TIM STEUER

THE BAXTER ROBOT: EXPLORING THE ROBOT BY RAPID PROTOTYPING

THE BAXTER ROBOT: EXPLORING THE ROBOT BY RAPID PROTOTYPING

TIM STEUER

Introducing a C++ Framework for Rapid Prototyping Demo Scenarios

Applied Computer Science
School of Engineering
HTW des Saarlandes

Tim Steuer: *The Baxter Robot: Exploring the robot by Rapid Prototyping,* Introducing a C++ Framework for Rapid Prototyping Demo Scenarios,© July 2014 - September 2014

SUPERVISORS:
André Miede
Ehud Sharlin

LOCATION:
Calgary

TIME FRAME:
July 2014 - September 2014

ABSTRACT

The Baxter research robot has due to its orientation towards soft robotics potential application areas in human robot interaction as well as in the industry. To evaluate the pros and cons in these application areas, a rapid prototyping of demo scenarios is useful. However, the research robot's API provides only access to the hardware without offering high level abstractions. Consequently, developing of demo scenarios repeatedly includes the implementation of common features like motion planning, manipulation or object perception. While a rewriting of these software parts might be beneficial for specialized applications, it hinders rapid prototyping aiming for evaluating a hypotheses with minimum programming effort. We introduce in this thesis a rapid prototyping framework for the Baxter robot. It accelerates demo development through rapid prototyping for evaluating hypotheses in human robot interaction research or industry prototypes. As a result, researchers and developers can focus on their hypothesis and using the Baxter robot for rapid prototyping becomes easier.

CONTENTS

1 INTRODUCTION 1
- 1.1 Motivation 2
- 1.2 Results 3
- 1.3 Organization 3

2 ANALYSIS 5
- 2.1 Introduction to the Baxter robot 5
- 2.2 Methodology 7
- 2.3 Workspace 8
- 2.4 Feature analysis 8
  - 2.4.1 Arm movement 9
  - 2.4.2 Multi-arm coordination 12
  - 2.4.3 Perception 14
  - 2.4.4 Manipulation 15
  - 2.4.5 Human Robot Interaction 18
- 2.5 Framework Design Fundamentals 20

3 FOUNDATIONS 23
- 3.1 Robot Operation System 23
- 3.2 AR Track Alvar 28
- 3.3 MoveIt! 29
- 3.4 Design Patterns 30
- 3.5 Readers Writers Problem 33
- 3.6 Construction vs. Usage 35

4 SYSTEM ARCHITECTURE 37
- 4.1 acamp_moving 38
- 4.2 acamp_perceiving 40
- 4.3 acamp_faces 43
- 4.4 acamp_planning 46
- 4.5 acamp_gripper_state_publisher 51
- 4.6 acamp_fixed_environment 51

5 IMPLEMENTATION DETAILS 53
- 5.1 Prototypes 53
- 5.2 Workspace Setup 55
- 5.3 Adapting for MoveIt! 56
- 5.4 Spline Interpolation Error 60
- 5.5 Perceiving Package 63

6 RELATED WORK 69
- 6.1 Projects in the blogosphere and YouTube 69
- 6.2 Research papers 69

6.3 Distinctive qualities of our work 71

7 RESULTS & DISCUSSION 73
7.1 Results 73
7.2 Discussion 77

8 FUTURE WORK 79

BIBLIOGRAPHY 83

LIST OF FIGURES

Figure 1 The Baxter robot 5
Figure 2 The layout of the end effector 6
Figure 3 The capacitive cuff buttons enabling the zero-G mode 7
Figure 4 Our Baxter robot on its workplace in the laboratory 9
Figure 5 Different types of robotic joints 10
Figure 6 The rotation axis of the arm 10
Figure 7 The different joint types in the kinematic chain 11
Figure 8 A motion that leads to a collision when solved with an IK-Solver 11
Figure 9 Sample-based motion planning from start (circle) to goal (cross) 13
Figure 10 Partition of the table working area in the multi-arm approach 14
Figure 11 Changing the camera position 16
Figure 12 Different grasping approaches 16
Figure 13 Grasp stages 17
Figure 14 The uncanny valley 19
Figure 15 An example for a State Machine 21
Figure 16 Node Communication 24
Figure 17 Client Service Communication 25
Figure 18 Actionlib Communication 25
Figure 19 Two different tf frames of the robot. The position of the object can be specified relative to the end effector's frame. 26
Figure 20 Simulated robotic environment 27
Figure 21 Four fiducials with a one dollar coin as size reference 28
Figure 22 Singleton Pattern 31
Figure 23 Strategy Pattern 32
Figure 24 Template Method Pattern 32
Figure 25 Framework structure 37
Figure 26 acamp_moving Package class structure 39
Figure 27 acamp_perceiving Package class structure 41
Figure 28 acamp_faces Package class structure 44
Figure 29 acamp_planning Package class structure 47
Figure 30 A search_state as an UML Sequence Diagram 48
Figure 31 Gripper fingers obstructing the view of the camera 49
Figure 32 Network setup 56

Figure 33 Finger mesh 59
Figure 34 A quadratic spline composed of six polynomial segments [1]. 61
Figure 35 The weights of the moving average plotted for $n = 15, \alpha = 0.15$, the largest weight is for the newest datum 65
Figure 36 Calculation of a moving average illustrated 66
Figure 37 Sorted Lego pieces after the rainbow demo 73
Figure 38 Single arm State Machine used for benchmarking 74
Figure 39 Changes of the rainbow placing State Machine that led to the single arm box placing State Machine (highlighted in green) 75
Figure 40 Multi-arm State Machine build from two single arm machines 76
Figure 41 A typical automated optical inspection machine for electronic boards 79
Figure 42 A possible workspace of Baxter at the DSM assembly line. Close to the workers that do the functional tests 80

LIST OF TABLES

Table 1 Boost synchronization primitives 33
Table 2 Unpublished work on Youtube and in the blogosphere 70
Table 3 Benchmark results Python vs. C++ Node 74

LISTINGS

Listing 1 Solving the *Readers Writers Problem* with boost primitives 34
Listing 2 Introducing tight coupling through a violation of *Construction vs. Usage* 35
Listing 3 The *BlindSearchGoal* message that defines the searching rectangle 40
Listing 4 Aliases in the .bashrc making the work easier 55
Listing 5 The *gripper_state_publisher* Node 57

Listing 6 The *RobotTrajectory* message and its parts 60
Listing 7 Fixing the spline interpolation bug 63
Listing 8 The data structure used to store the raw sensor data 64

ACRONYMS

HRI Human-Robot Interaction

acamp Alberta Centre for Advanced MNT Products

RSDK Research Software Development Kit

ROS Robot Operating System

IK Inverse Kinematics

OMPL Open Motion Planning Library

URDF Universal Robot Description Format

DOF Degree Of Freedom

EWMA Exponentially Weighted Moving Average

gcc GNU Compiler Collection

DSM Dynamic Source Manufacturing

SMD Surface Mounted Device

UML Unified Modeling Language

tf transformation

API Application Programming Interface

FCL Flexible Collision Library

SRDF Semantic Robot Description Format

NTP Network Time Protocol

DHCP Dynamic Host Configuration Protocol

IDE Integrated Development Environment

MAC Media Access Control

IP Internet Protocol

TCP Transmission Control Protocol

STOMP Stochastic Trajectory Optimization for Motion Planning

1 INTRODUCTION

The idea of artificial life forms is very old. It is already present in ancient mythology, for example, in the Jewish legends of the manmade clay golems or in the Greek legends around the mechanical servants made by the god Hephaestus. In these ancient stories, the artificial life forms often act as a servant for their constructor. They interact with them, obey orders, or work side by side with people. In modern literature the term *robot* is often used for such a life form. First used in 1921 in the play *Rossum's Universal Robots* by the Czech writer *Karel Čapek* [8] it fast gained popularity through various other stories like *Isaac Asimovs Nighfall*. In these stories robots are reintegrated into human society and work as servants together with us.

However, the modern development of robots has taken another path. Starting in the 1960s, robots were introduced into various industries as relatively simple autonomous helpers , for example the *Unimate* robot that revolutionized General Motors production line[27, pp. 79]. They spread quickly because of their advantages in cost and performance compared to a human worker. This mass distribution led to more sophisticated robots with specialized capabilities in many industries. However, unlike imagined in literature, modern robots are rarely able to work co-located to humans or to communicate with them. They are often not built for this purpose and are too dangerous because of their focus on specialized working performance. In other words, they trade working speed for safety. In addition, most industry robots are rarely adaptable to tasks other those they were built for, making them inadequate to be used in a company that changes its production quickly [19, pages 9-13].

This development in robotics has led to a hesitant integration of automation in many fields. We do not have the automated servants as imagined in literature yet. Even though researchers in the field of Human-Robot Interaction (HRI) are working hard on this vision, the realization of it is still far away. However, there are niches for new ideas between the use of a robot as a personal servant at home and the now existing industry robots.

Rethink Robotics targets one of these niches with their in 2013 released Baxter robot. It aims to be the missing link between a heavily specialized and dangerous industry robot and a comparatively slow and expensive human worker. This means that it is safe to work among humans through newly developed safety concepts and it is more adaptable to abstract tasks.

While Rethink Robotics tries to find places for the robot as a commercial product, they also offer a research version of it. This version comes without the commercial software and is freely programmable.

1.1 MOTIVATION

The Baxter robot is an exciting research platform thanks to its new approach towards more co-located work and a more adaptable system. Besides this, it promises a good cost effectiveness making it interesting for middle scale industries. As a result the Alberta Centre for Advanced MNT Products (acamp) has purchased two Baxter research robots as the use cases of the robot fit into their business model to bring the Alberta industry forward through innovative techniques. To explore the capabilities of the robots acamp collaborates with the University of Calgary by providing it with one robot.

The motivation for this project stems from the different visions of the two project partners. The *uTouch* research group of the University of Calgary aims to explore the HRI capabilities of the Baxter robot in various studies. Due to the robot's safety features, its physical dimensions and its humanoid appearance it is an ideal candidate for HRI research studies like finding out what makes such a robot acceptable as a co-worker and the effects different collaboration behaviors have on humans.

On the other hand, acamp would like to explore the opportunities and limits that come with a Baxter robot as an industrial tool. While the commercial manufacturing version already shows some capabilities of the hardware, it is primary made for trainable tasks that only deal with a minimum of randomness. However, different industries in Alberta could have different use cases that cannot be realized with this manufacturing version. To introduce the Baxter robot to such companies acamp would like to have custom demos showing the capabilities of the robot besides the manufacturing version.

With these visions in mind both project partners are aware of the novelty of the robotic platform. Hence, this project is merely the start of working on these visions and does not aim to complete the different requirements arising from it. It rather shall form to a good base for further work, give a first overview of the platform and provide the fundamental knowledge needed to understand the limitations of the Baxter robot. Thus, unifying these goals, the outcome of this project will be comprised in a framework that allows the use of the core capabilities of Baxter for a rapid prototyping of demo scenarios. The evaluation of this framework is done by building demo scenarios that operate with Lego Duplo[1] blocks. We focus on Lego because they are a good abstraction mechanism for industry tasks as many tasks like constructing or sorting are transferable to a similar industry tasks.

1 in the following, just Lego for simplicity in this thesis

This framework can then be used as a base for the exploration of HRI challenges and the industry challenges in subsequent projects.

1.2 RESULTS

This thesis encompasses three different outcomes. First of all, we gained knowledge about the platform and what is achievable with it and what not. This is useful to manage the time constraints, scope and expectations of future projects. This knowledge is documented in this thesis. While the discovered capabilities are especially discussed in Section 2.4 the restrictions of the platform are mainly discussed in Section 5.1. We discovered for example restrictions regarding sample-based motion planning and the parallel arm usage, or restrictions that apply to the general object recognition with the hands.

Second, we developed an easy to use rapid prototyping software framework in this thesis. This framework is intended to speed up further project's development through an easy to use Application Programming Interface encapsulating low-level task to make them easily reusable. It includes abstractions for features like collision aware movement, sequential multi-arm usage, picking, placing, object search as well as recognition and basic Human-Robot Interaction. The framework's design is focused on easy usage rather than complexity and advanced capabilities.

Third, we developed demo scenarios showing the capabilities of the framework. In these demo scenarios, the Baxter robot sorts multiple randomly distributed Lego pieces. The whole movement of the robot is collision aware. The implemented demos can operate on one or both arms showing the adaptability of the framework through the development of a simple first demo (one arm) to a more sophisticated second demo (two arms).

1.3 ORGANIZATION

Chapter 2 encompasses the analysis of the problem domain and the inferred challenges for this thesis. Furthermore, it introduces general theoretical approaches to solve these challenges and familiarizes the reader with the Baxter robot. Finally, it introduces the reader to the framework design principles followed in this thesis.

Chapter 3 describes important libraries and underlying concepts. Most importantly, it introduces the reader to the Robot Operating System as it is the base for all development related to the Baxter robot. Besides robotic related concepts it also explains essential software design concepts that are used later.

Chapter 4 illustrates the software architecture developed to implement the analyzed features. The architecture is explained in detail using the Unified Modeling Language to formalize the concept and its collaborators. Furthermore, the Robot Operating System's specific project structure and the resulting separation of the features is explained.

Chapter 5 covers specific details of our work. It does not provide a complete explanation of every specific detail, but rather focuses on certain examples that are representative for the problem classes that were solved in this thesis. The described problems differ from each other and highlight key aspects of problems that arose multiple times during the implementation.

Chapter 6 makes the reader familiar with the related work by looking at related published papers and briefly summarizing them. It also provides an overview about the projects that were published outside academia on *youtube* and in the *blogosphere*. Finally, the distinctive qualities of this thesis are highlighted.

Chapter 7 discusses the final result of the thesis with focus on quantitative and qualitative results. It provides performance measurements showing the speed and reliability increase compared to naive implementations of features and discusses the demo perception of viewers.

Chapter 8 points out opportunities and challenges that are well suited for further projects with the framework and the Baxter robot. It presents a subsequent project idea that introduces the robot into a real industry scenario.

2 ANALYSIS

2.1 INTRODUCTION TO THE BAXTER ROBOT

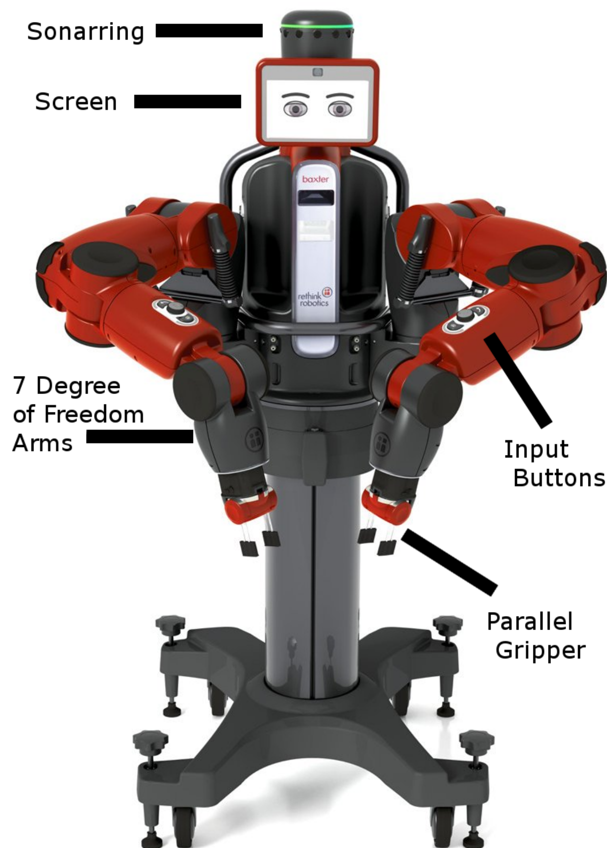


Figure 1: The Baxter robot

The Baxter robot is a new robotic platform introduced by Rethink Robotics in 2013. It is a still standing humanoid robot with two arms for manipulation and a movable head (see Figure 1). The company breaks fresh ground with this robot as it distinguishes itself in various ways from standard industry robots. Most importantly, it is intended to work in a different application domain than normal industry robots.

While industry robots are supposed to work in an environment built for their needs separated from humans, the Baxter robot will work co-located to humans or even hand in hand with humans. To realize this concept Rethink Robotics has implemented a new safety concept in the robot guaranteeing that the robot is completely harmless for humans. The robot stops automatically, for example, when-

Figure 2: The layout of the end effector

ever it hits something with too much force [26]. Another important approach Rethink Robotics pursues is the ability to solve many different tasks with the robot rather than one specialized task like a standard industry robot.

To do so, the robot is equipped with various sensors and actuators. It has two seven Degree Of Freedom (DOF) arms with an arm length of 1 m each. Thus, every arm is controlled by seven servos (see Section 2.4.1). At the end of each arm there is a mounting point for different types of end effectors. Rethink Robotics provides an electrical parallel gripper (see Figure 2) with exchangeable fingers and a vacuum cup end effector as default. Furthermore, the specifications for the end effector connectors are published and other end effectors are available [33].

The robot can carry 2.3 kg payload per arm and has a maximum movement speed of $1\,\frac{\mathrm{m}}{\mathrm{s}}$ on each empty arm and $0.6\,\frac{\mathrm{m}}{\mathrm{s}}$ on each loaded arm. While this is slow compared to a heavily specialized industry robot, it is sufficient for most abstract tasks that need to be solved at an assembly line.

For sensing the robot has three cameras: one at each hand and one on the head with a maximum resolution of 1280x1024 pixels. In addition, the hands also include an Infrared Rangefinder and, depending on the gripper, a pressure sensor. The head encompasses a screen with a resolution of 1024x600 pixels as well as a sonarring for sensing a 3D point cloud and is movable along the sides and can nod. Each arm can detect acceleration, velocity, linear twist and the force currently effecting it. For input and output the robot has various buttons on its chest and arms as well as several lights on its arms, chest and head.

Rethink Robotics delivers the robot in two different versions. The first version is a manufacturing version that comes with commercial

software and is able to perform a variety of tasks after training the robot by demonstration[26]. The training is based on the ability to move the robot's arms freely when the cuff buttons are pressed (see Figure 3). This triggers the training mode, allowing the user to teach the robot an action by showing arm trajectories and selecting actions from an interface displayed on the head. Such actions are, for example, to pick from the location the hand is currently over or to repeat the movement just showed.

Figure 3: The capacitive cuff buttons enabling the zero-G mode

The other version Rethink Robotics offers is the research version used in this project. Lacking Rethink Robotics' commercial software it is not ready y to be used immediately, like the manufacturing version and misses the training mode. However, it is freely programmable and has a Python and C++ Application Programming Interface (API) exposing complete control over the robot.

2.2 METHODOLOGY

We have chosen an iterative approach for the project. This decision was based on our missing experience with the platform and resulted from the vague requirements the project partners had at the start of the project. Due to these vague requirements, only these features important to the overall vision must be considered.

Iterative approaches are very common in software development and research as they are designed to deal with a high level of uncertainty. One of the very first iterative models in software development was the Spiral Model [6]. It states that iterative development is risk driven. This means the risk determines the level of detail and effort that should be spent in each cycle. In contrast to a low risk cycle, a high risk cycle may include fewer distinctive work items that, however, require a high degree of analysis. Work items are, for example, library evaluation or programming a feature. In addition, the Spiral Model defines risk evaluation as a key task in each cycle. So it is important to manage the expectations of the stakeholders and to adapt the project based on their needs.

Our model implemented these risk reduction features through the following tasks. First, bi-weekly status reports were part of every iteration, keeping the stakeholder in the information loop with the ability

to provide feedback. They also contained a detailed list of the proposed, remaining and finished work items of each iteration. Second, milestones were planned at the beginning of the project defining the dates when the current state of the project needs to be shown to the public. Hence, possible demo scenarios could be evaluated and implemented timely before the actual demo. Third, monthly project meetings were held to show the overall project progress, to manage the expectations of the stakeholders and to allow an exchange of ideas and suggestions. Based on this, a concrete iteration of our process normally comprised the following steps:

1. Explore the most basic feature that is important for the stakeholders
2. Implement the feature into the framework
3. Demonstrate the newly integrated feature in the monthly meeting

Sometimes an iteration included more than one feature and sometimes two iterations were needed to release a new demonstration.

2.3 WORKSPACE

To show the abilities of the robot we thought about a task that is booth familiar and entertaining for the viewer. In addition, as the robot has weight and movement speed restrictions, the task should be lightweight and doable in a reasonable amount of time. Therefore, our Baxter robot performs its demo tasks with *Lego* on a table (see Figure 4).

While it might not seem reasonable in the first place to perform such a task when the project vision is an industry usage of the robot, working with Lego pieces is closely related to many industry tasks due to similar spatial challenges. Consequently, motion, object recognition and manipulation tasks that can be demonstrated with a Lego piece can be transferred to an industry related task as long as the physical constraints like weight restrictions remain similar. For example, sorting Lego pieces can be transferred to the corresponding industry task of sorting or categorizing manufactured parts by color or quality.

