Julia Nitsch, Bsc.

Industrial Grasping

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieurin

Master's degree programme: Telematics

submitted to

Graz University of Technology

Supervisor

Ass.Prof. Dipl.-Ing. Dr.techn. Gerald Steinbauer

Institute for Softwaretechnology

Graz, March 2016

This document is set in Palatino, compiled with pdfLaTeX2e and `Biber`.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis dissertation.

Graz, ____________________ ____________________

Date Signature

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

Graz, ____________________ ____________________

Date Signature

Acknowledgments

First of all, I would like to thank my supervisor, Gerald Steinbauer, for supporting me throughout my Master studies. In particular I would like to thank him for the instructive discussions on any robotic related topics and for his patience and motivation in supervising my Master thesis.

Further I would like to thank the stuff from the Institute for Software Technology for their support in scientific and administrative matters. Many thanks are owed to Clemens Mühlbacher for helpful discussions about tricky implementation details.

Furthermore thanking the company Incubed IT for supporting this thesis.

Special thanks go to my friends and team colleagues from the RoboCup Rescue Team TEDUSAR for supporting me with useful tips and tricks throughout the past two years, or even longer.

Moreover I am thanking my boyfriend a lot, for spending so much time in seemingly never-ending discussions on any issues and topics occurring in our studies. Further for supporting me so much in everything, including moving my things through whole Middle Europe.

Finally, I would like to express my gratitude to my family for supporting me throughout my whole life in any projects I started.

Abstract

Within this work a robotics application in an Industry 4.0 domain is presented, namely a fully autonomous order picking system. It autonomously plans actions to perform the order picking task. The task is performed in an environment designed for human workers. Items are stored in boxes, which are located at different levels on shelves. The order tells about the amount and type of items which should be picked. Depending on this order the box, containing the requested item type, is pulled out of the level. Items are picked out of the box, placed at a delivery box and finally the box is returned to its level. Then the next item type is processed. This continues until the order is finished. We refer to this whole process as *order picking*.
Due to the non automatized environment, objects cannot be expected to be on the exact position, where they should be. So this systems needs to have perception capabilities to detect objects. Further it needs the ability to plan and execute collision free movements to manipulate objects. And last but not least it needs the capability to grasp objects. This includes collision aware grasp planning and grasp execution.
The system is based on a 3-TIER architecture. The planning layer uses an artificial intelligence (AI) planner to generate a list of skills the robot should execute. This planner parses Planning Domain Definition Language (PDDL) descriptions of the system capabilities (further on called skills), as well as the current state of the environment and the goal state. The planner outputs a list of skills, the robot needs to execute in order to achieve its goal. Skills are composed of skill primitives. These primitives can perform perception, manipulation, grasping tasks or any combination of those. Special attention is paid to calculating online collision aware grasps. Therefore the framework GraspIt! is used and extended. For planning and executing motions the MoveIt! framework is used. All skill primitives are implemented as Robot Operating System (ROS) actions.
This system can be easily executed on different robotic hardware systems.

We showed a proof-of-concept implementation using the two arm robot Baxter from Rethink Robotics.

Contents

Acknowledgements v

Abstract vii

1 Introduction 1
- 1.1 Motivation 2
- 1.2 Goals and Challenges 3
- 1.3 Contribution 4
- 1.4 Outline 5

2 Formal Problem Description 7
- 2.1 Environment 7
- 2.2 Use Case 11

3 Related Research 13
- 3.1 High-Level Planning 13
- 3.2 Grasping 16
- 3.3 Perception 19
- 3.4 Integrated Systems 20

4 Prerequesites 23
- 4.1 ROS 23
- 4.2 MoveIt! 26
- 4.3 GraspIt! 28
- 4.4 PCL 29
- 4.5 Baxter 30
  - 4.5.1 ROS support 32
  - 4.5.2 Sensors, Inputs and Outputs 32
  - 4.5.3 Arm controlling via ROS interfaces 33

