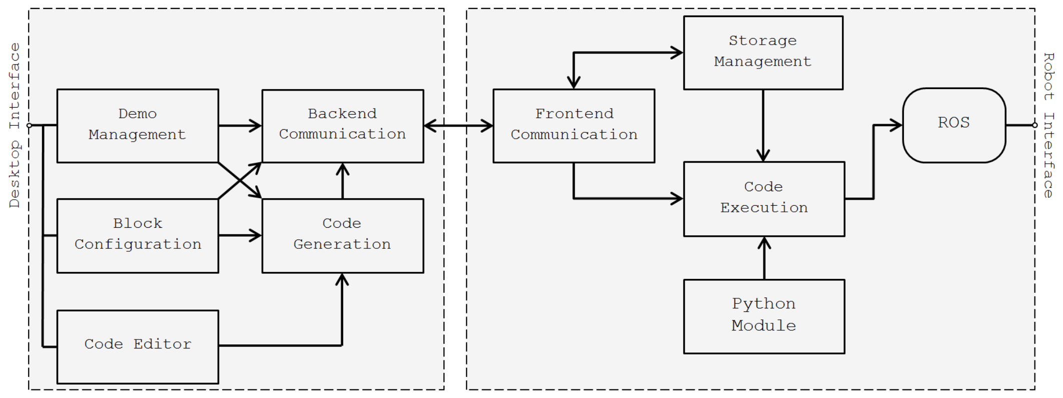

Figure 3.5: Frontend and backend architecture of Rockly

3.3.1 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 a "ROS Action Protocol", 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.

3.3.2 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 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.

3.4 Frontend implementation

The frontend of Rockly 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 managing custom blocks (**Block Configuration**, which then can be used for code generation.
- Sending and receiving data from the robot via the **Backend Communication**.

3.4.1 Demo Management

The Demo Management (Figure 3.6) 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 explicity to locate the file and run. Since usability and simplicity are main requirements for this tool, such a feature is a main advantage of it.

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.

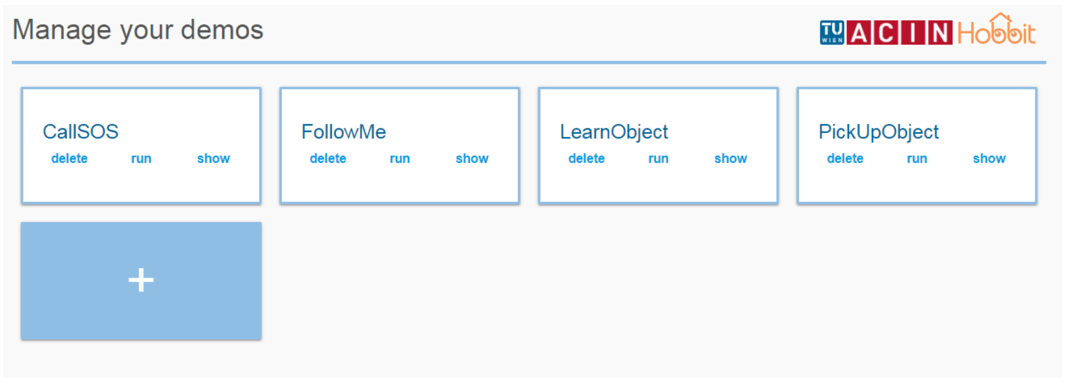

Figure 3.6: Demo Management Page

3.4.2 Code Editor

3.4.3 Code Generation

3.4.4 Block Configuration

3.4.5 Backend Communication

3.5 Backend implementation

The frontend of Rockly consists of the following four components:

- 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**).

3.5.1 Frontend Communication

3.5.2 Storage Management

3.5.3 Python Module

3.5.4 Code Execution

4 Evaluation

4.1 Methods

4.2 Results

4.3 Discussion

5 Conclusion

A Appendix

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

Table A.1: Possible messages for topic */head/move*

A.1 ROS reference for HOBBIT

sadadasdasdf

Listings

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).