2.4 FEATURE ANALYSIS

The following features were found during the iterations of the project. Each feature was refined in multiple iterations to its final stage of expansion. The different parts of the features were first explored and introduced in the framework in the following order:

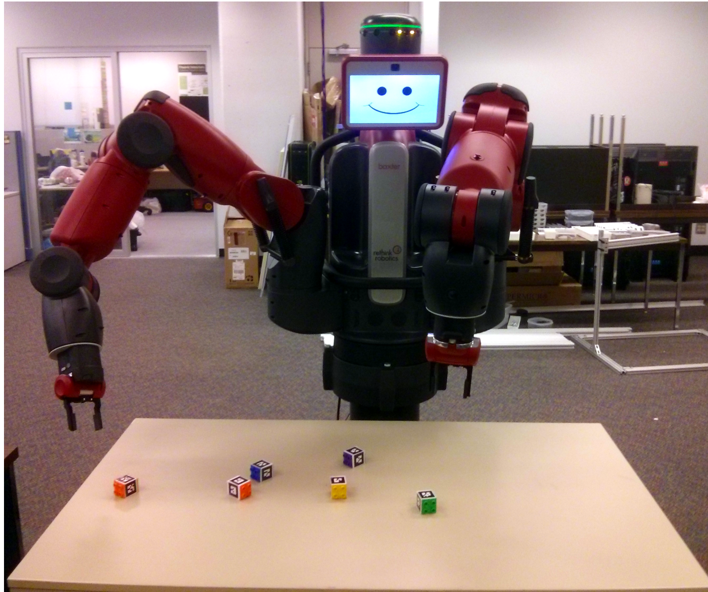


Figure 4: Our Baxter robot on its workplace in the laboratory

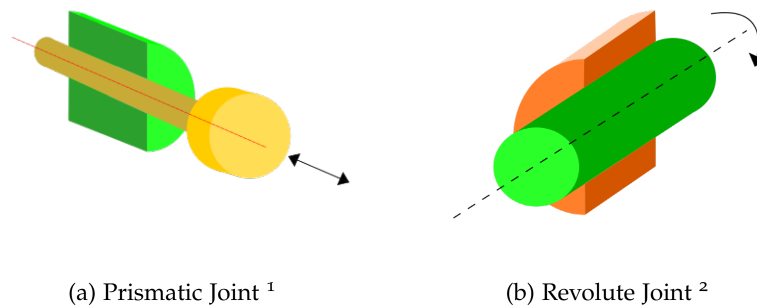
1. Movement through Inverse Kinematics
2. Robotic perception
3. Motion planning (collision aware)
4. Robotic manipulation
5. Human-Robot Interaction
6. Multi-arm motion planning

The analysis below shows each features challenges and their theoretical solution.

2.4.1 *Arm movement*

The arms are the only part of the Baxter robot that are suitable for manipulating objects. As a result, arm movement might be considered as the robot's most important functionality.

Arm movement is a spatial function operating in all three dimensions. It is also a physical operation and therefore restricted by hardware constraints. Typically the hardware underlies position, velocity and acceleration constraints. In our Baxter robot, for example, every joint has a maximum velocity of $1\ \frac{m}{s}$ and is either a *revolute joint* or a *prismatic joint* with a minimum and maximum position. A *revolute joint* has one DOF and moves rotational(see Figure 5b). A *prismatic joint* has also one DOF and moves straight along a certain axis (see Figure 5a).



(a) Prismatic Joint [1]　　(b) Revolute Joint [2]

Figure 5: Different types of robotic joints

Since each arm consists of a chain of revolute and prismatic joints connected by rigid bodies called links (see Figure 7), we have to specify the position of each of these joints when we want to move the arm to a certain position. Such an arm position is described as an arm configuration Q. The configuration Q of the arm is defined as $Q = (q_1, q_2, q_3, ..., q_7)$ where q_i specifies the position of the joint i. Moving Baxter's arm, for example, to an object that is relative to the robot at position $x = 0.3\,m, y = 0.2\,m, z = 0.5\,m$ would require to specify an arm configuration Q that corresponds to this position.

In robotics this problem and its solutions are summarized under the term Inverse Kinematics (IK). An IK-Solver is a program that calculates the arm configuration Q given a pose $p = (x, y, z, \alpha, \beta, \gamma)$ for the end effector link. While the coordinates x, y, z are the usual spatial translation as we intuitively use it and α, β, γ are the rotation angles of the end effector (see Figure 6) in degrees.

The way the IK-Solver finds the solution varies depending on the robot and the IK-Solver used. Common methods are either analytical, geometrical or numerical and depend on the properties of the robot.

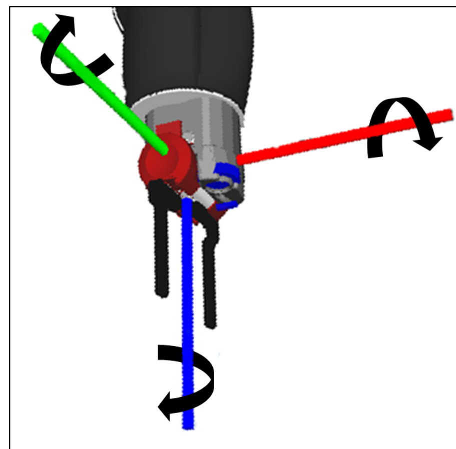


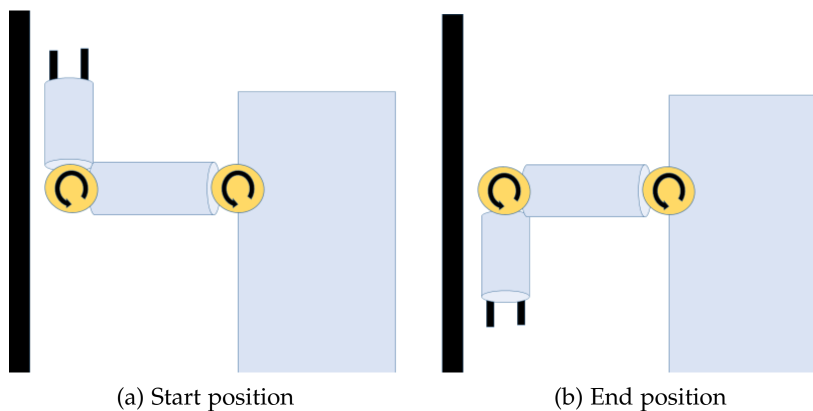
Figure 6: The rotation axis of the arm

An IK-Solver enables us to find the desired joint values for a pose p. However, another problem remains. Setting the joint values for the arm causes every joint to move into the specified position ignoring possible collisions due to the executed trajectory.

1 Prismatic joint by - Jameson L. Tai talk Licensed under CC BY-SA 3. via Wikimedia Commons
2 Revolute joint. Licensed under PD via Wikimedia Commons

Figure 7: The different joint types in the kinematic chain, all joints are marked by an arrow. Only the gripper joints are prismatic.

Collisions can occur even when it is not immediately evident. Imagine for illustration an arm configuration with only two joints (see Figure 8). We have the end effector link close to an obstacle facing



(a) Start position (b) End position

Figure 8: A motion that leads to a collision when solved with an IK-Solver

upwards. All we try to do is to move the end effector to a new pose facing down. An IK-Solver calculates the new position for each joint. The joint connecting robot and arm remains in the same position while the joint connecting arm and end effector moves. As a result there is a collision between end effector and wall during the motion even though start and goal pose are collision free.

A *Motion Planner* addresses this problem. Motion planning is the process of finding a trajectory between one pose and another without violating movement constraints [30]. The constraints given vary depending on the aim of the movement. The example above deals with collision constraints. Further constraints are, for instance, orien-

tation constraints like always moving a glass of water with the opening pointing upwards or path constraints like moving from start pose to goal pose in a straight line.

The algorithms used in *motion planning* for high DOF problems are usually sample-based. The general steps applied by different sample-based algorithms when planning a way from pose p_{start} to p_{end} are listed below. Inhere, C is the set of all possible arm configurations and C_{free} is the set of all arm configurations that are not in collision with the environment [30].

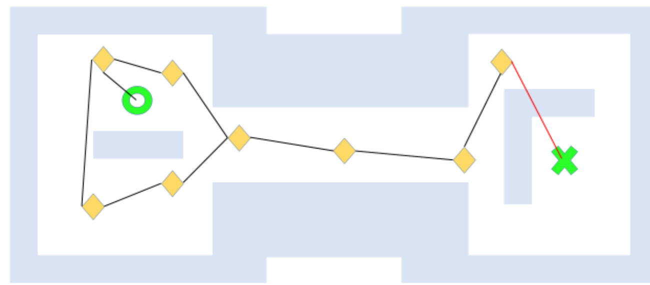
1. Sample a set $S \subseteq C$ arm configurations
2. Retain all $Q_i, Q_j \in S \cap C_{free}$
3. Create a graph that connects Q_i, Q_j if the whole way between Q_i and Q_j is collision free.

The algorithm can now find a collision-free trajectory between p_{start} and p_{end} if the constructed graph allows for a way between p_{start} and p_{end}. In contrast, not finding a way between P_{start} and P_{end} does not prove that no such way exists as the planner might not have sampled enough milestones to find the path (see Figure 9). In such a case running the algorithm multiple times might increases the chances to find a trajectory as the sample-based algorithms converge against a solution if it exists and unlimited time is spent. However, after multiple tries without a solution it is probable but not guaranteed that no trajectory between p_{start} and p_{end} exists and we have to choose another endpoint.

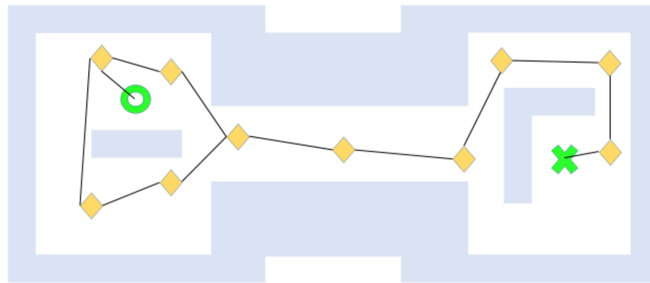
2.4.2 *Multi-arm coordination*

The discussed basic movement (see Section 2.4.1) is sufficient for operating one arm. However, it is beneficial in most of the time to use both arms together as the combination of both arms leads to a larger workspace, a higher working speed and the ability to solve more sophisticated tasks.

Although dual arm movement leads to advantages, it introduces two new challenges to motion planning. First, the algorithms for motion planning are primarily designed for single-arm operation. When it comes to multi-arm planning, the collision checking needs to perform additional computations. It is then possible, that the arms collide or get in the way of each other during the execution of the motion. While these additional checks are implemented in most sample-based motion planning algorithms, the resulting trajectories are cumbersome and the planning time increases drastically (see Section 5.1). Consequently, the robot only operates sequential in multi-arm mode



(a) Too few configurations sampled



(b) Valid plan due to the additional configurations sampled

Figure 9: Sample-based motion planning from start (circle) to goal (cross)

during this project. Second, even if the arms move sequentially reducing the complexity for the motion planner, they can still obstruct each other's way. An example situation would involve the following steps:

1. Both arms are in not obstructing positions
2. The left arm picks a Lego piece from the table and places it in a box standing in the middle of the table and remains there
3. Control of the arms switches: the right arm picks a Lego piece from the table
4. The right arm's placing path is now obstructed by the left arm that is still over the box

The solutions for this heavily depend on the exact task performed by the robot. One approach is to define working areas for each arm and a shared working area(see Figure 10). The non-shared areas can only be used by the corresponding arm. If an arm is in the shared area before it hands the control over to the other arm, it first has to leave the shared area and move into its non-shared area. This guarantees that the next movement will always happen in a state where the arms cannot obstruct each other. This is a better solution than to move the arms into a neutral position after every move. It requires in general only an additional move if an arm switches control while being in the shared area.

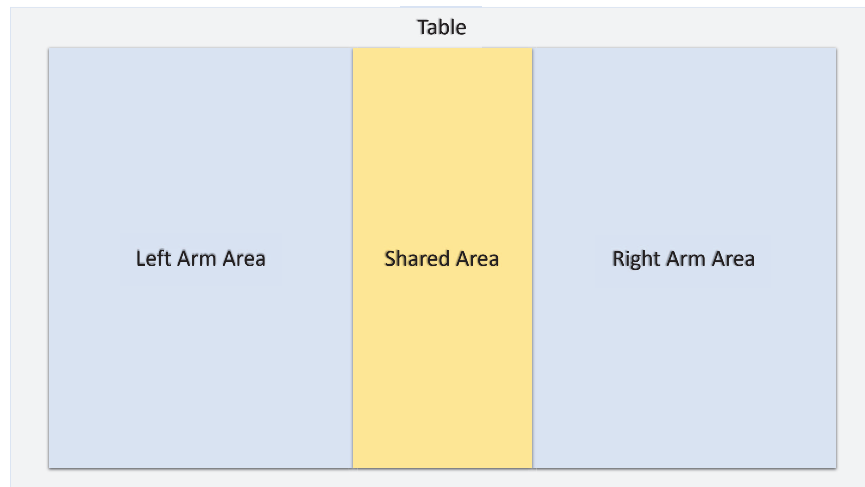


Figure 10: Partition of the table working area in the multi-arm approach

2.4.3 *Perception*

Perception is the second most important ability of the Baxter robot. While it is indeed possible to solve some basic tasks completely by repetitive movement without perception, a good perception is essential for any task that involves interaction with a changing environment. In such an environment the robot needs to perceive changes and react to them, for instance, an adjustment of the arm pose when trying to grasp a randomly distributed Lego piece.

Therefore the Baxter robot is equipped with many sensors as described in Section 2.1. However, an effective work with some of these sensors requires effort, as the sensor data needs additional processing before it is usable. An exception are the sensors located directly in the arm reporting acceleration, velocity, pressure, twist and force. This data is directly usable and in physical standard format.

Camera, IR-Sensor and sonar ring depend on preprocessing, as the raw sensor data lacks structure and semantics. We focus on the camera as we do not use the other two sensors in our current project.

The raw data structure of an image is nothing more than a large matrix containing a color information in each cell. Consequently, processing the data aims first for a better structuring of the raw data. The usual approach for the structuring of image data is image segmentation. Its intention is to create meaningful pixel groups based on the pixel matrix described above.

Meaningful groups encompass pixels that belong to the same real world object, edge or boundary. By finding such groups the image structure becomes clearer. Thereafter, feature extraction and pattern

recognition methods are usually used to map the visual representation to a real world object providing us with semantic value.

Although these algorithms enable us to identify objects on a camera stream, we still miss any spatial information about the object. This means that the robot is not able to interact with the object because its coordinates are unknown. Inferring spatial information from an image requires a transformation from 2D space into 3D space as the image is missing depth information. Such a transformation can be based on different methods, common methods are:

- Transformation through a known object size
- Transformation through a known distance between camera and object

These two methods have in common that they need to consider parameters like lens distortion or dots per inch of the camera.

After applying one of these transformation methods we have successfully processed the raw camera data to more meaningful and useful data that is suitable for planning and moving the arm to an object.

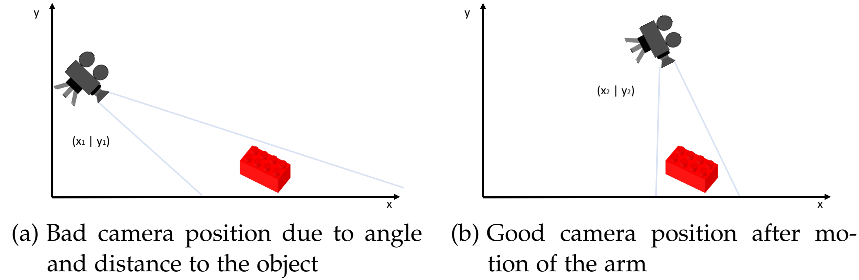
However, a spatial estimation of an object's pose extracted through image processing techniques is inevitably inaccurate because of the limited resolution, changing illumination properties or numerical rounding errors. For the same reasons, there is always a probability of false positive recognition, for instance, because some glare in the background resembles the same shape or features as a recognizable object.

2.4.4 *Manipulation*

Manipulation is the ability of the robot to interact physically with an object in its environment. The common manipulation tasks we focus on are picking and placing. All manipulation tasks are based on the movement capabilities of the robot. Furthermore, most manipulation tasks also need perception abilities as they rely on object recognition and feedback loops, such as if the robotic hand has successfully gripped something or not.

While manipulation tasks like picking a Lego piece from a table are considered simple by most humans, they encompass several complex stages and involve a permanent synchronization between hand and vision. Transferring such an ability on a Baxter robot raises the following challenges:

1. How to refine the pose of the object for accurate grasp planning
2. How to calculate a suitable grasp for the object



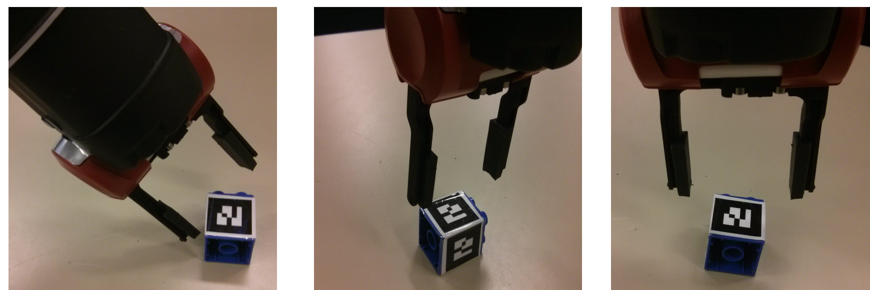
(a) Bad camera position due to angle and distance to the object

(b) Good camera position after motion of the arm

Figure 11: Changing the camera position

3. How to check if we picked the object successfully

The first challenge involves synchronization between motion and perception as the Baxter robot perceives a large part of its environment with its hands. Since a shorter distance between object and camera leads usually to a better pose estimation, the arm has to move to the newest information provided by perception. That the shorter distance leads usually to better pose estimation is evident from the parameters that influence the object recognition. Moving closer leads to a larger object in the camera frame and therefore to a better resolution of the object. A clever move can also improve the camera angle (see Figure 11).



(a) Bad approach, gripper angle does not match Lego piece orientation

(b) Bad approach, gripper targets the long side of the Lego piece

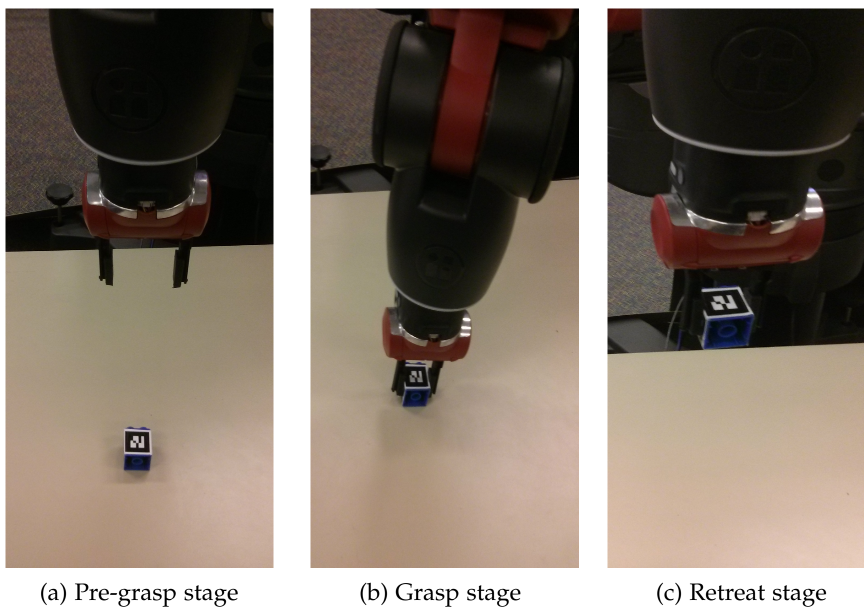
(c) Good approach

Figure 12: Different grasping approaches

As the second challenge involves grasp planning, we need to define a grasp first. The term grasp as used here refers to a complex compound motion that comprises a pre-grasp stage, a grasp stage and a post-grasp stage. We base this definition closely on the technical definition [29][section planning pipeline] to simplify later understanding for the reader. The pre-grasp stage defines the desired minimal distance between object and end effector before the grasping starts as

well as the desired approaching trajectory leading to the grasp stage. The grasp stage defines the pose the arm needs to reach right before the end effector closes, the maximal contact force between end effector and object, and the required velocity for the closing of the end effector. Finally, the post-grasp stage defines the retreat direction and distance of the end effector after the grasp stage.

While the arm executes a grasp, it traverses these three stages. The calculation of a suitable grasp therefore involves the calculation of all parameters needed in the stages. While pre-grasp minimal distance and post-grasp retreat are almost arbitrary in the Lego picking example, and therefore easy to choose, the other parameters require calculation. The grasp stage pose needs to reflect the orientation of the Lego piece and to exclude the underlying table surface from collision planning during grasp stage. The exclusion is in effect during the translation between pre-grasp and grasp stage and allows the robot to slightly touch the table. The most promising end effector poses pick the piece either from above or from the small side of the piece (see Figure 12).



(a) Pre-grasp stage (b) Grasp stage (c) Retreat stage

Figure 13: Grasp stages

Hence, depending on the grasp stage pose, approach and retreat should either happen only over the z-Axis or over a combination of the z-Axis and the Axis the small side of the Lego piece is parallel to. The end effector moves in a straight declining trajectory along the defined approach (see Figure 13). After the object is between the fingers of the end effector, closing it requires only the control of a single prismatic joint, as one finger of the parallel gripper mirrors the other.

The last stage is to check the successfulness of a pick. A pick is successful when the arm executes the complete motion and the end effector is gripping the element. Controlling this is simple because the robots end effector provides us with a sensor that reports if it is holding something or not. However, a failed attempt can result from two general reasons.