5 System Overview **37**
5.1 Overview of the architecture for the robotic order picking . . 37
5.2 Planning Layer 38
5.2.1 Domain 39
5.2.2 Problem 41
5.2.3 Planner 43
5.3 Executive Layer 45
5.4 Behavioral control 47
5.4.1 Detect Handle 49
5.4.1.1 movement 50
5.4.1.2 detection 50
5.4.2 Grasp Handle 52
5.4.2.1 Initial Measurement 53
5.4.2.2 Visual Servoing 54
5.4.2.3 Validation 55
5.4.2.4 Local Recovery 55
5.4.3 Move to Support Pose 56
5.4.3.1 Calc. Pose 57
5.4.4 Pull Box 59
5.4.5 Move Box 61
5.4.5.1 Calculate Movement 62
5.4.5.2 Movement Validation 63
5.4.5.3 Grasp Validation 63
5.4.6 Deliver Box on Tray 64
5.4.6.1 Turn Box 64
5.4.6.2 Release Handle 65
5.4.7 Detect Item 65
5.4.7.1 Detect Box 68
5.4.7.2 Move Over Box 68
5.4.7.3 Item Detection 71
5.4.8 Grasp Item 73
5.4.8.1 Calculate Grasping Points 73
5.4.8.2 Pick Item 75
5.4.9 Deliver Item 75
5.4.10 Deliver Box On Level 76
5.4.10.1 Push Box Into Level 77
5.5 Layout for 3-TIER high-level architecture as HSM 79

6 Implementation **83**
6.1 Environment . . . 83
6.2 High Level Node . . . 83
6.2.1 Class Structure . . . 84
6.3 Skill Primitives and System Functionalities . . . 86
6.3.1 Perception Module . . . 88
6.3.1.1 Detect Handle Node . . . 89
6.3.1.2 AR Tag Detection Node . . . 90
6.3.1.3 Detect Item Node . . . 90
6.3.1.4 Detect Box Node . . . 91
6.3.1.5 Head Node . . . 95
6.3.2 Arm Movements Module . . . 98
6.3.2.1 Move To Pose Node . . . 98
6.3.2.2 Move Over Box Node . . . 99
6.3.2.3 Move To Level Node . . . 99
6.3.2.4 Move To Support Pose Node . . . 99
6.3.3 Grasping Module . . . 99
6.3.3.1 Grasp Handle Node . . . 100
6.3.3.2 GraspIt! . . . 102
6.3.3.3 Pick and Place Node . . . 104
6.3.4 Manipulation Module . . . 107
6.3.4.1 Pull Box Node . . . 107
6.3.4.2 Move Box Node . . . 108
6.3.4.3 Deliver Box On Level Node . . . 108
6.3.4.4 Deliver Box On Tray Node . . . 108

7 Evaluation **109**
7.1 Skill MoveBoxFromLevelToTray . . . 109
7.1.1 detectHandle . . . 110
7.1.2 graspHandle . . . 111
7.1.3 moveToSupportPose . . . 113
7.1.4 pullBox . . . 114
7.1.5 moveBox . . . 115
7.1.6 deliverOnTray . . . 116
7.2 Skill graspItem . . . 117
7.2.1 detectItem . . . 117
7.2.2 graspItem . . . 118

7.3 Skill moveBoxFromTrayToLevel 119
7.3.1 pullBox 119
7.3.2 deliverOnLevel 120
7.4 Box Detection 120
7.5 Grasp Item 122
7.5.1 Left Middle Item Pose 124
7.5.2 Right Middle Item Pose 125
7.5.3 Upper Left Item Pose 126
7.5.4 Lower Left Item Pose 127
7.5.5 Upper Right Item Pose 128
7.5.6 Lower Right Item Pose 129
7.5.7 Upper Middle Item Pose 130
7.6 Parallel Jaw Gripper in GraspIt! 131

8 Conclusion **133**

9 Future Work **135**

Bibliography **137**

List of Figures

1.1 Manual Order Picking Station 3

2.1 Single shelving Unit S_{Ui} on Plane P 8
2.2 Tray R_a on Plate P 9
2.3 Transport Box B_i 10
2.4 Robot R_o With Two Robotic Arms Mounted 11

4.1 Concept of Topics Within ROS 24
4.2 Concept of Services Wssithin ROS 25
4.3 Actions in ROS 25
4.4 Planning Scene Monitor in MoveIt! 26
4.5 Overall Concept of MoveIt! 27
4.6 Setup Assistant of MoveIt! 28
4.7 PCL Point Cloud Viewer 30
4.8 PCL Sample Point Cloud 31
4.9 PCL Cluster of Sample Point Cloud 31
4.10 Overview Baxter's SDK 32
4.11 Asus Mounted on Baxter's Head 33
4.12 Overview Baxter's Sensors 34
4.13 Detailed View of Baxter's Endeffector. 35
4.14 Overview of JTAS From Baxter's SDK 35