First, as perception is imperfect, the robot might estimate the pose of the object incorrectly resulting in a missed pick. Second, as the grasping is a compound move an occurrence of movement failures like missing IK solutions or non-collision free paths (see Section 2.4.1) is possible.

Each of these errors is handled differently. A missed grasp leads to a new attempt after locating the object on the table again. A failure during the grasp motion because of a non-collision free path might be solved through another try allowing the motion planner to sample other milestones or needs an adjustment of the grasp stage pose. A failure due to the IK-Solver always needs an adjustment of the grasp stage pose as the specified pose cannot be reached by the robot.

A place action is very similar to pick action. During the action the robot traverses the same stages as during a pick. We therefore spare the detailed description of the place action. However, it is possible to add additional constraints during placing as it is during picking. A typical movement constraint is, for example, to keep the placed object in a specific orientation during the whole action, like a knife facing away from a human or a glass of water always upright.

2.4.5 *Human Robot Interaction*

As said before HRI is an interdisciplinary research field that focus on the interaction between humans and robots. It encompasses several disciplines like robotics, computer science, artificial intelligence or social sciences.
The Baxter robot is, through its safety features, a promising platform for HRI research and shall be used in future projects to conduct research studies in this field. Therefore, HRI primitives need to be part of our framework. In addition, all features analyzed before are often used in the presence of humans as the typical use case of Baxter is the intersection between humans and industry robots. That being said, these features need to be human understandable to improve the acceptance of the robot among its human coworkers. A robot not fulfilling human expectations will often be perceived more dangerous by its co-workers decreasing the acceptance to work co-located to it [12]. An example for this is moving the right arm while looking left.

The basic HRI primitives considered in this project are head movement and facial expressions. The head movement of the robot is limited to horizontal movement and nodding. That is why the robot is

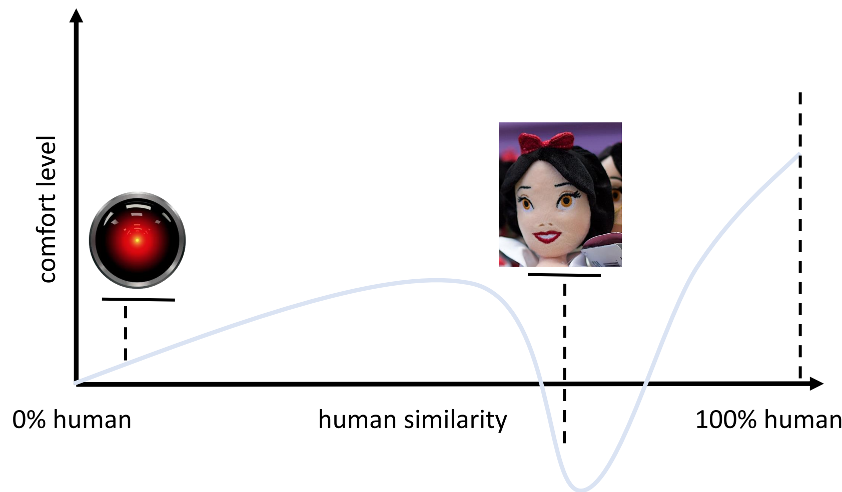


Figure 14: The uncanny valley

not capable of head gestures like tilting and or shrugging. However, nodding the head is possible expressing agreement and shaking the head is possible expressing discomfort or refusal. Furthermore, the head movement can make the arm movement more appealing for humans to watch. Letting the head face to the point where the arm is going to move signalizes a viewer what is happening next.

Facial expressions are the main nonverbal communication channel for humans. They can express many emotions like joy, excitement, stress or discomfort. How facial expressions cause the impression of a feeling in us is a heavily researched area. However, it is certain that our facial muscles and the key components of our face like eyes, mouth, nose and eyelids play a leading role to the different perceptions of facial expressions. Transferring this complex expression system to a robotic face is difficult even though promising prototypes exist [5]. However, these prototypes work with a physical 3D model of a face miming the face structure as opposite to the face of a Baxter robot that needs to be displayed on the screen. In addition, highly detailed faces have the tendency to be perceived as part of the *uncanny valley*. The uncanny valley [20] is a concept describing the paradox that more natural models of figures or faces decrease the comfort level of viewers than total artificial models of figures or faces (see Figure 14). In other words, a face that almost but not completely looks like a human face is less comfortable for viewers than a face that looks totally artificial. Such a face lies in the uncanny valley. Faces right of the valley are more comfortable although they are more artificial, faces left of the valley are more comfortable because they are close enough to be perceived as actual human faces.

As a consequence of the uncanny valley and the simple screen in contrast to a physical model of a face the usage of a high detailed face on a Baxter robot seems to be the wrong approach in our eyes.

Recent studies show that artificial faces can also convey the right emotions as long as they comprise some of the main features of a face like eyes, mouth or nose [10]. This seems intuitive as we us emoticons in our daily life to express feelings. This lead us to the decision to use a simple artificial smiley face during this project that can easily be upgraded to a more detailed face in future projects. An additional advantage is that we are able to print further information below the face giving our viewers more information about the state the robot is currently in.

2.5 FRAMEWORK DESIGN FUNDAMENTALS

As a consequence of the further usage scenarios for the Baxter robot envisioned by the project partners (see Section 1.1) we focus on the design of an *object oriented software framework* for *rapid prototyping*. We use the term *framework* as described in [11] and first defined in [9].

> ″A framework is a set of cooperating classes that make up a reusable design for a specific class of software [11].″

This definition makes clear that we mean an object oriented framework as it states explicitly the usage of classes as part of the framework. In addition, it defines that a framework is only reusable for a specific class of software. This specific class of software is in our case software that supports Rapid Prototyping for the Baxter robot.

Rapid Prototyping is a term first used in construction for describing the process of quickly fabricating a model of a physical part. These models are usually based on 3D data and are produced by sending this data, for example, to a 3D printer or a CNC machine. The advantage of such models is that they provide a direct haptic feedback and the ability to quickly and cheaply evaluate the pros and cons of the physical part. Rapid prototyping is nowadays also used as a term in software engineering to describe similar approaches that also focus on the fast evaluation of prototypes in software development. While there are whole methodologies transferring rapid prototyping to software engineering like [13], we refer to Rapid Prototyping as the ability to develop prototypes to show core functionality quickly, ignoring the methodological aspects because we use the methodology described in Section 2.2 . This means especially that these prototypes are constructed to demonstrate a main idea without solving every other problem related to it, as opposite to a commercial product having the goal to solve every corner case it might encounter. In other words, our framework provides the users with the ability to rapidly evaluate ideas without worrying about low-level details.

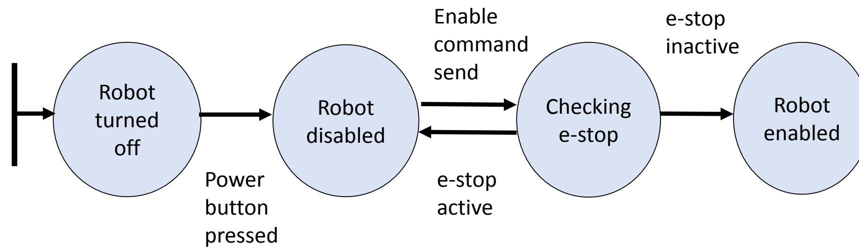


Figure 15: An example for a State Machine

Since the framework shall be the base for Rapid Prototyping approaches in future projects, it ideally encapsulates all low-level functionality into a good API. A good API is hereby characterized by the following items [4]:

- Easy to learn
- Easy to use, even without documentation
- Easy to read and maintain code that uses it
- Hard to misuse
- Easy to extend
- Appropriate for the audience
- Sufficient powerful to satisfy requirements

Each of these items is equally important. We address the first three items with a clean, defined *State Machine* concept our API is based upon.

A State Machine is a mathematical model to describe a sequential logical process [14]. It consists of a finite amount of states and transitions between these states. Transitions are connections between two states that are triggered by events. The State Machine can only be in one state at a time this state is called current state. Whenever a transition event for the current state is reached, the machine changes to the state defined in the corresponding transition (see Figure 15).

This concept allows us to define the functionality provided by the framework as a simple sequential process. The framework provides different states like a grasping state or a searching state. In addition, it provides a mechanism to introduce transitions between states (maximum two). This allows us, for example, to have states with or without error condition and states that end the computation because they do not have a transition.

The trigger for a transition in the framework is state dependent. An object recognition state, for example, triggers a transition after the recognition while a pick state triggers a transition after an executed grasp. The concrete transition that is chosen by the trigger depends on the state (successful vs. failed grasp transition).

Having this simple State Machine based concept as the base of the framework decreases the learning effort for the user, as the user just needs to know the different implemented states and developers are normally familiar with State Machines in general.

The framework is also designed around the idea that a misuse results in direct feedback to the user. It relies programmatically on asserts and exceptions that express the misuse in a clear and meaningful error message. The user knows when she or he violated a contract of a method and how to fix this violation.

Besides of that, it shows internal exceptional state, like a missed grasp because of inaccurate perception on the face of the robot (see Section 2.4.5), so that a user can see this exceptions during the tests of a prototype. However, this is not only relevant for the user but also for the person who watches a demo as it makes the behavior of the robot easier to understand for humans.

The extensibility of the framework is directly inferable from the basic State Machine concept. It is easy for a user to create another state that either encapsulates many existing states to a higher level state or to create a complete new state. The framework offers here object oriented extension mechanisms like abstract classes to work with, defining the interface needed to integrate the new state into the existing framework. In addition to this state extension some state offer a more lightweight extension mechanism through the altering of their internal behavior. So it is for example possible to alter a grasping state from a simple block grasp to a more sophisticated grasp without introducing a new state.

Testing if a framework is appropriate for a user and powerful enough for the requirements of users is something that needs the introduction of the framework into daily development processes. While we believe that the framework is a good choice for our target audience, time is needed to evaluate this assumption. Nevertheless, the framework is extensible and simple, so that adding functionality for new requirements should be easy manageable.

3 FOUNDATIONS

In this chapter we introduce the general concepts and tools used in our work. This provides the background needed to understand the details of our implementation. This details are independent from the project itself and often relate to the Robot Operating System or a certain library of it.

3.1 ROBOT OPERATION SYSTEM

The Robot Operating System (ROS) is an open source meta operating system based on Linux that has the goal of making the work with robots easier through a better way of sharing and collaborating during robotic research. To accomplish this goal ROS provides some important features to simplify the work with a robot [23].

- It provides a hardware abstraction layer for the robot
- It provides core functionality like algorithms for movement and perception most robots need
- It provides a communication infrastructure for the different programs that are running on a robot
- It provides tools like visualization and simulation that simplify the development process
- It provides a way of easy extending the frameworks capabilities

It is licensed under the BSD license and is used by various companies and researchers on different robots around the world.

It is important for the reader to have an idea of the basic concepts of ROS, to understand the main points of this thesis. For that purpose we provide here a short overview. For a more complete documentation please look at `http://wiki.ros.org/`.

Conceptually, ROS is a distributed system that is running on the robot. This design leads to multiple programs running on the robot at every point in time. That means that on a Baxter robot the programs for capturing the IR-Sensor data, the program that reads the different joint values and many other programs getting started and run in parallel.

To communicate between each other these programs use the network (Transmission Control Protocol (TCP)/Internet Protocol (IP)) and exchange data structures called Messages. These data structures are

much like the structures in the C programming language and can encompass other structures or primitive data types like Integers, Floats or Booleans. As C structures, these Messages are also extensible through custom Message types what leads to a flexible interchange format. ROS offers three kinds of communication techniques based on these Messages.

First, there is a many to many communication trough programs called Nodes[1] (Figure 16). Nodes work as Publisher or Subscriber. They communicate over a message queue called Topic. A Topic is a message queue with an address in the form
/[namespace1]/[namespace1]/[name] where namespaces are optional. The Publisher sends a message to a certain Topic and the Subscriber can subscribe to that Topic and read the Messages published. A Node is not bound to be either a Publisher or Subscriber and can act as both. Also multiple Nodes can publish or subscribe to the same Topic. A Topic message queue is registered in the system when the first Node publishes to that Topic and gets unregistered when the last publisher Node is shutdown. Furthermore, a Topic can be remapped during the start of a Node to another name. This is useful when the Topic's name is already used for something else on the robot.
The most common use case for Nodes is publishing or subscribing to sensor data or processed data.

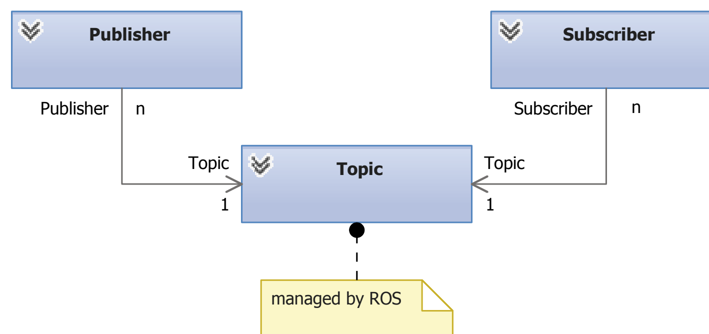


Figure 16: Node Communication

Second, there is a request-reply communication trough Services and Clients[2] (Figure 17). In this model, a Client sends a request Message directly to the Service and waits until it receives a reply. Vice versa the Service sends the matching reply Message directly to the Client. Because of the direct connection this is ideal for a one to one communication as it is faster and saves resources compared to message queues. A typical use for a Service is a short taking request to

1 http://wiki.ros.org/Nodes
2 http://wiki.ros.org/Services

another program, for example to test if the robotic hand is currently gripping.

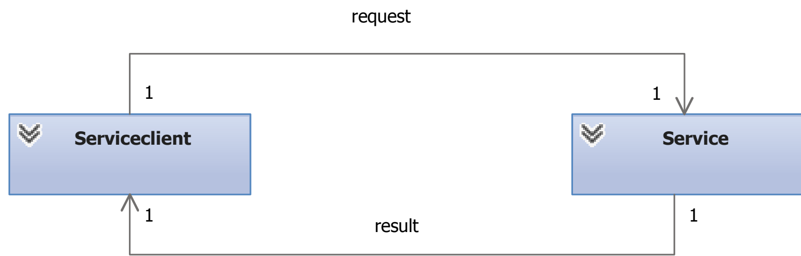


Figure 17: Client Service Communication

Last, we have asynchronous request-reply communication based on the actionlib[3] interface (Figure 18). This is a specialization of service-based communication and deals with the disadvantage of the waiting client. In contrast to a normal service call, the Actionclient does not wait for the reply and can continue its work. The Actionservice notifies the Actionclient when the action is done and also provides it with feedback during execution. In addition, the Actionclient can cancel the running action any time. This makes the actionlib interface useful whenever we have to execute a long taking task like moving an arm or manipulating an object in contrast to the blocking Service interface.

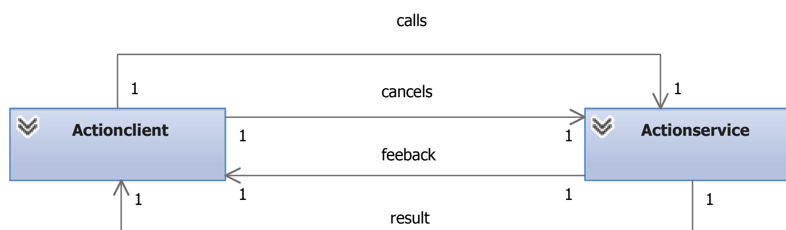


Figure 18: Actionlib Communication

Besides of the conceptual view, ROS uses different mechanisms to describe the nature of the robot it is currently working with.

To begin with, the Universal Robot Description Format (URDF) specifies the robots physical appearance and its kinematic properties. It comprises joints, links and collision geometries and the connection between these. In addition, it defines the coordinate origin for each of the links relative to its parent link. It is the most important file for

3 http://wiki.ros.org/actionlib

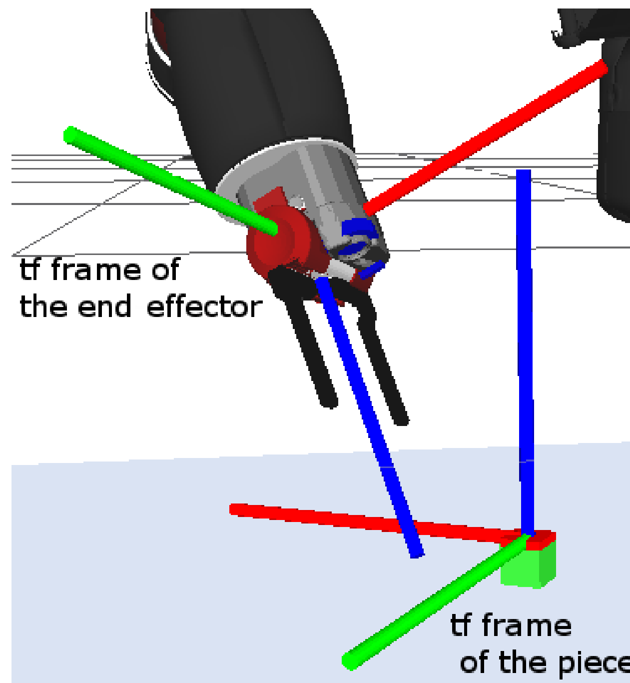


Figure 19: Two different tf frames of the robot. The position of the object can be specified relative to the end effector's frame.

any kind of movement executed on the robot, as all ROS dependent APIs rely on its information.

Furthermore, ROS uses different coordinate systems called transformation (tf) frames. This concepts makes spatial planning often easier because it allows to compare poses relative to a certain tf frame. This allows, for example, to compare the orientation of an object on a table relative to the orientation of the end effector telling us how to rotate the end effector to grasp the object (see Figure 19).

Note, that the position of the end effector can change. The origin of the tf end effector frame relative to the tf world frame is therefore time dependent. In other words, the distance and orientation between the tf world and the tf end effector frame can change in every time step, invalidating the old position. This is why an accurate clock synchronization between robot and controlling host is needed as the evaluation of the tf data is done on the controlling computer. A difference between computer and robot clock results in tf frames seeming too old for the controlling computer causing it to ignore the frame. Finally, to convert one tf frame to another ROS offers a tf Package that includes fast reliable conversions mechanisms usable through the Python and C++ API. These mechanisms provide for a given pose in a source frame any representation in another frame.

An important point is that using Baxter together with ROS is that the Research Software Development Kit (RSDK) is nothing else than a collection of different Nodes, Services and Actionservices provided

by Rethink Robotics. Functions included in these Nodes are, for example, the configuration of cameras or simple Nodes for controlling movement.

Viewing it from an administrative view ROS is a collection of tools to run programs on a robot. This means especially that every ROS program is either a C++ or Python program compiled with GNU Compiler Collection (gcc) or interpreted by a normal Python runtime. To run this programs on the robot, we invoke the rosrun command that loads our program to the robot and executes it. As we might want to start many programs in parallel ROS offers a XML dialect called Launch files that allows us to write a start script executing all these programs in the right order with the right parameters. We have apart from these fundamental tools, various other tools for debugging and maintaining our ROS programs. To mention is here *RVIZ* as it is used on a daily basis for robotic simulation. It allows to explore the world as the robot perceives it (see Figure 20). We used it during our work to debug or simulate scenarios or to get an overview about the various states of the robotic hardware.

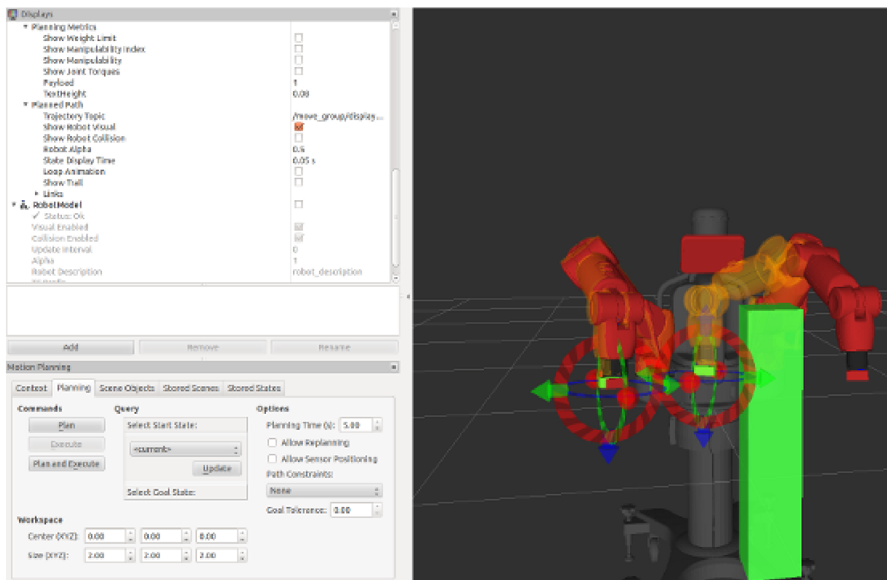


Figure 20: Simulated robotic environment

Finally, we would like to emphasize that ROS programming always needs to take into account the performance and concurrency of the processes running on the robot. We always have to keep in mind that ROS is a parallel system that has many components that need to run almost in real time. It not only contains parallelism on the Node level but also on the thread level. Every ROS program comes along with multiple threads for callbacks, helper tasks and related things. Therefore, the programmer has always to consider the various synchronization and speed factors that are needed to get the Node work

fast and correctly. This includes many parallel data structures like for concurrent lists or lock free queues and optimization techniques like for example move constructors or copy and swap in C++.