5.1 Conceptional Overview of 3-TIER Architecture 38
5.2 Types, Classes, Predicates, Functions in Order Picking Domain 40
5.3 Example for Initial State 43
5.4 Example for Goal State 44
5.5 Skill moveBoxFromLevelToTray 47
5.6 Skill moveBoxFromTrayToToLevel 48
5.7 Skill graspItem 49

5.8 Detect Handle Skill Primitive 50
5.9 Detection Part of Detect Handle Skill primitive 51
5.10 Pinhole Camera Model 52
5.11 Poses Detected by DetectHandle 53
5.12 Grasp Handle Skill Primitive 54
5.13 Grasp Handle Visual Servoing Routine 55
5.14 Move to Support Pose Skill Primitive 56
5.15 Support Pose with Respect to Camera Frame 57
5.16 Baxter Maniputlating Box 60
5.17 Pull Box Skill Primitive 60
5.18 Move Box Skill Primitive 61
5.19 Move Box Trajectories From Level To Tray 63
5.20 Deliver Box on Tray Skill Primitive 64
5.21 Turn Box on Tray 66
5.22 Release Box's Handle on Tray 67
5.23 Detect Item Skill Primitive 67
5.24 Inspect Box Poses 71
5.25 Inspect Box Poses (Detailed) 72
5.26 Baxter Inspecting Box 72
5.27 AR Tag 73
5.28 Grasp Item Skill Primitive 74
5.29 GraspIt! - Collision Aware Grasp Planning 75
5.30 Deliver Item Skill Primitive 76
5.31 Grasp Item Skill Primitive 77
5.32 Pushing Box Into Level 78
5.33 General HSM for 3-TIER Architecture 80
5.34 General Skill of the PlanExecution State Machine 81

6.1 Objects in Current Setup 85
6.2 Implementation of Top Level State Machine 86
6.3 Overview of Functionalities Skill Primitives Use 89
6.4 Collsion Scene With Pubished Box 90
6.5 Detected Item 92
6.6 Sequence Diagram Detect Item Node 93
6.7 Box Model in Blender 93
6.8 Poses to Verify Box Detection 97
6.9 Collision Scene Sensed and Real 98

6.10 Sequence Diagram Grasp Handle Node 101
6.11 GraspIt! ROS Nodes 105
6.12 Grasp Visualization in RViz 107

7.1 Handle Detection Evaluation 110
7.2 Grasp Handle 112
7.3 Top View Support Pose 113
7.4 Pull Box 114
7.5 Moving the Box 115
7.6 Moving the Box 116
7.7 Item Detection 117
7.8 Box Detection Environmental Setup 121
7.9 Evaluation Box Detection Pose 1 - Pose 7 122
7.10 Evaluation Box Detection Pose 8 - Pose 15 123
7.11 Item Poses Inside the box 123
7.12 Grasp Item Evaluation Left Middle Pose 124
7.13 Grasp Item Evaluation Right Middle Pose 125
7.14 Grasp Item Evaluation Upper Left Pose 126
7.15 Grasp Item Evaluation Lower Left Pose 127
7.16 Grasp Item Evaluation Upper Right Pose 128
7.17 Grasp Item Evaluation Lower Right Pose 129
7.18 Grasp Item Evaluation Upper Middle Pose 130

List of Tables

5.1 General Skill 39
5.2 Skills for Order Picking Problem 41
5.3 Skill primitive composition 46

7.1 Evaluation Grasping Item - Left Middle Pose 124
7.2 Evaluation Grasping Item - Right Middle Pose 125
7.3 Evaluation Grasping Item - Upper Left Pose 126
7.4 Evaluation Grasping Item - Lower Left Pose 127
7.5 Evaluation Grasping Item - Upper Right Pose 128
7.6 Evaluation Grasping Item - Lower Right Pose 129
7.7 Evaluation Grasping Item - Upper Middle Pose 130
7.8 Baxter Gripper and Schunk Gripper in GraspIt! 132

1 Introduction

Industry 4.0. This is one of the keywords, when we talk about the next level of production. Industry 4.0 stands for the 4^{th} industrial revolution. It promises cost reduction through self organizing and self optimizing processes. It addresses the needs of high quality products, which are also highly customized, but still ready for mass production. These are products where the customer can select the base product with different options for specific parts of the product. It would not make sense to store all possible combinations of items for the product. But to assemble the base product on demand, when the customer orders that specific variant of the base product. Expressions which come along with the 4^{th} industrial revolution are: *Internet of Things* or *Big Data*. All these technologies are used by one center part of Industry 4.0, namely the *Smart Factory*.
Smart Factories are intelligent and connected with other factories and customizers. So these connections affect the supply chain as well as the product line. Through interconnection with other factories, these will be self organizing systems. This leads to optimized processes and as a consequence to cost reduction. The connection to customizers promises customized products within a mass production. In order to cope with this new type of products, robots will assist workers. They will no longer be only in an automatized process but they will assist human as coworkers. For example they can assist in lifting heavy goods, they can be used in too dangerous zones for humans, or they can support workers in monotonous working steps. If robots can overtake such tasks completely autonomously, human forces could be used for more demanding tasks.
Within the next section we talk about the motivation, the goals we achieve and their challenges. Further we give a brief overview over the thesis and the approaches we contributed to.

1.1 Motivation

Our work contributes to the field of smart factories by developing an assisting robotic order picking system. This is a new generation of industrial robotics, where robots can operate in human environments. In a warehouse system items can be be stored in larger transport boxes. The transport boxes again can be in shelves to save space. So if a specific item should be picked, first the transport box needs to be pulled out of the shelf and then the item can be picked and delivered. This procedure is called order picking. For items with a moderate frequency this type of picking is usually done by hand, which is a monotonic and time consuming task. In our scenario we tend to automize that task.
Our approach will be used in the production of customized products. When the customizer orders a product, the robot fetches the right parts at its working station and places it at a delivery point. The picked parts can then be delivered to the next processing station where either human workers or robots further assemble the single parts, or even pack it for delivering the good.
One can argue that those systems already exist in an automatized process. But there are two main, novel aspects to pick out, which makes this system such an interesting Industry 4.0 application. First, the robot can work and operate within an environment designed for humans. It operates fully autonomously and can adopt to changes in the environment. Secondly, there is no cage around the robot. A human can safely work side by side with the robot without being in danger.
In Figure 1.1 a manual order picking station is shown. A worker picks items which are stored in transport boxes at his working station, or delivered by a robot shuttle. The worker places the picked item in a transport box and hands the box over to another robot. Our approach can be used in this kind of scenarios. It can take over the tasks of the worker. This worker can now be used for more challenging tasks.

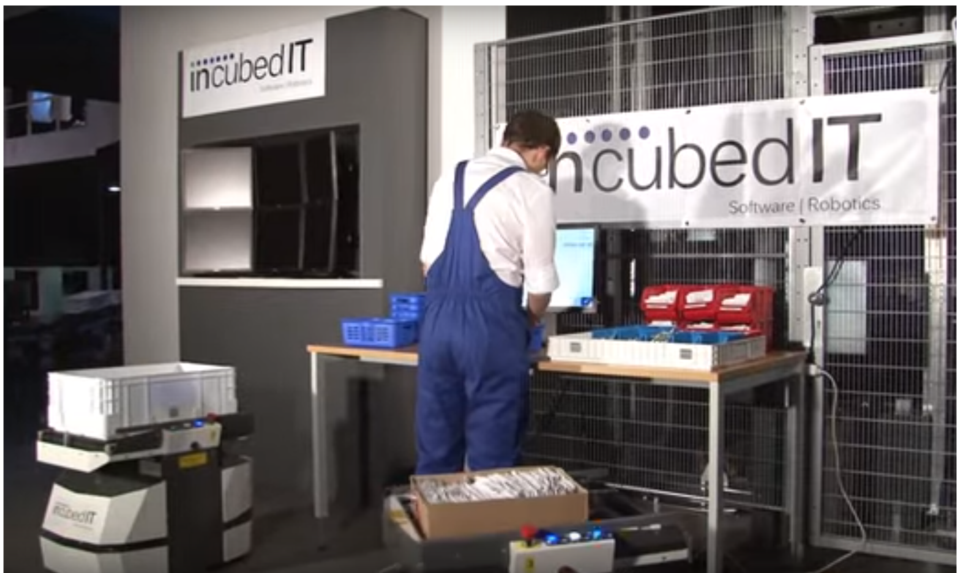


Figure 1.1: This figure shows a manual order picking station. ©2016 INCUBED IT - AUSTRIA