3.2 AR TRACK ALVAR

AR Track Alvar is an open source library that provides fast object recognition through fiducials [24]. Fiducials or markers are print outs looking much like a QR-Code. They have a special structure that makes it easy for the common object recognition algorithms (see Section 2.4.3) to find them inside an image or camera stream.

First, they have a clear border that detaches the marker from the outside world. Second, inside this border there is a unique pattern that is easily recognizable by the image recognition algorithms as it provides clear edges. Third, the *AR Track Alvar* library knows the size of the marker helping it to transform the 2D image coordinates to 3D world coordinates.

That is why, if there is a Marker on the camera image, AR Track Alvar provides us with the pose of the marker relative to the cameras tf frame. However, it has some restrictions that apply to the marker size, to the environment the marker is used in and to the angle of the camera. The library automatically discards markers that seem too large or too small in the image. As we can control the size of the marker in the image we need to pay attention to not move the arm cameras to close or to far away from the marker. In addition, the library only works well when the border of the marker is clearly recognizable. As the marker recognition library converts the image in the first step to a bitonal (black and white only) image, the borders need to have high contrast difference to the background. Typical ways to control this is to create a white background for each marker and to control the illumination conditions.

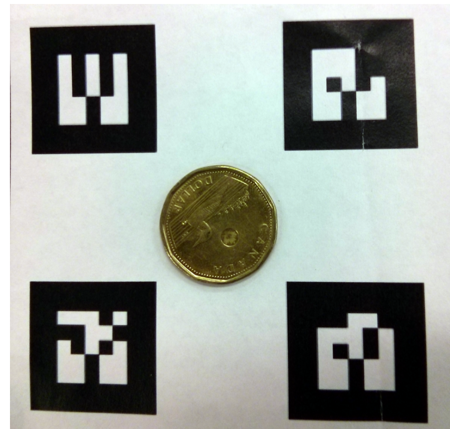


Figure 21: Four fiducials with a one dollar coin as size reference

Finally, the angle of the camera is important for the pose estimation. As the pattern is only 2-dimensional, the pose estimation works best when the camera has a 45 degree angle to the marker. Having a 90 degree angle works also well for translation although it is less accurate for the orientation [24][page 46].

3.3 MOVEIT!

MoveIt! is the standard manipulation framework in ROS [29]. It is developed by the Willow Garage team as an industry version (called ROS-industrial) and as an Open Source version. The Open Source version encompasses many different ROS Packages including the following functionalities:

- Motion planning adaptable for many robots
- Flexible environment representation
- Manipulation primitives adaptable for many robots

The adaptability of the library is based on different assumptions. First, the position of every joint in the robot needs to be known and published to the */joint_states* Topic. Second, the tf frame of every link needs to be published into the tf tree of the robot. Third, all collision geometries of the robot need to be specified in the URDF. Once a robot meets this assumptions, the library can be configured for the robot.

Each robot configuration is a ROS Package including a *config* folder with a Semantic Robot Description Format (SRDF). The SRDF is the MoveIt!'s main config file, grouping joints and an end effector together into planning groups for motion planning and defining the collision matrix of the robot. Besides of the SRDF MoveIt! encompasses other config files, for example, specifying the physical constraints of the robot (*joint_limit.yaml*) or the properties of the IK-Solver (*kinematics.yaml*).

Motion planning is based on an IK-Solver, the Open Motion Planning Library (OMPL) [30] and the Flexible Collision Library (FCL) [22]. While the IK-Solver is either a general IK-Solver [28] or a robot specific IK-Solver, OMPL and FCL are fixed components used for the motion planning algorithms.

The OMPL contains several state of the art sample-based motion planning algorithms. Depending on robot, planning constraints and planning environment each algorithm performs different. A typical distinction can be made between graph-based and tree-based planners. Tree based planners are due to their reduced overhead better for planning in changing environments where the graph would be soon invalid, while graph based planners are better for planning many plans in a fixed environment reusing the same graph.

The FCL provides the collision checking interface for the OMPL. It handles self-collisions as well as collisions with the environment. Every motion plan created by MoveIt! is calculated by both libraries together.

The environment is organized internally by MoveIt!. However, we can easily alter it by publishing messages to the */collision_object* topic. Through that, we can add, update and remove objects in the environment of the motion planner without thinking about the internal

structure. The published objects are considered during motion planning and visualized in RVIZ.

MoveIt! defines pick and place primitives. This means, picking and placing of objects is supported assuming that the robot exposes the needed information and, in the case of picking, a grasp Message is provided. The grasp Message encompasses the same information as described in Section 2.4.4 and needs to be calculated manually by the user of the pick and place functionality.

3.4 DESIGN PATTERNS

A common approach to write robust and flexible software is the use of Design Patterns. A Design Pattern is a well-known solution to a standard problem. However, not every solution to a well-known problem is directly a Design Pattern. To become a Design Pattern the solution needs to be formalized and needs to have the essential elements as described by [11]:

1. The *pattern name* describes the problem
2. The *problem* statement states where to apply the pattern
3. The *solution* describes the design, relationship and the collaborations of the pattern
4. The *consequences* describe the pros and cons of the pattern

The solution can be expressed differently based on the application domain of the Design Pattern. Well-known Design Patterns exist for example for the parallel design application domain or for the object oriented design domain. We use in this work only *object oriented design patterns* formalized by the use of a Unified Modeling Language (UML) class diagram that describes the different classes of the Pattern as well as their communication.

The common Design Patterns that we know today in object oriented design are already 20 years old and were made popular by the famous book *Design Patterns*. With the introduction of these Patterns into software systems over the last 20 years software become more stable and easier to understand because they did not only solve the specific problem they were designed to, but also became common vocabulary for developers to use. This means, that most software engineers know how a specific component of a system works just by hearing the name of the Pattern the component is designed upon. This is especially useful in a framework that needs to be easily understandable and extensible.

In consequence of this advantages we designed our core system around several Design Patterns that are described in this chapter.

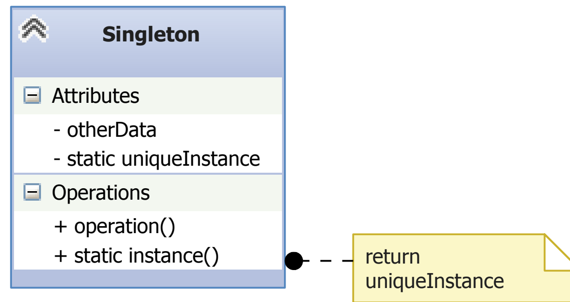


Figure 22: Singleton Pattern

The first Pattern we use is the *Singleton* Pattern (see Figure 22). The idea behind the *Singleton* Pattern is to have an object that we can only instantiate once. This leads to the following advantages:

- The object is easy accessible
- The object only occupies memory for one instance
- The object can be specialized through polymorphism

While some argue that the *Singleton* is nothing more than a global variable, it has some improvements over a global variable. First, we are able to introduce access restrictions into a *Singleton* easily. Second, the instantiation and destruction is clearly defined and we are certain to always receive a probably instantiated variable. Third, it is easier to refactor a system that uses a *Singleton*, when more objects of the same instance are needed compared to refactoring a system that uses a global variable. This is a result of the clear access pattern a Singleton defines.

A typical example for the usage of such a Pattern is the control of an input device such as a keyboard as they usually is only one input device connected to a computer.

The next pattern we use is the Strategy pattern (see Figure 23). The idea behind this Pattern is to define interchangeable algorithms during runtime.
It offers the following advantages:

- We can define a group of algorithms that are interchangeable
- We can encapsulate each algorithm of this group
- We have an increase in flexibility through the different algorithms in the groups

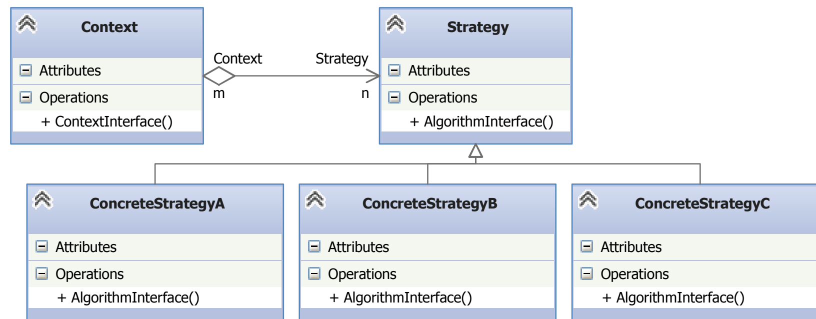


Figure 23: Strategy Pattern

A good example for the use of the Pattern is the use of different decompression algorithms in an unpacking program. While the representation of the result remains the same, we can define different strategies to decompress *zip*, *tar* or *rar* files.

Another Pattern used is the *Template Method* Pattern (see Figure 24). This Pattern is used to defer some stages of an algorithm to subclasses reducing the amount of implementation for similar algorithms. It has the following advantages:

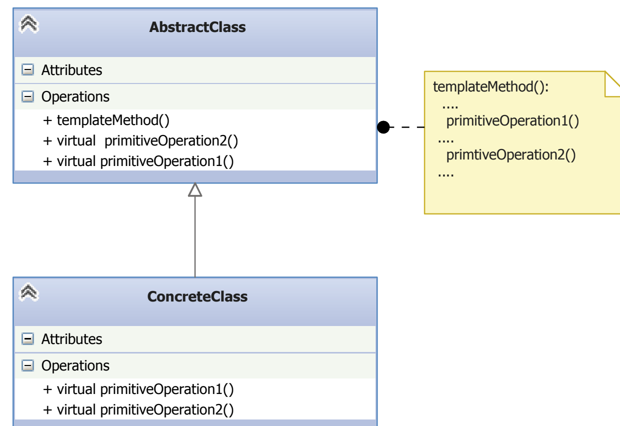


Figure 24: Template Method Pattern

- reduces the amount of code needed to implement familiar algorithms
- allows the definition of a skeleton of an algorithm that is can be further specialized
- decouples algorithm interface and concrete implementation

An example scenario uses the Pattern to define a generic transformation of a file. In such a case, the base class defines a concrete

method *transform(file)* and abstract methods for reading, transforming and writing the file. The *transform* method comprises then calls to reading, transforming and writing without knowing about the concrete implementation. As a result, different transformation algorithms can be used with the same interface.

3.5 readers writers problem

Table 1: Boost synchronization primitives

primitive	description
boost::shared_mutex	A mutex that is usable for multiple readers and a single writer. It lets the Operating System decide who acquires the lock next and is therefore not vulnerable to reader or writer starvation.
boost::shared_lock<Mutex>	A lock that blocks only when it is not possible to receive shared access from the Mutex. Otherwise it allows multiple processes access.
boost::unique_lock<Mutex>	A lock that blocks until it gets exclusive ownership of the Mutex allowing the lock holder to be certain to operate exclusively in the critical section.

In Computer Science the *Readers Writers Problem* describes a synchronization problem in concurrent programs. In this problem, a data structure is accessed by readers and writers. However, a simultaneous access of a reader R and a writer W can lead to a corruption of the data read by the reader. If for example R counts the elements in a vector and W deletes the first element after R has already passed it R's data is corrupt. Solving this problem with a primitive Mutex that locks the critical section seems straight forward but leads to a performance problem (*first Readers Writers Problem*). Now each reader has to enter the section sequentially although multiple reader in the critical section are not a problem [31][page 215-217].

Addressing this performance issue by allowing multiple readers in the critical section does however rise another problem. In this case, a writer might never enter the critical section and starves because the lock is always hold by one of the readers (*second Readers Writers*

Problem). Therefore an optimal solution to this problem requires the following conditions:

- Read and write actions are performed sequentially
- Multiple reader can enter the critical section at once
- The writer waits no longer than absolutely necessary

To achieve this condition we use the boost locking primitives in our project (see Table 1).

These primitives have the advantage to operate on a higher level than normal operating system synchronization mechanisms do and allow therefore a simpler approach to the problem. Furthermore, the *boost* synchronization is used in thousands of products and is well tested compared to a custom implementation.

Regarding the *Readers Writers Problem* the *boost::shared_mutex* is now used to establish a fair locking mechanism. Every method that reads data from the data structure acquires a *boost:shared_lock* allowing multiple readers. Every method that writes data to the structure acquires a *boost:unique_lock* to get exclusive writers access to the data (see Listing 1).

Listing 1: Solving the *Readers Writers Problem* with boost primitives

```
boost::shared_mutex m;
void reader()
{
  // get shared access
  boost::shared_lock<boost::shared_mutex> lock(m);
  //do Stuff below
  ....
  //locks get automatically freed through their destructor
}

void writer()
{
  // get exclusive access
  boost::upgrade_to_unique_lock<boost::shared_mutex> uniqueLock(m
      );
  //do Stuff below
  ....
  //locks get automatically freed through their destructor
}
```

3.6 Construction vs. Usage

A concept that we use in our design process is that we separate the system construction from the usage as described by Robert C. Martin in [18][page 154].

The idea is to remove the dependency from high level components on concrete low-level components by introducing indirection into the instantiation process. This is useful as the high level component is no longer responsible for the instantiation of its low-level components enabling us to replace the used low-level components without changing the high level component itself. The only thing we have to consider is that our replacement satisfies the same interface as before.

Another advantage is that we remove the wiring code out of the business logic what leads to a more compact and easier to understand application code. A typical implementation of this idea are the various dependency injection frameworks like Spring or Guice.

To demonstrate this idea and its advantages consider Listing 2.

Listing 2: Introducing tight coupling through a violation of *Construction vs. Usage*

```
class GraspState: public State{

  //other methods
  ....

State getNextState(){
        //constructing a new state
        PlaceState state(object, environment,arm);
        return state;
}

}
```

This is a violation of the Construction vs. Usage principle as the *PlaceState* is constructed in the *GraspState*. The disadvantage is that the *GraspState* is now directly dependent on the *PlaceState* class although the *getNextState()* method returns a parent *State* object. While might after grasping follows picking in the current application this design does not allow later changes of the state returned by *getNextState()*. In addition, we introduce not only dependencies to the *PlaceState* but also to all its constructor arguments.

While this is a relative easy example to illustrate the principle many applications have such instantiation dependencies in some methods introducing unnecessary dependencies.

To not create such dependencies all our designed ROS Nodes follow a strict separation of construction and usage. Therefore, all instantiation and object wiring is done in the main method of a Node. After-

wards the actual program logic is invoked by calling a method on an object of the now wired object graph.

4 SYSTEM ARCHITECTURE

The system architecture is based on the framework design fundamentals described in Section 2.5. As already discussed, the key concept behind the framework is a State Machine. However, transferring this abstract concept to a technical ROS design requires more effort, since the State Machine relies on different low-level features as described in Section 2.4.

Consequently, these low-level features need to be designed in such a way that they integrate well into the State Machine and the framework. Hence, our system architecture comprises several ROS Packages defining the lower level features and a separate ROS Package for the State Machine itself (see Figure 25). Dividing the features into distinct ROS Packages leads to clear benefits in the overall design because monothematic Packages are easier to maintain and extend due to the clear boundaries on the interface level and the associated reduced complexity [23]. Similar approaches for complexity reduction exist for example in Java through the Package system and in Python through the module system. In addition, distinct ROS Packages ensure that using the low-level functionality separately is possible.

The following sections describe the different ROS Packages, their design and their usage. The final section describes the State Machine Package and illustrates the integration of the low-level functionalities into the Package. Note that the UML diagrams use the following color code: yellow is a public framework class, blue is an internal class and green is a class useful for extending the current framework's capabilities.

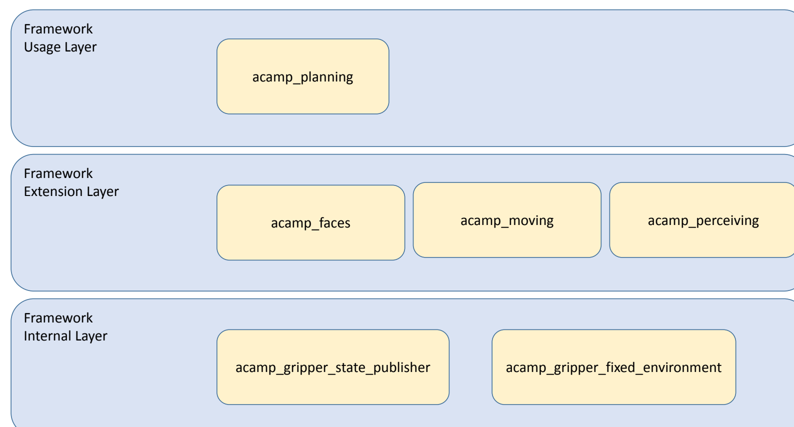


Figure 25: Framework structure

4.1 ACAMP_MOVING

The acamp_moving Package comprises every functionality relating to the movement of the arms. It encompasses one ROS Node and a complete interface for basic collision free operations with the arms of the robot. However, it does not include manipulation functionalities like picking and placing because they rely on several other things such as perceiving and planning (see Section 2.4.4).

The Package is designed around the MoveIt! library (see Section 3.3). We have chosen MoveIt! because it is the standard motion planning library for ROS including well tested and up to date algorithms. In contrast, other motion planning libraries evaluated were either outdated (e.g. *ROS arm navigation*) or missed important features. For example, Rethink Robotics motion planning API lacks the ability for collision aware motion planning, even though their IK-Solver was faster during our benchmarks.

Due to this dependency, our motion planning API is a direct extension of the motion planning API of MoveIt!. This has the advantage of a single well defined class (*BaxterMoveGroup*) that contains all methods for doing movement. On the other hand, such an extension leads to a stronger coupling between the library and the developed framework. Although, this can be a problem when the library becomes deprecated, we accepted this trade-off because MoveIt! is the standard library for movement in ROS directly maintained by ROS core developers.

Our *BaxterMoveGroup* class includes additional methods that configure MoveIt! for our needs. For example, The constructor specifies the expected goal tolerances during a motion of the Baxter robot. Furthermore, our class includes a direct connection to the *joint_trajectory_action_service* of the arm, allowing us to execute lower level functionality like cancelling a running motion plan. To conform to MoveIt!'s API, the newly introduced methods apply the typical error handling through the *MoveItErrorCode* class. An error code matches exactly one error cause giving a detailed feedback.

Also contained in this Package is the grasp planning interface that is used for manipulation. The grasp planning classes in this Package are purely dependent on movement and assume a perfect perception to keep the Package boundaries clear.

The purpose of the grasp planning interface is to simplify the general grasp planning procedure and to have some functionality that returns basic block grasps. Since the grasping message contains many fields, the grasping interface (see Figure 26) works with a *template method* pattern. The base class fills the fields, that are independent from the target object but dependent on the robot, and the inherited class populates the remaining fields depending on the target ob-

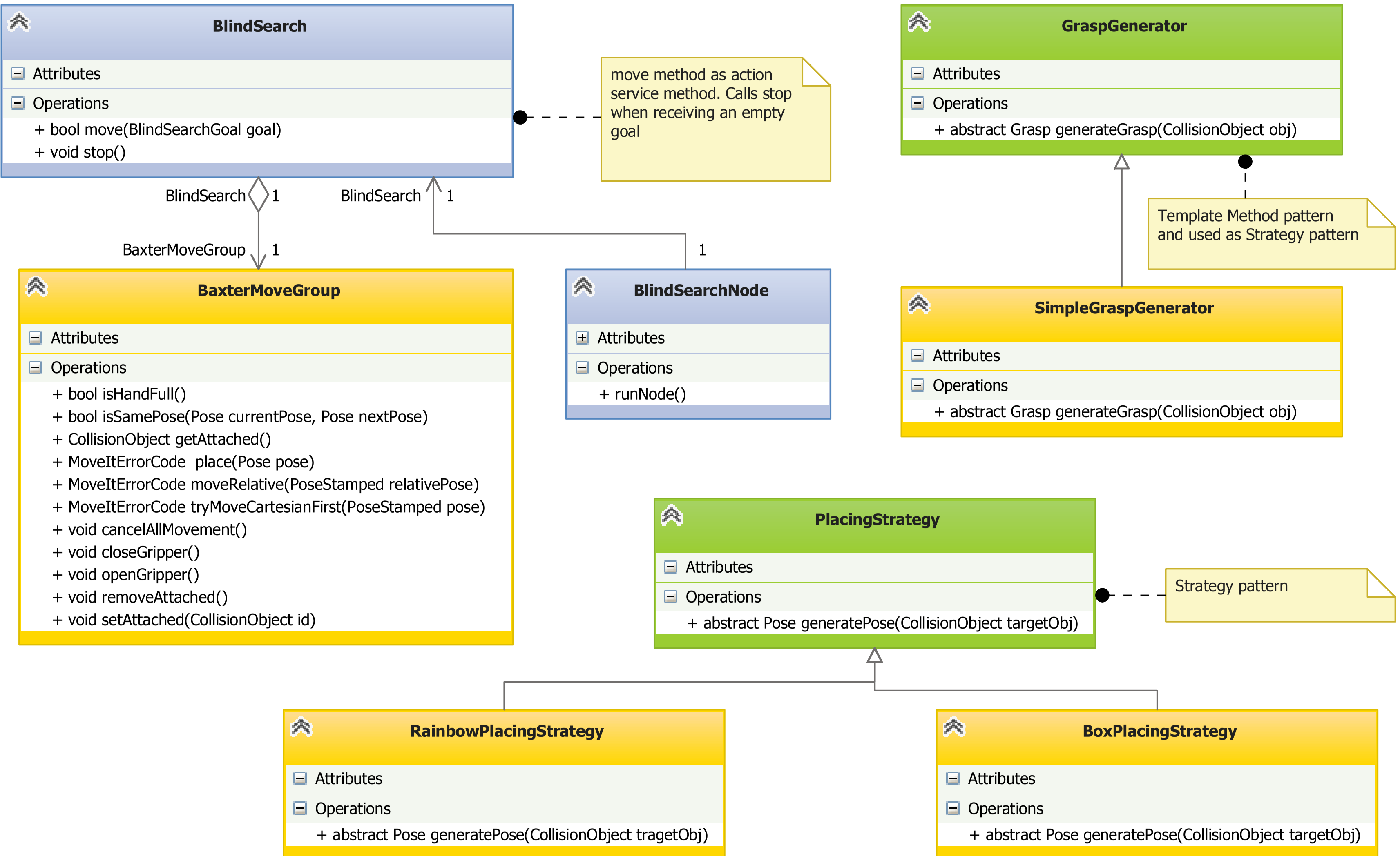


Figure 26: acamp_moving Package class structure

Listing 3: The *BlindSearchGoal* message that defines the searching rectangle

```
# the lower right corner of the rectangle
geometry_msgs/Pose startPose
std_msgs/Float32 width # in meter
std_msgs/Float32 height # in meter
```

ject. So, the grasp message is adaptable for many different geometries without the urge to calculate redundant fields again.