1.2 Goals and Challenges

The overall goal is to develop a proof of concept application to show autonomous order picking in a humanly adequate warehouse setup with a two arm robot. This work focuses mainly on these aspects:

1. Design of **modular system architecture** : A 3-TIER architecture is used to keep the system as modular as possible.
2. **Manipulation of transport boxes**: This deals with grasping the handle of the transport box and pulling it with a single arm. But this also includes two arm manipulation, when moving the box.
3. **Grasping of items**
   This includes online grasping point calculation with collision awareness. Because grasps are calculated online, the objects need not be known in advance. Only the pose and its shape need to be estimated, as well as the collision scene around the object, which should be grasped.

4. **Perception**
   The perception pipeline is one key issue in the system. We need to percept the environment to detect obstacles, which should be avoided during arm movements and manipulation of objects. Further transport boxes need to be detected first before they can be manipulated and also the items pose need to be precisely determined before it can be grasped. Large errors in the perception pipeline would lead to unsuccessful manipulation and grasping tasks.

The design itself is challenging to keep it modular and also portable to different hardware setups. Further the definition of skills and skill primitives need to be balanced in a way, that primitives can be reused in different skills but also not too granular to produce a huge overhead in programming skill primitives.
Problems in manipulation and grasping arise due to various reasons. The main problems in manipulating and grasping object is the inaccuracy of the used robot Baxter. So it happens quite often that actions are planned correctly, but they fail because Baxter is not able to perform the computed movements. So the design of recovery behaviors is needed. Through these recovery behaviors many actions which fail at the first time, could be successfully completed after a recovery and another trial.
Planning collision aware grasps ensures at least during the picking itself the robot does not collide with the environment. At the first glance it seems to be an overhead but in this way we ensures, if the grasp is precisely executed, that the robot is able to grasp the object.

1.3 Contribution

In this work we present a proof of concept implementation of autonomous order picking that can be easily integrated in industry 4.0 applications. We show the whole planning queue as well as a system for the successful execution of the planned tasks on a physical system.
Further we show online grasp planning with collision awareness. We point out that no predefined grasps are used or selected. In order to plan the online grasps, we extend the GraspIt! framework. We include a parallel jaw

gripper model in the framework. In order to proove this model we added Baxter's parallel gripper and the Schunk WSG 50 gripper to the framework. We also implement the physical model for evaluating grasps on different objects. Further we make available a ROS wrapper for GraspIt! including an Eigengrasp planning interface. This implementation is developed and tested under Ubuntu 12.04 and ROS Hydro.

1.4 Outline

The following sections are arranged as follows. In Chapter 2 we introduce the order pricking problem in a formal way. Further we show a use case for a better understanding of the task. In the Chapter 3 we discuss related research in that field. This includes the field of high-level planning, grasping and perception. In the end of this section we summarize already existing systems and similar approaches in literature. Because this system includes many well investigated problems, we extend some frameworks to concentrate on the goals described in the section 1.2 before. This prerequisites are presented in Chapter 4. This includes a robotic framework, as well as existing approaches for motion planning and execution. Following we describe existing grasping and perception frameworks. At the end of this chapter we describe also the robotic platform we use for our system. In Chapter 5 we outline the design of the whole system and all design decisions taken. The different parts of our system are described in detail. This is followed by the description of the implementation in Chapter 6. Here the implementation is described using class diagrams. The system is evaluated and the results are presented in Chapter 7. The last parts include the conclusion of the thesis in Chapter 8 and outlook for future work in Chapter 9.

2 Formal Problem Description

Within the the first section 2.1 of this chapter we define all objects which appear in the environment. Further the used robot is defined, with a special concentration on its robotic arms. Primarily this section contains all physical objects, but also places and functions which are used. For a better understanding of the problem a typical use case is described in the section 2.2.

2.1 Environment

Given is an empty three dimensional (3D) space E containing a single floor plane P. A 3D pose is assigned to each object within E through the function $\mathbb{P}$. Each pose is a 3D pose given in the world frame.

A set of shelving units is defined as $\mathbb{S}_U = \{S_{U0}, S_{U1}, \cdots, S_{Un}\}$. All units have a predefined pose $p_{S_{Ui}} = \mathbb{P}(S_{U_i})$. The number of shelving units is defined as $s_u = |\mathbb{S}_U|$. The position of a shelving unit is given through the coordinates on P (so the z-coordinate will be 0). The orientation is defined as the normal to the front of the shelving unit $\mathbf{n}_i$. S_{Ui} has properties of width w_i, height h_i and depth d_i, which describes the geometric shape of the unit. Further each unit has mounted equal shelves in different heights. A specific height is defined as $h_{i,j}$, where i identifies the shelving unit S_{Ui} and j describes the height in meters of the top edge of the shelve. The total number of shelves of one shelving unit S_{Ui} is defined as $\#s_i = S_{Ui} \rightarrow \mathbb{N}$. The amount of shelves of a single unit S_{Ui} is defined as s_i. Further a set of levels is defined as $\mathbb{L} = \{L_{0,0}, \cdots, L_{0,n}, L_{1,0}, \cdots, L_{1,m}, \cdots, L_{j,0}, \cdots L_{j,r}\}$. The first index identifies the shelving unit and the second index belongs to a single shelf within the unit. The total number of levels is defined as $\#L = \sum_{i=0}^{s_u-1} s_i$. In Figure 2.1 a single shelving unit S_{Ui} is shown.

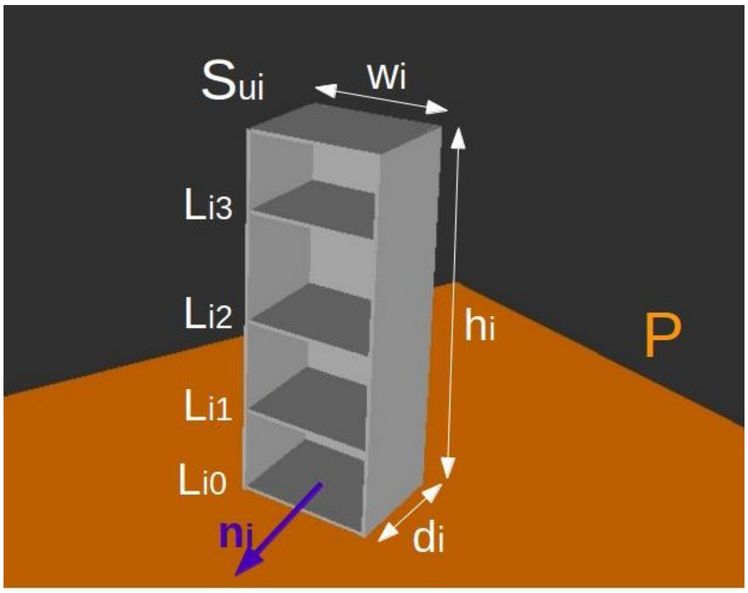


Figure 2.1: Single shelving unit S_{Ui} mounted on the plane P with 4 shelves mounted on the levels L_{i0}, L_{i1}, L_{i2} and L_{i3}. The geometric shape is described through w_i, d_i, and h_i. The orientation is described through $\mathbf{n}_i$.

Further E contains a single Tray R_a. The tray has a fixed pose $p_{R_a} = \mathbb{P}(R_a)$, where the top plate is mounted parallel to P. So the orientation is defined through the normal on the top plate n_a. The tray itself is defined through the width w_a and depth d_a of the top plate, and the height h_a in which the plate is mounted. In Figure 2.2 the tray R_a is shown.