Finally, the Package also encompasses a Node (*blind_search_node*) that moves the cameras around for searching tasks. The idea behind this Node is to cover every visible point of a rectangle with the cameras while another Node checks the camera stream for recognizable objects. For that reason, the Node exposes an Actionservice (see Section 3.1) that expects a request containing the rectangle to cover (see Listing 3).

After receiving such a request, the Node searches depending on the start mode with one or both arms trough the rectangle. To reduce the collision potential of the arms (see Section 2.4.2), each arm monitors a distinct area during multi-arm search.

To stop the search, another request to the Actionservice is sufficient (see Figure 26).

4.2 ACAMP_PERCEIVING

The *acamp_perceiving* Package encompasses the functionalities to transform raw sensor data to an object in the world. Furthermore, it provides an interface to query for specific objects in the robot's environment representation.

We build our perceiving Package around the AR Track Alvar library (see Section 3.2). While other ways exist to do object recognition in ROS, we have chosen this library because of the fiducials. Compared to normal object recognition, fiducials are well suited for Rapid Prototyping approaches because they work without training or adaptation for every object containing a marker.

The perceiving is based on two instances of the *ar_track_alvar* Nodes with remapped Topics for each arm. The remapping is necessary as the Nodes otherwise publish to the same Topic, making it impossible to distinguish between the cameras that report the marker.

To process the raw data coming from the marker recognition, our perceiving Node passes several steps:

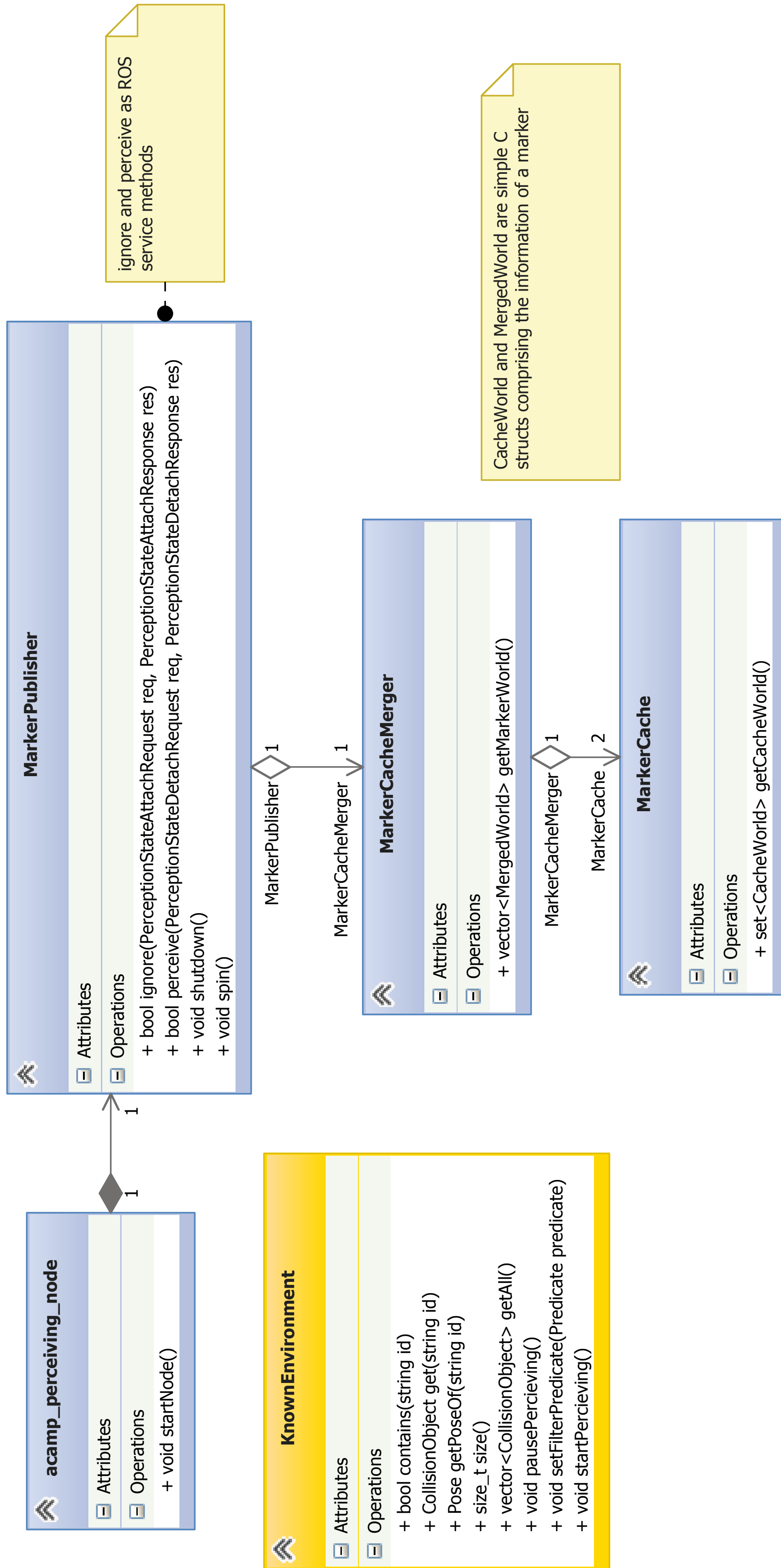


Figure 27: acamp_perceiving Package class structure

1. Transform the marker pose from the camera's frame to the world frame
2. Apply the filter to the marker data
3. Decide if the marker is valid or if it is a false positive
4. Remove markers if they are outdated
5. Merge the data streams from both hands to one valid data stream
6. Publish a collision object for each marker

We described the steps of this process in detail in Section 5.5. However, the result of this process is that a marker is only accepted if it meets the following criteria:

- It is seen in > 5 consecutive sensor packets, to prevent false positives
- once contained in one of the last 10 sensor packets, to remove unseen markers
- It is seen at maximum $10\,\text{cm}$ over the table, to prevent false positives and the publishing of already attached markers
- It is seen at a maximum changed distance of 30cm in the last 50 sensor packets , to prevent false positives

While these steps are sufficient to detect markers and to drop false positives, another important need is to pause perception. This need arises because of a bug in the MoveIt! library during grasping. When we grasp some object in the world, the framework removes this object from the environment automatically and attaches it to the robot's end effector. However, as the cameras of our robot are next to the end effector, the camera sees sometimes the already attached object. Without stopping the perception, we would now publish two objects: one attached and one in the environment. This would result in a collision error between the attached and the unattached objects causing a failure in the MoveIt! grasping pipeline. Therefore, the *acamp_perciving_node* offers two Service calls (see Section 3.1) to start and pause perceiving (see Figure 27).

As a result of the process above, the perceiving Node can sense markers and publish them as objects to the environment. However, the environment is internally managed by MoveIt! without a predefined API. However, many states of the State Machine rely on some information provided by the environment. Picking, for example, is only possible if we know the pose of the target object in the environment. As a consequence, this Package provides a simple interface

(*KnownEnvironment* class) for querying information about the environment. It provides, for example, methods to list the objects of the world, to filter the world objects by generic criteria and to calculate distances between objects. Filtering is especially important as it is often necessary to exclude some objects from the result list that is processed. An algorithm that tries to count objects on a table needs to filter out the table itself. As this filtering can be very complex, the API provides a generic way of filtering through predicates. Predicates can be just Boolean functions that satisfy the specified interface, but they can also be functors. A functor is an object that can be called as a function [1][page 126]. This allows the filters to decide, based on the internal state of the functor object what is, for example, helpful if one is interested only in the ten nearest objects. In this example, the functor would internally count how often it was already called and return *false* after the tenth call. Furthermore, as this class receives its data from the environment and writes it to the different clients, it contains a Reader Writer Problem. Therefore, the methods are all synchronized by read and write locks according to the boost solution (see Section 3.5).

4.3 ACAMP_FACES

This Package contains an interface for HRI primitives like head movement and displaying different faces. The design is oriented on the typical design of logging frameworks (see Figure 28). Although logging and the control of the head seem very different in the first place, they share the same conceptual attributes. First, both interfaces need to be accessible from any part of the program. Second, both interfaces control the access to a central unique resource (logging file or head of the robot). Third, both interfaces produce output that is only needed for humans, and their invocations are therefore spread through many methods.

Due to these insights, the design of the interface (*FaceLogger* class) is based on a *Singleton* pattern like logging frameworks are. This allows easy access to the interface from every location in the code without most of the disadvantages of a global variable (see Section 3.4).

In addition, an easy to use interface is required for logging-like methods due to their spreading throughout the code. Setting up such a method invocation by instantiating objects and calling other methods before leads to a clumsy interface. Therefore the invocation of the interface should be need exactly one line of code. On the other side, the *FaceLogger* class has to publish an image to the screen what would normally require the instantiation of the image first. A naive workaround for this issue involves a method in the interface instantiating the image that is published during the call. However, this would create a tight coupling between the images used and the *FaceLogger*

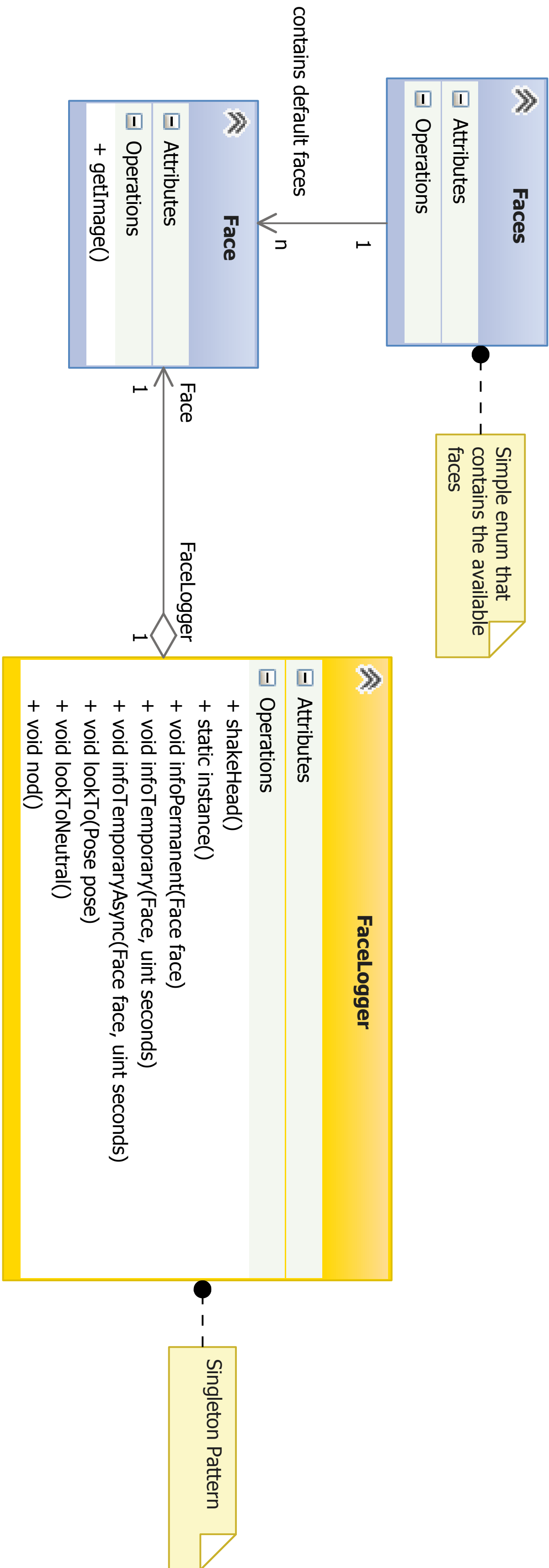


Figure 28: acamp_faces Package class structure

class denying a user to publish another face image. Thus, our approach is based on one single method to publish a face in the *FaceLogger* class in combination with a collection that provides direct access to all available standard faces. Consequently, it exist a direct access pattern for the provided faces that allows one to invoke the method in one line while the user has still the ability to publish its own faces if needed. This approach is again similar to the handling of logging levels in modern frameworks.

Finally, the *FaceLogger* class provides simple methods for head movement.

4.4 ACAMP_PLANNING

This Package contains the implementation of the State Machine concept together with concrete states implemented and the abstraction mechanisms allowing the creation of new states (see Figure 29). The State Machine itself is designed around an encapsulating *StateMachine* class. This class represents a whole State Machine that can be executed and returns after it is in a final state. The advantage of the encapsulation of a State Machine in an object rather than using the beginning state directly is that it allows one to change the runtime conditions of a complete State Machine at once. It enables a multi-threaded approach through the running of one *StateMachine* object on another thread. In addition, it simplifies the use of a State Machine, as the user only has to operate with one object instead of all the states. The State Machine execution is simply triggered by calling the *run* method of the class.

Although encapsulation of the State Machine in a class is beneficial, a naive construction approach of such an object would lead to a large amount of initialization code as every state needs to be initialized before the connection between the states happens to not violate the *Construction vs. Usage* principle. Therefore a builder class (*StateMachineBuilder*) is introduced to create a *StateMachine* object with as few lines as possible. Note, that this builder class does not represent the classical builder Design Pattern as it does not comprise the abstract builder types. It is a representation of the typical *StringBuilder* pattern as found in C++ or Java, exposing an easy construction mechanism by method chaining.

The following list comprises the currently implemented states:

- Exploring a table (search_state)
- Focusing an object on a table (focus_state)
- Picking an object (grasp_state)
- Placing an object (place_state)
- Deciding about the arm to use to work with an object (decide_arm_state)
- Leaving a shared area in dual arm operation (leave_shared_area_state)

The *search_state* is responsible for controlling the table exploration and is the connector between the *blind_search_node* and the *perceiving_node* (see Figure 30). The *handle* method of the state calls the *blind_search* Actionservice and executes a search as long as nothing is in the environment. This is necessary as it allows parallel execution of motion through the *blind_search_node* while the state itself is able to

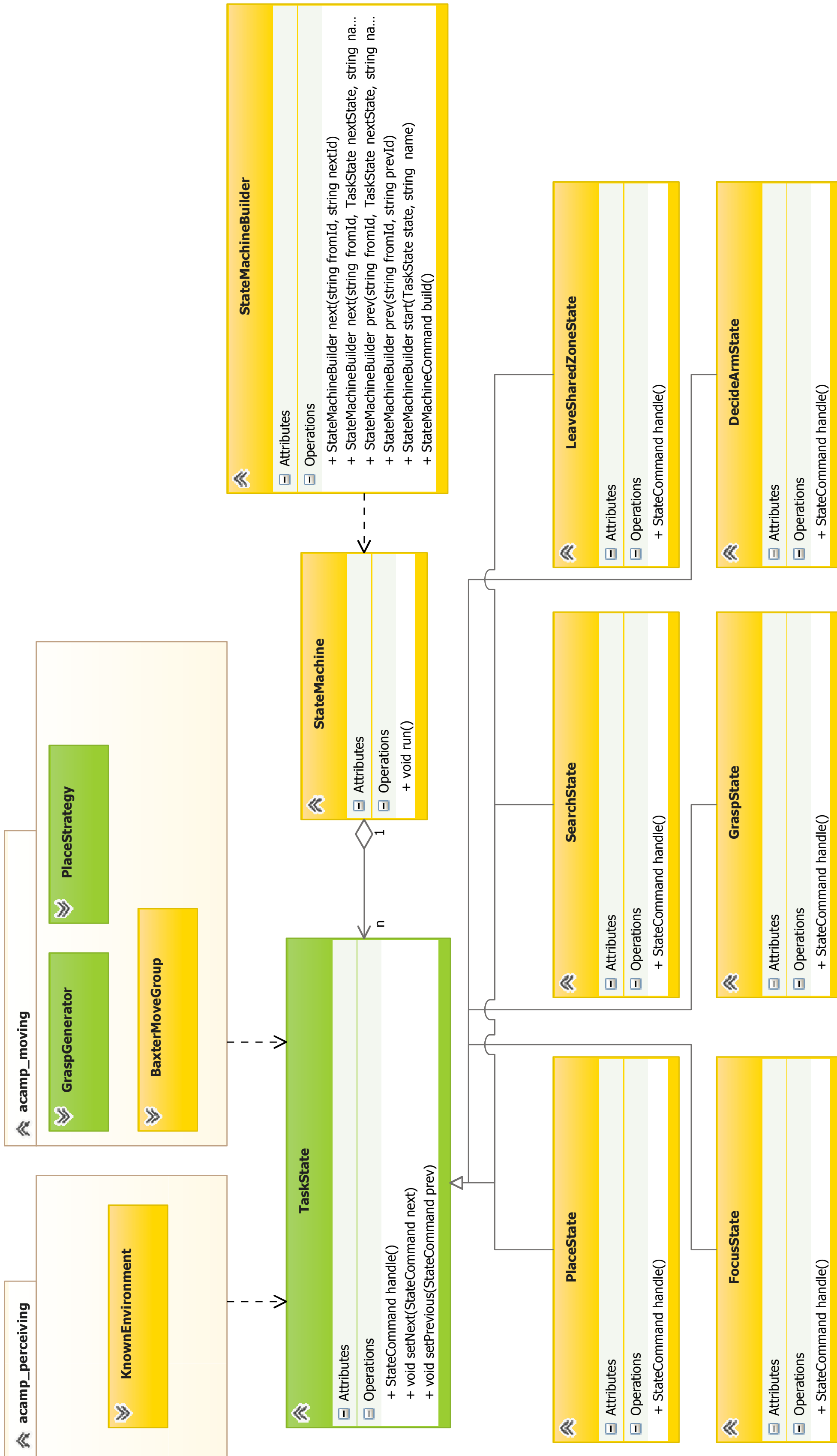


Figure 29: acamp_planning Package class structure

look for actual objects. In contrast, having no *blind_search_node* would cause the state to block during the execution of the motion, and thus it would not be able to look for objects. Therefore a search would never return results. While multi-threaded approaches are able to circumvent this problem on the state level, they would nonetheless result in higher synchronization effort, as the actionlib interface already encapsulates synchronization. Note that the cameras nevertheless rest shortly on some fixed positions as the AR Track Alvar library works best with stable non-moving pictures.

The search criteria itself can be adjusted by setting the desired filter predicate on the *KnownEnvironment* class before calling the search (see Section 4.2).

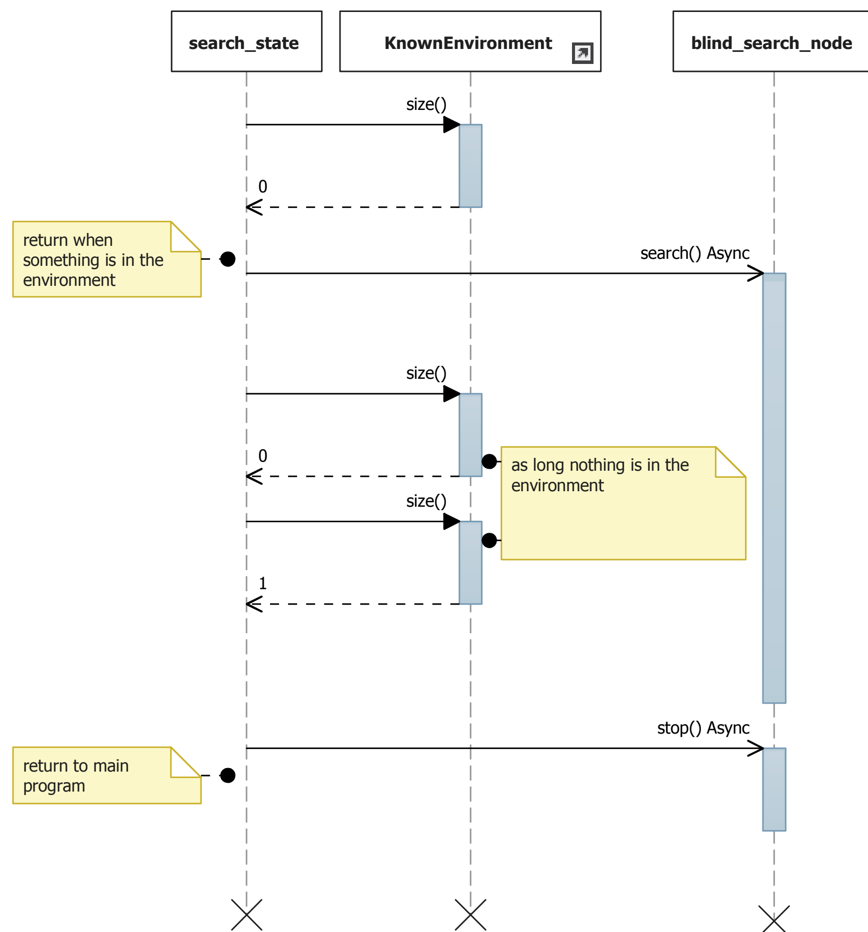


Figure 30: A search_state as an UML Sequence Diagram

The *focus_state* is used to refine the pose of the object the robot currently works with. While it is possible to estimate the pose of an object with the *perceiving_node*, this estimation is often inaccurate be-

cause of the suboptimal angle and distance between the object and the arm (see Section 2.4.3).

Although moving the camera closer to the object is the easiest way to refine the pose estimation, some restrictions apply. The new position needs to have a good camera angle and respect the minimal distance between the marker and the camera. In addition, as the end effector has gripper fingers attached, some areas of the camera image are obstructed (see Figure 31).

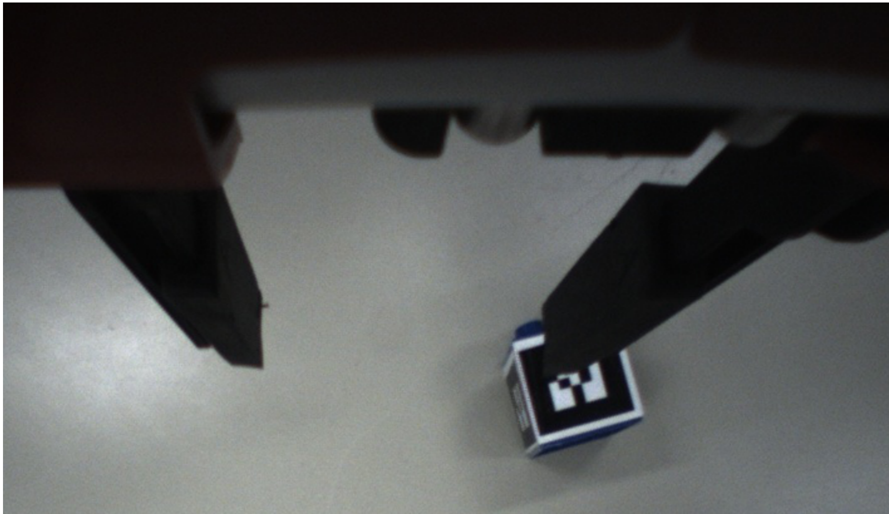


Figure 31: Gripper fingers obstructing the view of the camera

This state deals with this restrictions by moving the camera not directly over the object, but rather shifting the new position depending on the object in such a way that the gripper fingers do not obstruct vision.

On the class level, this state connects perception and movement through the use of the *KnownEnvironment* class and the *BaxterMoveGroup* class. In contrast to the *search_state*, motion and perception are not executed in parallel. A sequential iterative approach is used, moving the camera to the estimated position and checking perception for the distance between object and camera again. This approach is favorable because the AR Track Alvar library works best on camera streams coming from fixed positions, and the time loss during these relative small motions is negligible.

The third state encompassed in the default states is the *grasp_state*. The *handle* method of this state grasps the object that is the closest to the end effector as it is usually the most promising target. This state handles a variety of errors that can happen during grasping, like a missed grasp or a failure in the motion planner itself. The grasp contains all three stages as described in (see Section 2.4.4). The concrete characteristics of the grasp are injected through a Strategy pattern

(see Section 3.4) with the abstract *GraspGenerator* type as a generic strategy. This enables changing the grasp type not only by object, but also by external state. If we need a handle grasping algorithm, we implement another *GraspGenerator* that calculates the grasp for objects with handles. The *grasp_state* itself with its logic remains unchanged.

The *place_state* combines MoveIt!'s primitive movements to build a generic placing approach. During these steps, we control the movement with the *BaxterMoveGroup* class. Each step is done in a transactional way. This means it is impossible to jump from the first step to the third if the second state yields an error. In addition, the state comprises error handling like failing motion planning or failing controllers by starting the current step one more time. This is needed as movement and gripper hardware controllers of the robot sometimes return a failure even though the actual action is still possible. If we still run into the error, we change our position slightly to give the motion planner a chance in a new sampling situation (see Section 2.4.1). Finally, when both error handling strategies fail, the state transits over its previous transition.

As defined in the *grasping_state*, this state contains a generic place strategy implemented by a strategy Design Pattern (see Figure 26). We pass the strategy during initialization of the class, and during the runtime it is not visible what strategy we use. This indirection allows a framework to change the placing pose algorithm without altering the other behaviors of the *place_state*, reducing the work effort needed.

The framework also includes states that are useful for operating both arms together. These states solve two problems: first they help to move the arms out of a specific region over the table, and second they help to decide with which hand a motion is executed. Note that to use this states, both arms need to be controlled by the same state machine.

The *decision_state* decides what arm is used based on two concepts. The distance concept always favors the arm that is closer to the desired pose, while our absolute position concept divides the workspace in two disjunctive areas such that each arm is responsible for its area. We combine these concepts in the following way. If the desired pose is more than a threshold value away from the border of a disjunctive area, the arm responsible for that area is used. However, if the desired pose is relatively close to the border of the areas, the arm closer to the object is used. This has the advantage of often using the arm that is closer to the target and nevertheless having only a few situations when arm inference problems need to be solved, because every arm has its disjunctive area (see Section 2.4.3). However, the control could change while one arm is in the shared area, causing a potential interference.

To solve this problem, we introduce the *leave_shared_area* state in the State Machine. This state can be called whenever a switch from

one arm to the other happens. It checks if the currently used arm is in the shared area and moves it to the middle of its core area. This guarantees that we always have a free shared area after we switched the operating arms.

Finally, the Package includes the necessary interface to construct new states. The approach is thereby easy as the framework user just needs to inherit from the *TaskState* base class. This inheritance provides the new state directly with access to the Environment through the *KnownEnvironment* class and with the ability to move an arm through a *BaxtermoveGroup* class. Furthermore, the state is usable in any State Machine because of its polymorphism. The state can also be completely equipped with own members. It is important to mention here that C++ does not contain automatic memory management. While the base state class destructor is virtual and frees all resources used by it, an inherited state might needs to implement its own destructor.

4.5 acamp_gripper_state_publisher

The *acamp_gripper_state_publisher* Package contains a simple Python helper Node. The *gripper_state_publisher* Node in this Package is used to publish the gripper states to the default topic */joint_states* where the MoveIt! expects them for planning (see Chapter 5).

4.6 acamp_fixed_environment

The *acamp_fixed_environment* Package contains a helping Node (*acamp_fixed_environment*) publishing the fixed parts of the environment like the table. This Node is written in Python and publishes in a fixed interval of 2 Hz to MoveIt!'s environment.

5 IMPLEMENTATION DETAILS

5.1 PROTOTYPES

This section describes briefly implementation ideas that were tested during the project but were not pursued until the end. In general, ideas were not pursued for one of the following three reasons. First, if a better idea, more tailored for the Rapid Prototyping approach was found. Second, if an idea's implementation effort was too high and it was dropped because of time constraints. Third, an idea was desirable except that all known third party implementations were not fully developed and a custom implementation would again result in too much effort. This implies that some ideas listed below could be pursued in future projects as they indeed provide some value for the framework.

To begin with, during the analysis of the perception feature (see Section 5.5) several other approaches were evaluated. We experimented with various image processing ROS Packages. However, most of them were old and not ported to the new ROS version and therefore had installation problems due to the old build system (*rosmake*). Finally, aside from AR Track Alvar, two other approaches seem promising, the *Object Recognition Kitchen* of Willow Garage and the *OpenCV* Library.

The Object Recognition Package is tightly coupled with the use of a *Microsoft Kinect* camera. However, in our evaluation we were not able to use the Kinect for tracking Lego pieces. Probable causes for this were either suboptimal positioning of the camera, as we needed to mount the Kinect on the robot, or an inadequate training of the recognition database due to time limitations. We therefore dropped the Object Recognition Kitchen therefore as the time and integration effort was considered too high.

However, the Object Recognition Package might be a good idea for object recognition assuming integration time is available. Compared to the AR Track Alvar Library it offers more accurate depth perception and fewer restrictions regarding illumination. In addition, it allows for the use of complex geometrical shapes without prior preparation through sticking a marker on them or providing the geometry of the shapes.

The *OpenCV* approach was also postponed because of the limited time frame of our project and the difficulty of use of this relatively low-level library. For instance it offers no simple way of coordinate-

frame transformation or depth perception although, it provides all of the standard algorithms that could be used as building blocks for these problems.

Considering the power of this low-level image processing library operating directly on the cameras of the robot, it could be suitable for future projects relying widely on complex information from images. A good example for such a project would be the use of a Baxter robot for some sort of inspection.

Second, for robotic movement Rethink Robotics API was also evaluated. Its IK-Solver is faster and often more accurate than MoveIt!'s IK-Solver as it operates directly on the hardware of the robot, without network latencies. Another advantage is that the IK-Solver also performs well for planning with two arms often reporting a short way. However, the API does not provide motion planners. This means, that no knowledge about the environment is considered during planning leading to collisions, with object in the environment, like the table or the Lego pieces (see Section 2.4.1). This is the main reason we chose MoveIt! in favor of Rethink Robotics' motion API. However, depending on the task, Rethink Robotics, motion planning can also be favorable, especially when collision-free planning is not needed or realizable without a general motion planner. In such cases, the hardware IK-Solver outperforms MoveIt! significantly.

During the integration of MoveIt!, the multi-arm parallel motion planning was also tested. Despite the fact that the basic planning worked, the resulting trajectories were often not usable for three main reasons. First, the trajectory of each arm was too long and often arrived at the goal after several strange and unnecessary arm movements. Second, due to these strange trajectories, the robot became less acceptable for human beings as it performed many unpredictable movements. Furthermore, the robot was perceived as much less effective because of the long trajectories. Third, the general planning time increased, leading to a robot that often did not move for several seconds after receiving a planning request. We therefore moved away from the idea of parallel dual arm usage, although further development of the MoveIt! algorithm could change our opinion.

Third, we switched from the Python API to the C++ API. The change was mainly needed because of the use of MoveIt! as our primary motion planning library. While Rethink Robotics' API is based on Python, MoveIt!'s Python integration is rudimentary. It does not provide all functionality needed as a Python function and lacks some useful return information in its Python API. The rewrite of our first implementation attempts from Python to C++ therefore gave more control over the motion planning and enabled us to, for example, fix a severe spline interpolation bug (see Section 5.4).

5.2 WORKSPACE SETUP

The workspace setup is briefly presented in this section as it fixes some practical problems that the default way of setting up the workspace introduces. This section only describes the changes of our setup as compared to the guide at:
https://github.com/RethinkRobotics/sdk-docs/wiki/Networking.

To begin with, our general folder structure is the same as the standard ROS workspace structure except that custom Nodes are aggregated in an additional folder by programming language. This does not affect the build system of ROS and is therefore a good idea for developers working with Integrated Development Environment (IDE)'s like *Eclipse*, as they can now set up a workspace for a specific programming language in this additional folder. To use this workspace with Eclipse it is important to generate the ROS independent project files with the command shown in the *.bashrc* example.

Second, we defined some aliases in the *.bashrc* file of our user for often used commands (see Listing 4). Over time, this saves a lot of typing effort and eases the daily work with the robot. The *bxt* alias opens a shell completely configured for working with the robot, removing the need to set environment variables or run setup scripts. The other aliases simplify the start, stop and reset of the robot, allow querying of its state, create a PDF with the recent tf frames and their connection or set up a Node for editing it in the Eclipse IDE.

Listing 4: Aliases in the .bashrc making the work easier

```
BXT_WS=~/ros/baxter_ws

source /opt/ros/hydro/setup.bash
source $BXT_WS/devel/setup.bash

#
# Aliases
#
alias bxt="cd $BXT_WS && $BXT_WS/baxter.sh"
alias bxtEnable="roslaunch acamp_demo baxter_startup.launch"
alias bxtDisable="rosrun baxter_tools enable_robot.py -d"
alias bxtReset="rosrun baxter_tools enable_robot.py -r"
alias bxtState="rosrun baxter_tools enable_robot.py -s"
alias tf='cd /var/tmp && rosrun tf view_frames && evince frames.
    pdf &'
alias catkin_eclipse_generate="cmake -G \"Eclipse CDT4 - Unix
    Makefiles\"
```

Third, the most important change from the default installation is the different network setup (see Figure 32). This was needed so that the robot could not reach the Internet due to security considerations,

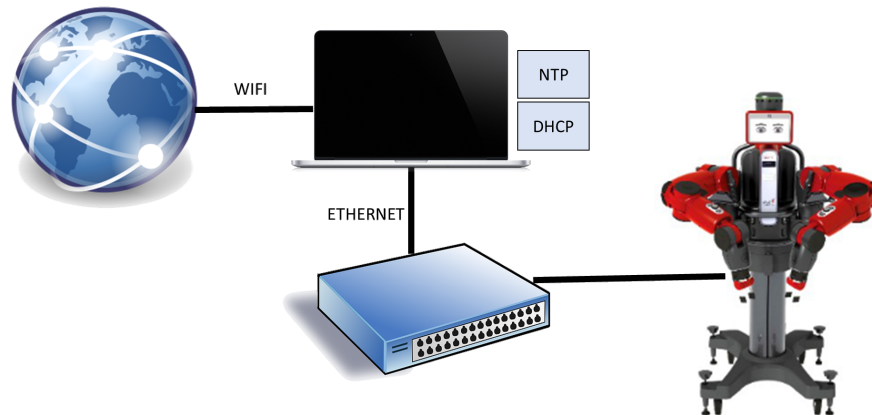


Figure 32: Network setup

but could nevertheless synchronize its time over the Network Time Protocol (NTP) as the time between developer machine and robot need to be the same to receive valid sensor data (see Section 3.1). Furthermore, our networking should erase the need to set the IP address after rebooting the developer machine.
To resolve these issues, our workstation runs its own Dynamic Host Configuration Protocol (DHCP) and NTP server. The DHCP server is configured so that it returns a fixed IP and the developer machine as NTP server when the robot queries it during start up. Determining the address to provide is done by binding the IP address to the Media Access Control (MAC) address of the robot that can be found in the field service menu. As our developer machine has two network interfaces, one is used to communicate with the robot (*eth0*) while the other one (*wlan0*) is used to communicate with the Internet and to keep NTP in sync. Therefore the *eth0* interface has a fixed address in the subnet of the robot configured through *ifconfig*.

5.3 ADAPTING FOR MOVEIT!

MoveIt! is an excellent library for motion planning for almost any kind of robot. However, this flexibility implies integration effort as robotic hardware varies greatly. For example, the library needs an exact model of the robot and its actors for IK solving and motion planning. While this is not the first time MoveIt! has been used on a Baxter robot[1], we describe our integration approach here because it fixes some detail problems of the former integration procedures and was a large part of our work.

To adapt the library for the robot the following steps are necessary:

1 BaxterCPP by Dave Coleman https://github.com/davetcoleman/baxter_cpp

- remapping the */joint_states* topic to the */robot/joint_states* topic
- delivering the missing robot state information to the library
- changing the URDF to contain the end effectors
- adjusting the collision meshes
- publishing the tf frames for the end effectors
- altering the controllers
- altering the joint speed
- configuring the planner and the tolerances

Starting by remapping the */joint_states* topic to the */robot/joint_states* topic is important, as MoveIt! bases all motion planning on the received joint states telling the library the joint positions. It is necessary because the robot publishes the joint states to the non-standard */robot/joint_states* topic. That is the reason why we are remapping the states in the MoveIt! launch file to enable the library to read the states of each joint of the robot for planning. This mapping is also used in the former integration approaches and does not influence the robot's general behavior as the */joint_states* Topic is still present.

The next step is it to complete the published joint state information. Since the robot has multiple different types of grippers that can be attached to the end effector at runtime, Rethink Robotics decided to exclude the end effector joint states from the standard *joint_states* topic published. Instead, the */robot/[side]_end_effector/state* includes this information in a custom Message format. However, as long as these states are not known to MoveIt!, it ignores the end effectors completely. We therefore implemented the *gripper_state_publisher* Node, which extracts the missing information from the */robot/[side]_end_effector/state* Topics custom message format and publishes it under the */joint_states* topic (see Listing 5).

Listing 5: The *gripper_state_publisher* Node

```
DEFAULT_NODE_NAME = "acamp_gripper_state_publisher"

class GripperStatePublisher(object):
    BASE_LINK = 'base'
    FINGER_OPEN_POSITION = 0.0095  # open position of the gripper
    FINGER_CLOSE_POSITION = -0.0125  # close position of the
        gripper

    def __init__(self):
        self._publisher = rospy.Publisher("/robot/joint_states",
            JointState)
```

```
        self._left = "left"
        self._right = "right"
        self._gripperLeft = Gripper(self._left)
        self._gripperRight = Gripper(self._right)
        self._fingerJointStroke = GripperStatePublisher.
            FINGER_OPEN_POSITION - GripperStatePublisher.
            FINGER_CLOSE_POSITION;
        self._fingerJointMidpoint = GripperStatePublisher.
            FINGER_CLOSE_POSITION + self._fingerJointStroke / 2;

    def publish(self):
        jointState = JointState()
        self._createJointState(jointState, self._right,self.
            _gripperRight)
        self._createJointState(jointState, self._left,self.
            _gripperLeft)
        self._publisher.publish(jointState)

    def _createJointState(self,jointState, arm, gripper):
        jointState.header.frame_id = GripperStatePublisher.
            BASE_LINK;
        jointState.header.stamp = rospy.Time().now()
        jointState.name.append("%s_gripper_l_finger_joint" % arm)
        jointState.name.append("%s_gripper_r_finger_joint" % arm)
        jointState.velocity.append(0)
        jointState.velocity.append(0)
        jointState.effort.append(gripper.force())
        jointState.effort.append(gripper.force())

        position = GripperStatePublisher.FINGER_CLOSE_POSITION +
            self._fingerJointStroke * (gripper.position() / 100);

        jointState.position.append(position)
        jointState.position.append(position * -1)

if __name__ == '__main__':
    rospy.init_node(DEFAULT_NODE_NAME)
    gripperStatePublisher = GripperStatePublisher()
    rate = rospy.Rate(50)

    rospy.loginfo("publishing gripper states...")

    while not rospy.is_shutdown():
        gripperStatePublisher.publish()
        rate.sleep()
```

Although one of the former approaches designed a similar Node we decided to write this small Node again. Consequently, our Node removes the tight coupling with the rest of the other integration process resulting in fewer dependencies.

Besides making the missing joint states of the end effectors available for the library, the URDF must also specify these missing joints

and their physical attributes. Otherwise, the library sees joint states that have no physical representation and ignores them. Therefore an updated URDF was used, including this joint information and the collision meshes for the end effectors (see Figure 33).

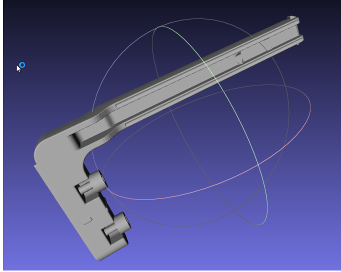


Figure 33: Finger mesh

To publish this URDF file to the robot, we wrote a small launch script that needs to run one time after powering on the robot.

Although the previous step introduces collision meshes, they were initially not used for collision checking due to an incompatibility in the mesh models with MoveIt!. In the model the up axis is the Y-axis while in MoveIt! the axis pointing up is the Z-Axis. Therefore MoveIt! interprets the mesh as lying on its side.

For this reason, we altered the coordinate system of the mesh to mirror the other fixing a major bug in the former integration processes causing the grippers to not be considered for collision checking. Note, that it is especially hard to find this bug as RVIZ uses a different mesh loading library to display the robot visually. This loader handles the axis representations correctly leading to a seemingly correctly positioned mesh in the simulator.

Having joint states and an URDF representation the end effectors still miss the tf frames, which are critical for a proper motion planning. They are important as they provide information about how the end effectors, coordinates relate to the rest of the coordinate systems. This information is needed to find the position of the end effector during a move through the relation between its tf frame and the world tf frame.
To publish the end effectors tf frames, we used a Node from that calculates them through the information provided in the URDF.

Having a complete robot model available for MoveIt!, the configuration files need to be adjusted. This allows the use of the default hardware controllers of the robot by specifying their actionservice topics. Although, MoveIt! provide generic hardware controllers, the usage of Rethink Robotics hardware controllers was preferred based on benchmarking both approaches.

During these benchmarks, Rethink Robotics' controllers usually reacted more smoothly to the received trajectories and were more reliable in opening and closing the end effectors.

In the last step, the library is set up to control the robot. To verify that the library does not send commands that exceed the physical boundaries of the joints setting, the acceleration and velocity boundaries in the corresponding configuration files is necessary. Otherwise, send trajectories can result in a faster hardware wear or might even lead to hardware defects.

5.4 SPLINE INTERPOLATION ERROR

We encountered a problem in the combination of the motion planner of MoveIt! and the hardware trajectory controllers of Rethink Robotics during this project. This problem and its fix are described in the section below. We included this section here because it illustrates the sophisticated interaction between the different parts of the program that results in an executed motion. Furthermore, this problem was very critical as it affected many executed motions especially during manipulation tasks.

The problem appeared during the execution some trajectories, resulting in stopped trajectories or endless wiggling motions. Observing the logs showed that the trajectories were not correctly executed by the hardware as some joints reported a position constraint violation causing the joint to not execute the send position change. As a result, the arm as a whole failed to reach the goal position and stopped earlier or began to wiggle due to endless attempts to reach the position.

Executing such a trajectory involves two stages. First, MoveIt! calculates a *RobotTrajectory* message. Second, the *RobotTrajectory* message is sent to the hardware controller of the robot (*/joint_trajectory_action_service*) and is executed by it. The *RobotTrajectory* message comprises the positions, velocities, accelerations and joint efforts for each step in time of the trajectory. During the execution of such a trajectory, the hardware controller spline interpolates between two points in the trajectory. The data points used by the spline are here the values of each *JointTrajectoryPoint* (see Listing 6).

Listing 6: The *RobotTrajectory* message and its parts

```
//RobotTrajectory
trajectory_msgs/JointTrajectory joint_trajectory

//JointTrajectory
Header header
string[] joint_names
JointTrajectoryPoint[] points

//JointTrajectoryPoint
float64[] positions
```

```
float64[] velocities
float64[] accelerations
float64[] effort
duration time_from_start // time the position should be reached
```

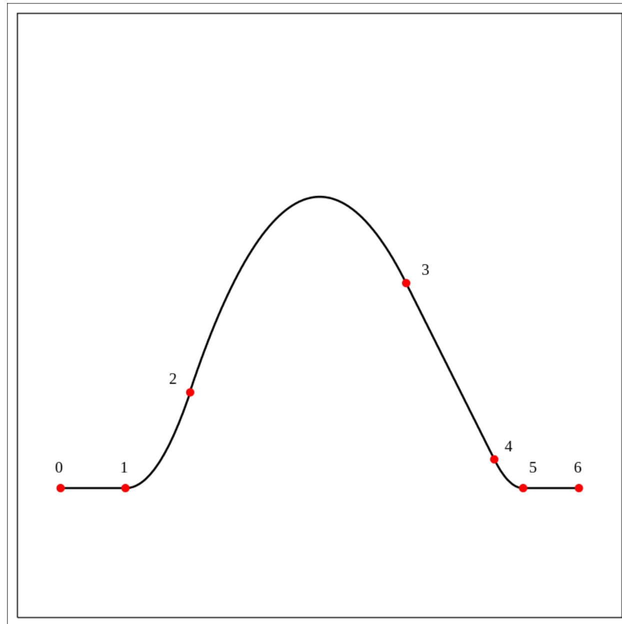


Figure 34: A quadratic spline composed of six polynomial segments [1].

A *spline interpolation* is a numerical algorithm that approximates a set of data points by a function of connected piecewise polynomials (the spline, see Figure 34). Depending on the model of the source function the degree of the polynomials can be chosen higher or lower. This technique is often used for interpolation as even rapid to calculate low order polynomials lead to relative small errors.

Mathematically a spline S of order k has the following properties [32]:

1. S is on $[x_i, x_{i+1}]$ a polynomial of order $\leqslant k$
2. $S(x_i) = f_i, i = 0, 1, 2, ..., n$
3. $S(x)$ is continuous and continuously differentiable to the order of $k - 1$

A spline polynomial that computes the position p at time t has then the form:

$$p(t) = c_0 + c_1 * t^1 + c_2 * t^2 + ... + c_{n-1} * t^{n-1} + c_n * t^n \qquad (1)$$

Where $c_i | i = \{0, 1, 2, ..., n\}$ are the spline coefficients calculated by the interpolation algorithm. Furthermore, as we can assume that the partition of our data points is uniform due to the uniform time intervals

1 "Quadratic spline six segments" by Stamcose - Own work. Licensed under Creative Commons Attribution-Share Alike 3.0-2.5-2.0-1.0 via Wikimedia

we are interpolating on, the general error estimation for our spline is the Lagrange Error Bound $E_n(x)$, where f is the interpolated function:

$$|E_n(x)| \leqslant \frac{f^{(n+1)}(\xi)}{(n+1)!}(x_{i+1} - x_i)^{n+1} \tag{2}$$

In the most basic form, the spline polynomials used to interpolate the *RobotTrajectory* have an order of one. In this case, every polynomial in the spline connects two adjacent data points with a straight line. This calculation is based on the start and end time t_i, t_{i+1} and the start and end position p_i, p_{i+1} of the motion between two points in the *RobotTrajectory*.

$$p(t) = c_0 + c_1 * t = p_i + \frac{p_{i+1} - p_i}{t_{i+1} - t_i} * t \tag{3}$$

The maximum expected error is:

$$|E_1(x)| \leqslant \frac{f^{(2)}(\xi)}{2}(x_{i+1} - x_i)^2 \tag{4}$$

The next used spline is comprised of cubic polynomials (order 3). Here, the interpolation is not only based on position and time but also on velocity v_i, v_{i+1}, as additional information is needed to construct the spline. The maximum error is:

$$|E_3(x)| \leqslant \frac{f^{(4)}(\xi)}{4!}(x_{i+1} - x_i)^4 \tag{5}$$

The highest order spline used in the controller comprises quintic polynomials (order 5). The interpolation needs position, velocity and acceleration a_i, a_{i+1} as parameters to construct the spline. The maximum error is:

$$|E_5(x)| \leqslant \frac{f^{(6)}(\xi)}{6!}(x_{i+1} - x_i)^6 \tag{6}$$

The error equations show that it is in general not true that the maximal error decreases using higher order splines. This can be inferred from the unknown factor $f^{(n+1)}(\xi)$ in the error equations [25][pages 40-48]. Since f and ξ are not known we can for example not tell if $E_5(x) < E_3(x)$ because $f^{(6)}(\xi)$ might be much larger than $f^{(4)}(\xi)$. Therefore a higher order spline might overestimate the function leading to a larger error than a lower order spline.

However, the bug only appeared when the controller interpolated linearly. Although it was impossible to calculate the exact error bounds as the source function f that generated the trajectory in the first place was not documented, we inferred that the interpolation error was too high and the joints therefore refused to alter position. Having made

this observation, it was still unclear what caused the controller to interpolate linearly as it seemed that MoveIt! calculates the trajectories with all necessary values for a higher order interpolation. However, while it calculates position, velocity and acceleration for all normal motions does not do this during the calculation of straight motions (Cartesian paths) due to a bug in the library. Such motions only contain the positions for the joints and no other values explaining what caused the controller to do a linear interpolation.

To fix this, we introduced an additional time parameterization in our *BaxterMoveGroup* class calculating velocity and acceleration for every following Cartesian Path. As this functionality already exists in MoveIt! for non-Cartesian Paths, it provided us with a class that does this time parameterization given a *RobotTrajectory* with position fields filled (see Listing 7).

Listing 7: Fixing the spline interpolation bug

```
void BaxterMoveGroup::doTimeParametrize(moveit_msgs::
    RobotTrajectory& traj) {
        // First to create a RobotTrajectory object
        robot_trajectory::RobotTrajectory rt(
                        this->getCurrentState()->getRobotModel(),
                             this->getName());
        // Second get a RobotTrajectory from trajectory
        rt.setRobotTrajectoryMsg(*this->getCurrentState(), traj);
        // Third create a IterativeParabolicTimeParameterization
            object
        trajectory_processing::
            IterativeParabolicTimeParameterization iptp;
        // Fourth compute computeTimeStamps
        bool success = iptp.computeTimeStamps(rt);
        ROS_INFO("Computed time stamp %s", success ? "SUCCEDED" :
             "FAILED");
        // Get RobotTrajectory_msg from RobotTrajectory
        rt.getRobotTrajectoryMsg(traj);
}
```

Finally, it is important to say that this fix works only in combination with firmware version 1.0 or higher. Before this version, the hardware controller interpolated always with a linear spline.

5.5 PERCEIVING PACKAGE

This section shows some of the implementation details of the perceiving Package. While the previous sections described non programming related challenges, this chapter discusses, based on the example of the perceiving Package, typical programming challenges solved during this project. The concurrency, memory management and per-

formance challenges are inherent to all ROS programs because they integrate in a distributed system.

As described earlier (see Section 4.2) the perceiving Package deals with sensor data coming from both cameras and merges the most reliable data into the environment while the other data is dropped.

The raw data is reported in a frequency of 10 Hz and includes all markers currently visible in the camera stream. Each camera reports the data on its own thread allowing parallel processing of the two data streams at once without locking, as each data stream is read-only. Assuming that each data stream includes up to ten markers, 10 ms are left to process both data streams.

To use the data, the following points needs to be inferable from it:

- is a marker visible
- is a marker no longer visible
- is the perceived pattern really a marker or a false alarm by the library

Inferring about these three items in 10 ms requires an algorithm that does not involve large calculations. We therefore use a cache-like data structure (see Listing 8) that allows us to deduce the first two conditions from two simple counters.

Listing 8: The data structure used to store the raw sensor data

```
class MarkerCache{
        struct CacheRow {
                /*****
                * general fields
                *****/
                ar_track_alvar::AlvarMarker marker;
                int lastSeen;
                int addTreshold;
                geometry_msgs::Pose armPose;

                /*****
                * fields for moving average calculation
                *****/
                //position in the ringbuffer
                int pos;
                //counting until buffer is completely filled
                int valuesInBuffer;
                //current moving average
                double movingAvg;
                // for the computation of the delta used in
                    movingAvg calculation
                double oldDistance;
                // ringbuffer for the last n deltas
```

```
                double distances[BUFFER_SIZE];
        };

        /***
        * more things leaved out for simplicity
     ***/

        private:
                std::map<int, CacheRow> markerCache;
}
```

The *add threshold* tells us when a marker becomes visible. A marker having this field over the threshold will be considered visible. It prevents false positives as they usually only get reported for a very short amount of time. The *remove threshold* tells us when a marker is no longer visible. A marker with a *lastSeen* field over the threshold will be considered invisible. The calculation of each such field includes only one add and one if-statement to prevent overflows that were otherwise common because of the fast frequency of reported data points.

Although these two values cover most false positives cases, there exists another case where the maker begins to jump through the environment because of an excessively steep angle between camera and marker.

For this, the callback includes also two basic filter mechanisms. An Exponentially Weighted Moving Average (EWMA) filter that is used to reduce the influence of short term jumping and a *Moving Average* filter that is used to exclude markers that jump too much.

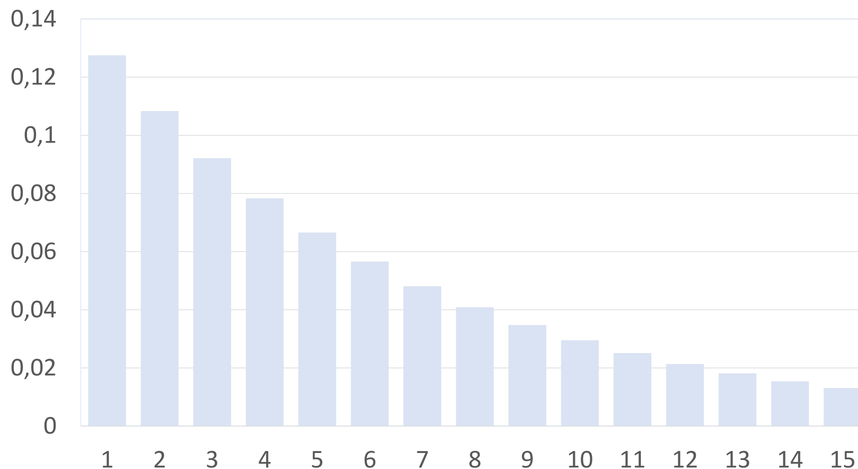
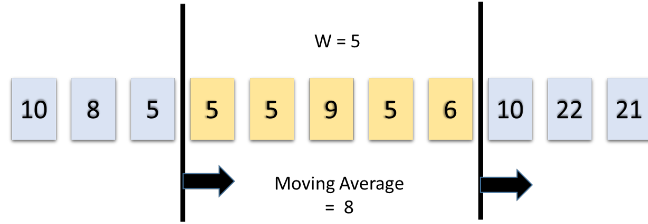


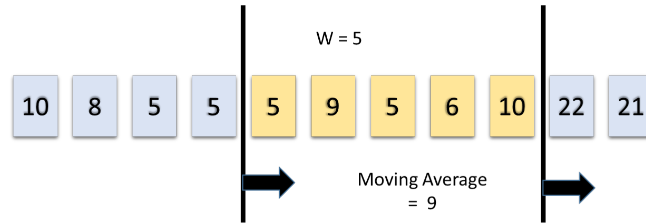
Figure 35: The weights of the moving average plotted for $n = 15, \alpha = 0.15$, the largest weight is for the newest datum

The EWMA filters middle position changes while giving younger data points a higher importance (see Figure 35). In other words, the

influence of older data points on the new position decreases exponentially. Therefore, it is good to reduce short term random position switching, as it happens because of the cameras moving while recognizing the marker. In such cases the exponential weights prevent large jumps as the previous position data is also considered. In addition, the calculation of such a filter is fairly easy and fast (alpha is a filter constant, in our case 0.3):



(a) Moving average calculation over a window of five elements



(b) Window moved one step further

Figure 36: Calculation of a moving average illustrated

$$\vec{P_{new}} = \vec{P_{old}} + \alpha * (\vec{P_{new}} - \vec{P_{old}}) \tag{7}$$

The *Moving Average* (see Figure 36) is used to look for long-term jumping markers. It is calculated for the last 30 differences of the distances and the marker is dropped if this average difference is too high. Using the recursive formula, the calculation complexity during each callback invocation reduces as we do not need to calculate the sum of all 30 differences (the w in the equations) first:

$$MA(t) = \frac{1}{w} * \sum_{t-w+1}^{t} y_i \tag{8}$$

$$MA(t+1) = MA(t) + \frac{y(t+1) - y(t-w+1)}{w} \tag{9}$$

Note that the calculation necessitates storing the last 30 differences as we need to know what difference to exclude from the calculation next. For the storing of these differences a simple *Ringbuffer* is used.

The introduced cache structure (*set<CacheWorld>*) together with the algorithms used allow us to deduce the necessary information in a

short amount of time and to reliably process the marker data. Note, that a set is used here to keep the cache as dense and small as possible. Having circa 1 kb marker data every 100 ms would otherwise either result in a large search effort to find the correct entry in another data structure or in an excessive memory usage when old values are not overwritten.

The further steps that decide upon which marker to rely are not time critical as they work with a copy of each cache that includes all information. Next, the content of each cache is merged in $O(n)$ retaining the marker that was closer to the camera. This works because both sets of marker data are sorted by ID allowing to apply a linear merge. Finally, the content needed is published every 10 Hz to the environment. Here, a slow rate is important as publishing to the environment locks it for every other process including the motion planner. Publishing too often slows the motion planner therefore down.

6 RELATED WORK

The Baxter robot is a relative new platform with about 500 units delivered to customers around the world. This customers are either firms that try to integrate Baxter into their production environment or academic users who start research projects with Baxter [19]. We summarize the main points of this works in the following sections. After that we describe where and how our work is different from the already existing projects and research papers.

6.1 PROJECTS IN THE BLOGOSPHERE AND YOUTUBE

To begin with, Rethink Robotics mentions four customers on its website that use Baxter with the manufacturing version already: GENCO, Vanguard plastics, Keter Plastics and the Rodon Group[1]. We see Baxter performing tasks in this firms like sorting and packing. It is used as a work load multiplier helping with repetitive logistic tasks often at an assembly line. All firms state that they like the ability of the robot to work close to humans and being easy programmable by untrained staff.

Second, the scientific community starts first projects with Baxter. In contrast to the commercial sector we can see a variety of use cases being currently explored in unpublished work (see Table 2).

Most of them are presented through small demo videos created by Rethink Robotics or the research team. Because the demos shown are often first implementation attempts and in an early stage, there is normally no further documentation published.

Finally, some research papers that involve Baxter are already published.

6.2 RESEARCH PAPERS

First, [15] and [2] focussed on the learning of trajectory preferences without optimal training data. They present an algorithm that allows Baxter to learn how to preferably carry things trough iterative feedback from users. This algorithm is evaluated in different grocery store and household scenarios. These scenarios fit in three categories. First, a manipulation centric scenario, in that Baxter does tasks like pouring water into a glass. Second, an environment centric scenario in that Baxter is working with fragile objects like eggs. Last, a human

1 http://www.rethinkrobotics.com/resources/videos/#baxter-customer-spotlights

Table 2: Unpublished work on Youtube and in the blogosphere

use case	explored with
use of custom devices	• custom hands • custom Vacuum Cups
solving logic riddles	• playing connect four • solving a rubics cube
remote control	• Wii mote • Kinect • Hydra + Oculus Rift • XBOX Controller • custom devices for disabled people
introducing more complex behavior	• handling dangerous objects • reacting to changing environments
doing abstract human tasks	• folding a shirt • performing a simple chemical experiment

centric scenario in that Baxter carries dangerous objects like knifes. The authors state that they were able to teach the robot the preferred trajectory with only a few iterations. In addition, they state that the algorithm generalizes well on tasks the robot has not done before.

D. Nunez, M. Tempest, E. Viola, and C. Breazeal presented their work in progress paper [21] that discusses the use of the robot as an assistant for a magician. The authors pay special attention to the timing and chronology concerns. During the performance the robot interacts with the magician in various ways, it passes for example a ball to the magician. Also it executes a playlist of different poses during the performance. To record a pose the authors developed an interface that enables the magician to create poses in the zero-G mode of the robot. They also build a program enabling them to compose multiple poses into one big performance. This program shows a timeline and a simulation of the robot doing the poses in the performance and the corresponding transitions. It allows also to directly control the robot. In addition, it is possible to stop the playback and adjust the current joint positions. To keep the performance of the robot and the magician in sync, they use different cues that tell the magician where the robot is currently in its performance. The authors conclude that there are multiple future research topics, like branching into different storylines, getting a better production cycle or better timing between robot and human in this field.

T. M. Caldwell, D. Coleman, and N. Correll presented a paper [7] looking at algorithms for flexible object manipulation aiming to identify the behavior of a flexible object through touch only. The described algorithm implemented on a Baxter allows him to identify stiffness characteristics of a flexible loop.

Finally, *I. Lenz, H. Lee, and A. Saxena* presented in [17] an algorithm for robotic grasps trough deep learning. The goal of the papers was to find the optimal grasping point for the robotic hand, using a two stage detection system. It was implemented on a Baxter.

6.3 Distinctive qualities of our work

Compared to the recent published projects our work contains some unique approaches. To begin with, most of the work mentioned earlier deals with a very narrow problem field such as trajectory planning or flexible object manipulation. In contrast, our work focusses on a wider field and tries to elaborate the limits and opportunities of the whole robotic system. This means that it often gives priority to faster approaches like usage of fiducials that are more limited but lead to acceptable results in many cases.

Second, the demo videos from Rethink Robotics show some capabilities that our framework introduces. However, the base for these demos is their commercial software. This software is specialized on manufacturing and not freely programmable. On the other hand, we introduced a research framework in this project that is adaptable and can be used for a variety of tasks.

Third, the described projects that work with object recognition and randomness have different approaches to object recognition than we have. They use for example a Kinect as an additional camera with depth sensing or train their object recognition algorithms offline to recognize some objects before they use the robot with these. The fiducials used in our approach are a completely different approach. They provide easy recognition without the need of training given that the underlying geometry is not too complex. Furthermore, through the adaptability of the framework the reported position can be interpreted differently depending on the state allowing the use of the fiducial only as a fix point for a more complex geometry. In conclusion, this project introduces new ways in the work with the robot through the framework, its adaptability and the involved object recognition techniques.

7 RESULTS & DISCUSSION

7.1 RESULTS

The results of the thesis comprise a qualitative and a quantitative analysis. The quantitative analysis of the system compares the reliability and performance of a sorting demo built with the framework with the same demo implemented in Python without the framework. The Python approach was built in the start phase of this project while exploring the abilities of the robot. The demo has the following starting conditions:

- Lego pieces are randomly distributed on the table area (60 cm x 60 cm)
- Each piece is reachable with the arms in a 90 degree angle from above
- Each piece contains fiducials for pose estimation

The goal of the robot is to sort the randomly distributed pieces according to the spectrum of light on the side of the table (see Figure 37). The color information is retrieved by the fiducials identifier. The framework implementation uses a state machine comprised from the basic states as described earlier (see Figure 38).

To evaluate the different programs the following criteria were used:

- Number of missed grasps
- Average time needed to sort all Lego pieces
- Average time needed to move a single Lego piece to its final position

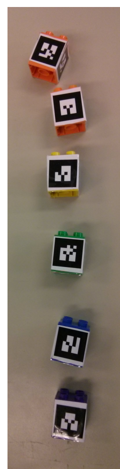


Figure 37: Sorted Lego pieces after the rainbow demo

The results of the evaluation show that the framework outperforms our first Python implementation. However, they also show that even with the framework the robot fails to pick in 20% of the grasps. This is a matter of the natural inaccuracy of physical manipulation given by the pose estimate and the hardware. Furthermore, while the framework implementation operates quicker, the robot is still much slower

Table 3: Benchmark results Python vs. C++ Node

TEST	C++	PYTHON
Number of grasps	40	40
Missed grasps	8	15
Maximum of retries before successful grasp	1	3
Grasp reliability	80%	62.5 %
Average time per grasp	45 s	72 s

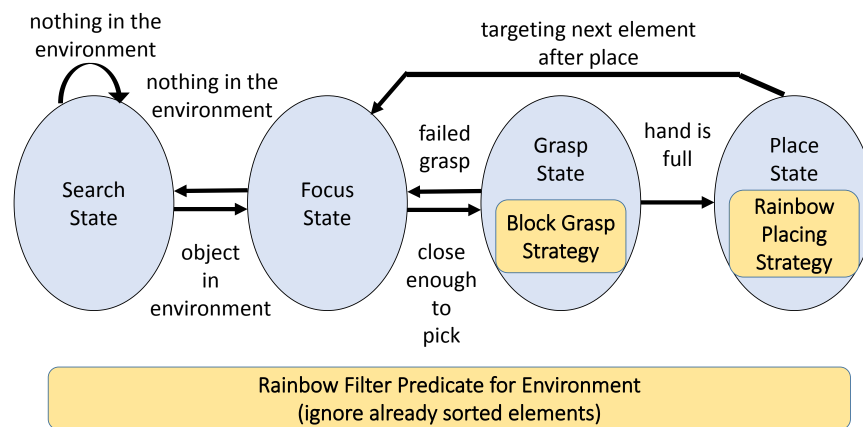


Figure 38: Single arm State Machine used for benchmarking

than a human solving the task. The explanation for this is based on two aspects. First, sample-based motion planning does usually not find the optimal arm trajectory leading to longer movements. Second, the object recognition with fiducials requires a still standing camera stream and does not allow a smooth ongoing movement for the whole task.

The qualitative aspects of the thesis were evaluated throughout a company seminar held by acamp. The attendees of the seminar were either robotic experts giving a talk about their current work or company executives informing themselves about the newest trends in robotics. The robot was presented in the breaks between the talks of the seminar, performing the demo described above.

The overall feedback was positive. For many people the demo provided a good introduction to the robot and got them thinking about possible applications in their business. This confirmed our approach to demonstrate the capabilities of the robot with an industry unrelated task. One of the resulting discussions even led to a collaboration with a company for the exploration of potential use cases of the robot in their assembly line (see Chapter 8).

People were in general not afraid of the robot. They crossed the working area of a moving robot without worries of getting hit and also redistributed the Lego pieces on the table during a demo standing side by side to the operating robot.

On the other hand, some people asked about the strange not human like motions of the arms. This shows that besides of the robot's humanoid appearance more work is needed to increase the general level of comfort for humans around the robot. As a consequence, later demos were extended with the head movement that indicates the motion direction of the arm.

Finally, the state machine concept of the framework was evaluated through the implementation of three demos. The first demo, is the already described sorting by light spectrum. The second demo evolved the basic state machine from the first demo by placing the Lego pieces in boxes instead of sorting by the spectrum. The last demo extends the second one by introducing sequential multi-arm processing of the pieces. The quality criteria applied to this first evaluation was the amount of code that can be reused from the first demo to the third demo.

The results show (see Figure 39 and Figure 40) that large amounts of the program logic were transferred throughout the demos. The implementation effort of the second and third demo was therefore relatively small compared to rewriting each demo from scratch. However, it is clear that this is a first and very specific evaluation scenario and that further tests are needed to find more strengths and weaknesses of the framework.

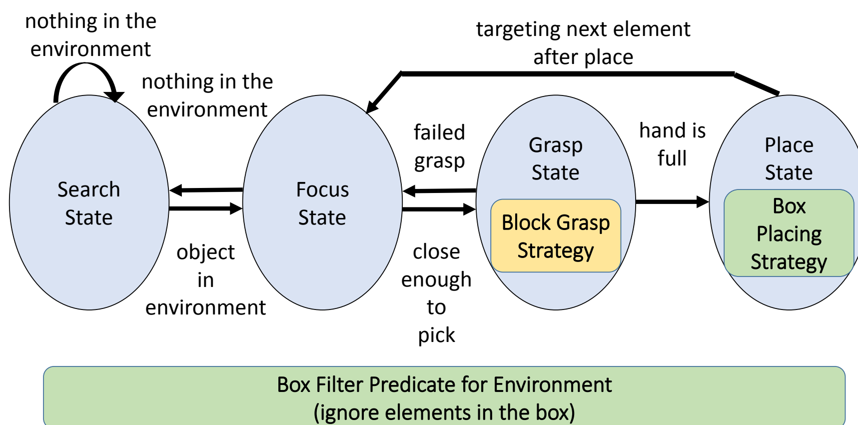


Figure 39: Changes of the rainbow placing State Machine that led to the single arm box placing State Machine (highlighted in green)

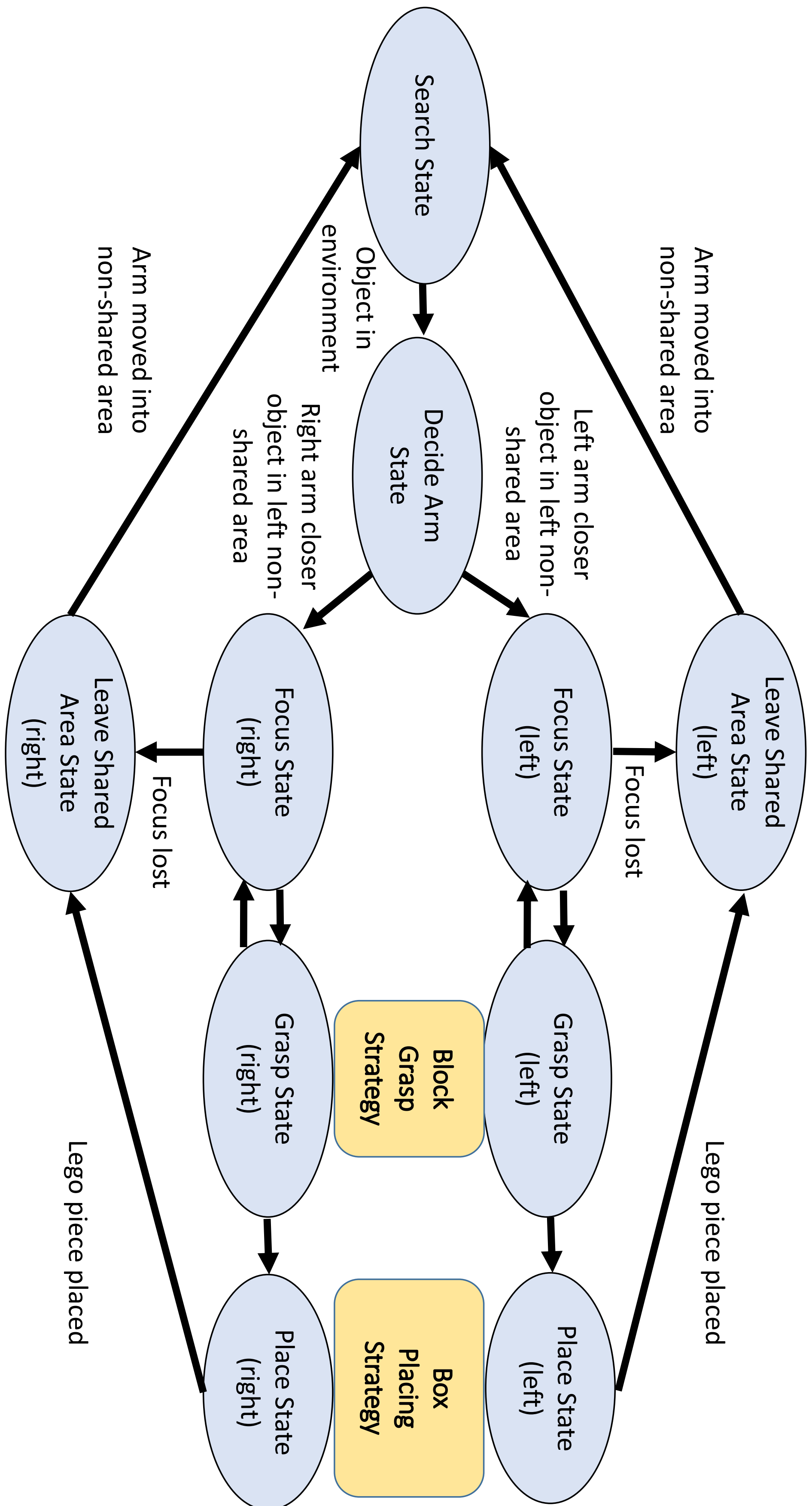


Figure 40: Multi-arm State Machine build from two single arm machines

7.2 DISCUSSION

Considering the results the project shows three main points to discuss.

First, the speed and performance evaluation of the robot during the demos indicates that the current working speed of the robot is slow compared to a human worker. However, this does not mean that the robot disqualifies as a replacement for human workers as it never get tired. A human might outperform the robot in the short run but the robot can work 24 hours a day without rest to obtain the same working performance per day. Furthermore, the motion speed of the robot is currently heavily depending on the sample-based motion planners in MoveIt!. Projects that work with IK only motion planning reach higher speeds on the cost of collision-awareness. That is why further developments in MoveIt! towards optimal path planning could lead to a large increase in performance. One project that currently targets this issue is the Stochastic Trajectory Optimization for Motion Planning (STOMP) integration into MoveIt! [16]. On the other hand, testing HRI research hypotheses is often a complex process where the speed of motion is only a small part of complete the evaluation. Consequently, in such use cases the current motion planning does not necessarily hinder the evaluation of a hypothesis.

Second, the demo scenarios received good feedback in our first presentation and can therefore be considered a promising approach. However, further studies are needed to find out the level of complexity that can be conveyed to the audience by something like Lego piece manipulation. The questions that arose on the seminar did for example often come from executives working in the electrical industry being concerned about the electro statically properties and the dexterity of the robot. Such things are hard to present with the current approach as the Lego pieces are larger than most electrical parts and do not require any additional safety mechanisms like antistatic bags.

Furthermore some restrictions of the current demos harm the appeal for the viewer. The Lego pieces need to lie flat on the table with the marker facing up, an artificial constraint whose need is not obvious for the viewer. The grasp angle is currently restricted to 90 degree from above shortening the general reach of the arm. As a result it sometimes seems that as the robot could grasp a piece but does not as it cannot grasp in the right angle. Last, the table and the boxes used in a variant of the sorting demo need to be on a fixed position and are not perceived by the robot breaking the general concept of a random environment. These restrictions are mainly because of time shortage during implementation and should be fixed in further projects.

Finally, the first framework evaluation illustrates some framework strengths. Due to the heavy code reuse a faster development process

can be generally expected. However, as this first evaluation is a really specific case, other evaluations are needed. The current test allows no conclusion about how steep the learning curve is and only allows limited inferences about the general adaptability of the framework as the modeled processes were similar. Nevertheless we believe in the framework as a good tool for an easier and faster prototyping process because it does not require deep knowledge about all the specific details involved in motion planning and perceiving.

8 FUTURE WORK

After exploring the different capabilities of the Baxter robot platform, we suggest that this exploration should now continue in an industry scenario. While our project showed basic low-level techniques that can be integrated in such an industry scenario, the future work in a concrete industry environment opens up many new research questions emerging from the nature of normal industry processes. For example, we can then evaluate the pros and cons and the most needed new functionalities for our current software framework. An industry scenario provides us with the opportunity to answer our existing HRI questions and to ask new ones. It allows us to explore, for example, a completely new kind of HRI in observing the social aspects arising during such an industry integration.

As a result of this considerations we propose a cooperation with a company that manufactures heavily specialized products in small scale as it has most likely a demand for the robot. Such a company is for example Dynamic Source Manufacturing (DSM) that expresses interest in a collaboration with us. They are an electronics manufacturing service provider, offering flexible, customized manufacturing solutions for their customers. Their service includes many electronic production capabilities like quick-turn prototypes or volume production with quality and reliability testing.

The manufacturing line of DSM includes an automated and manual stage. The automated stage encompasses assembling Surface Mounted Device (SMD) chips on an electrical board and is automatically controlled by industrial image processing machines (Figure 41).

Figure 41: A typical automated optical inspection machine for electronic boards

The manual stage consists of equipping the product with larger electrical parts by hand. This stage is missing the automated inspection that the first stage includes. This means it has two disadvantages compared to the automated stage. On the one hand, the failures introduced by manual work are not discovered directly but later in the process leading to more amount of work needed. On the other hand, it is impossible for the company to optimize the production process based on the typical errors that are found during this inspection process, because such data does not exist due to the expensiveness of manual data capturing. Introducing automated inspection into the manual stage has therefore the potential to increase the general manufacturing speed and quality. That is why we see a good application opportunity for the Baxter robot in this stage as the normal industrial inspection machines cannot handle the size of the manual parts and the inspection might need to include a turning and moving of the electrical component to get a better view on the added parts.

We could imagine the following scenario for the use of the robot in this stage. Baxter works co-located with the persons on the assembly line (Figure 42).

Figure 42: A possible workspace of Baxter at the DSM assembly line - the robot would inspect the boards on the left, close to the workers testing the functionality of the components.

It receives the components to inspect from the person that assembles it. After they hand the product over to Baxter, it inspects the product visually. It spots errors like a misalignment of the manually mounted parts, bad solder joints or a missing part and highlights failures on its display. Finally, it categorizes the products in error prone and flawless products for later manual control and correction. The company would benefit from such a scenario by having an inspection not only for the SMD stage but also for the manual stage. The application of the robot in this concrete scenario would also allow us to explore new research questions like:

- How the working process and the behavior of the workers itself changes through the presence of Baxter
- How Baxter's work can be integrated in as a non disruptive process for the human workers
- How and if it is possible to influence the working motivation of people through the appearance and gestures of the robot
- How well the camera systems are suited for visual inspection tasks
- How we can achieve a sufficient working speed

.

Besides of the introduction of the robot to industry scenarios further improvements of the framework are possible. Here, improvements of the motion planning are promising. First, the integration of an optimal motion planner like STOMP into MoveIt! can greatly increase the performance of the robot since trajectories looks more natural and are shorter, leading to a faster operation speed. Second, time parameterizing motions could be an interesting feature for HRI research. The difference between a gentle or shy slow motion towards a person and a fast more aggressive motion is a for example a good research field. This time parameterization should also be included into MoveIt! as it is a general feature that is useful for all robots. Third, the integration of a grasp framework for the pick actions of the robot would increase the adaptability of the manipulation tasks. Candidates for such a framework are MoveIt! SimpleGrasps[1] and GraspIt![2] offering extended grasping behavior.

Finally, an evaluation study of the HRI capabilities of the robot with is required. Techniques like the Godspeed questionnaire [3] are here promising approaches to gather first data as they can quickly reveal weak points in the perception. Furthermore, while the framework offers basic primitives of HRI capabilities more complex features should be introduced in the robot to tests its full potential. This includes a more expressive face, for example through more features like eyebrows or a nose and context awareness. The robot could for example raise the eyebrow when it is confused or it could wrinkle the nose to express anger about its recent performance.

1 https://github.com/davetcoleman/moveit_simple_grasps
2 http://www.cs.columbia.edu/~cmatei/graspit/

BIBLIOGRAPHY

[1] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.

[2] Shikhar Sharma Ashesh Jain and Ashutosh Saxena. Beyond Geometric Path Planning: Learning Context-Driven Trajectory Preferences via Sub-optimal Feedback. 2013. URL http://pr.cs.cornell.edu/coactive/.

[3] Christoph Bartneck, Dana Kulić, Elizabeth Croft, and Susana Zoghbi. Measurement Instruments for the Anthropomorphism, Animacy, Likeability, Perceived Intelligence, and Perceived Safety of Robots. *International journal of social robotics*, 1 (1):71–81, 2009.

[4] Joshua Bloch. How to Design a Good API and Why it Matters. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, pages 506–507. ACM, 2006.

[5] Mike Blow, Kerstin Dautenhahn, Andrew Appleby, Chrystopher L Nehaniv, and David Lee. The Art of Designing Robot Faces: Dimensions for Human-Robot Interaction. In *Proceedings of the 1st ACM SIGCHI/SIGART conference on Human-robot interaction*, pages 331–332. ACM, 2006.

[6] Barry W. Boehm. A Spiral Model of Software Development and Enhancement. *Computer*, 21(5):61–72, 1988.

[7] Timothy M Caldwell, Dave Coleman, and Nikolaus Correll. Optimal Parameter Identification for Discrete Mechanical Systems with Application to Flexible Object Manipulation. *arXiv preprint arXiv:1402.2735*, 2014.

[8] Karel Capek. *R.U.R. (Rossum's Universal Robots) (Penguin Classics)*. Penguin Classics, 3 2004. ISBN 9780141182087.

[9] L Peter Deutsch. Design Reuse and Frameworks in the Smalltalk-80 System. In *Software reusability*, pages 57–71. ACM, 1989.

[10] Carl F DiSalvo, Francine Gemperle, Jodi Forlizzi, and Sara Kiesler. All Robots are Not Created Equal: The Design and Perception of Humanoid Robot Heads. In *Proceedings of the 4th conference on Designing interactive systems: processes, practices, methods, and techniques*, pages 321–326. ACM, 2002.

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

[12] Jochen Heinzmann and Alexander Zelinsky. A Safe-Control Paradigm for Human–Robot Interaction. *Journal of Intelligent and robotic systems*, 25(4):295–310, 1999.

[13] Morton A Hirschberg. Rapid Application Development (rad): a brief Overview. *Software Technology News,(2: 1)*, 1998.

[14] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Prentice Hall, 3 edition, 7 2006. ISBN 9780321455369.

[15] Ashesh Jain, Brian Wojcik, Thorsten Joachims, and Ashutosh Saxena. Learning Trajectory Preferences for Manipulators via Iterative Improvement. In *Advances in Neural Information Processing Systems*, pages 575–583, 2013.

[16] Mrinal Kalakrishnan, Sachin Chitta, Evangelos Theodorou, Peter Pastor, and Stefan Schaal. STOMP: Stochastic Trajectory Optimization for Motion Planning. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 4569–4574. IEEE, 2011.

[17] Ian Lenz, Honglak Lee, and Ashutosh Saxena. Deep Learning for Detecting Robotic Grasps. *arXiv preprint arXiv:1301.3592*, 2013.

[18] Robert C Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education, 2008.

[19] Fred Gibbson Micah Siegel. Rethink Robotics - Finding a Market, 2013.

[20] Masahiro Mori, Karl F MacDorman, and Norri Kageki. The uncanny valley [from the field]. *Robotics & Automation Magazine, IEEE*, 19(2):98–100, 2012.

[21] David Nuñez, Marco Tempest, Enrico Viola, and Cynthia Breazeal. An Initial Discussion of Timing Considerations Raised During Development of a Magician-Robot Interaction.

[22] Jia Pan, Sachin Chitta, and Dinesh Manocha. FCL: A General Purpose Library for Collision and Proximity Queries. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 3859–3866. IEEE, 2012.

[23] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. ROS: an

open-source Robot Operating System. In *ICRA workshop on open source software*, volume 3, page 5, 2009.

[24] Kari Rainio. *ALVAR - A Library for Virtual and Augmented Reality User's Manual (v2.0)*. VTT, 2012.

[25] Rolf Rannacher. *Einführung in die Numerische Mathematik*. 2006. URL http://numerik.uni-hd.de/~lehre/notes/num0/numerik0.pdf.

[26] Rethink Robotics. Baxter Robot Product Brochure, 2013.

[27] Mark E Rosheim. *Robot Evolution: the Development of Anthrobotics*. John Wiley & Sons, 1994.

[28] Ruben Smits, H Bruyninckx, and E Aertbeliën. KDL: Kinematics and Dynamics Library. *Avaliable: http://www. orocos. org/kdl*, 2011.

[29] Ioan A. Sucan and Sachin Chitta. Moveit! URL http://moveit.ros.org.

[30] Ioan A. Şucan, Mark Moll, and Lydia E. Kavraki. The Open Motion Planning Library. *IEEE Robotics & Automation Magazine*, 19(4):72–82, December 2012. doi: 10.1109/MRA.2012.2205651. http://ompl.kavrakilab.org.

[31] Andrew S. Tanenbaum. *Moderne Betriebssysteme*. Pearson Deutschland GmbH, München, 3 edition, 2009. ISBN 978-3-827-37342-7.

[32] G. Teschl and S. Teschl. *Mathematik für Informatiker: Band 2: Analysis und Statistik*. Mathematik für Informatiker. Springer-Verlag Berlin Heidelberg, 2007. ISBN 9783540724520.

[33] Agisilaos G Zisimatos, Minas V Liarokapis, Christoforos I Mavrogiannis, and Kostas J Kyriakopoulos. Open-Source, Affordable, Modular, Light-Weight, Inderactuated Robot Hands. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2014.

colophon

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

http://code.google.com/p/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/

Final Version as of August 27, 2014 (`classicthesis` version 4.1).

SELBSTÄNDIGKEITSERKLÄRUNG

Ich versichere, dass ich die vorliegende Arbeit (bei einer Gruppenarbeit: den entsprechend gekennzeichneten Anteil der Arbeit) selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich erkläre hiermit weiterhin, dass die vorgelegte Arbeit zuvor weder von mir noch von einer anderen Person an dieser oder einer anderen Hochschule eingereicht wurde.

Darüber hinaus ist mir bekannt, dass die Unrichtigkeit dieser Erklärung eine Benotung der Arbeit mit der Note „nicht ausreichend"zur Folge hat und einen Ausschluss von der Erbringung weiterer Prüfungsleistungen zur Folge haben kann.

Calgary, July 2014 - September 2014

Tim Steuer