A set of transport boxes is defined as $\mathbb{B} = \{B_0, B_1, \cdots, B_n\}$. Each box is described through a minimum bounding box with width w_b, depth d_b and height h_b. The starting pose of each transport box is uniquely identified through a level $L_{i,j}$. Each box has a handle on its' front ends. The orientation of a box is the normal $\mathbf{n}_b$ on its front end, which is nearer to the opening of the containing shelving unit S_{Ui}. The number of transport boxes is limited

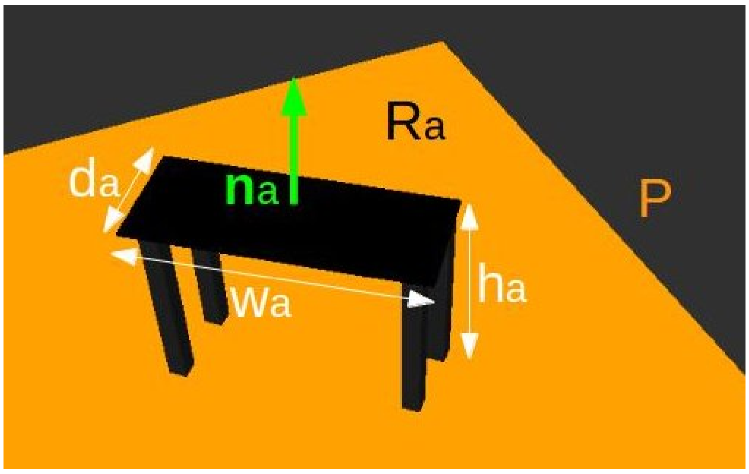


Figure 2.2: The dimensions of the top plate of the tray R_a are given with the depth d_a, the width w_a and the height of R_a is h_a. The orientation is described through the normal of the top plate $\mathbf{n}_a$.

due to the number of levels $\#B \leq \#L$. In Figure 2.3 a transport box B_i is shown.

Furthermore a set of item types is defined as $\mathbf{T} = T_0, T_1, \cdots, T_n$, where the number of item types is limited through the amount of transport boxes $\#T \leq \#B$.A set of items is defined as $\mathbb{I}_t = I_{t0}, I_{t1}, \cdots, I_{tn}$. Each item is assigned to **one** item type.

A robot R_o is defined as a subset of $\mathbb{E}$. A set of kinematic links $\mathbb{L} = \{L_0, L_1, \cdots, L_n\}$, a set of kinematic joints $\mathbb{J} = \{J_0, J_1, \cdots, J_n\}$ and a set of gripping devices $\mathbb{G} = \{G_0, G_1, \cdots, G_n\}$ is given. A gripping device G_i has two control parameters: *open* and *close*. Additionally the robot has assigned a predefined pose $p_r = \mathbb{P}(R)$, which is defined through the first link L_0 . A robotic arm A_i is defined as a kinematic chain of kinematic links which are connected with joints. The last link of a chain is called end effector. The pose of the end efffector is given thorugh the kinematic chain. A relation between links and joints is defined as $\mathbb{C} : \mathbb{J} \to \mathbb{L} \times \mathbb{L}$. The last link L_i of A_i can be a gripping device, $L_i \in (\mathbb{L} \cap \mathbb{G})$. A non empty set of A_i is mounted on R_o in a way that one base link (e.g L_0) is shared among the kinematic chains. (see Figure 2.4). A non empty set of sensors $\mathbb{S} = \{S_1, S_2, \cdots, S_n\}$ is

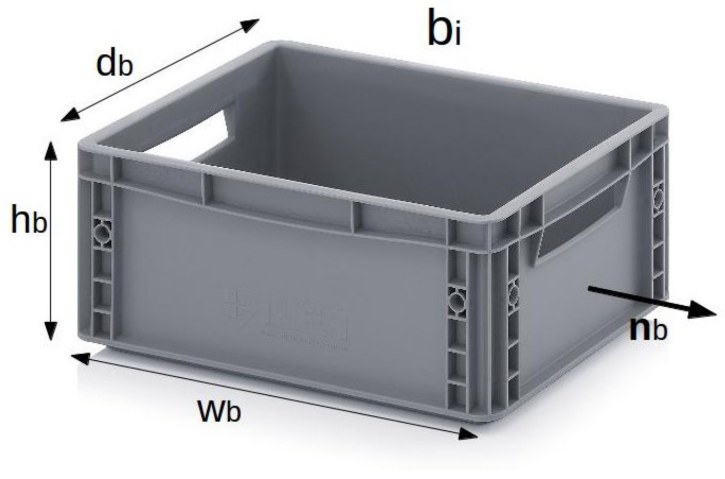


Figure 2.3: Transport box B_i, with dimensions w_b, d_b and h_b and orientation $\mathbf{n}_b$. Source `http://www.auer-packaging.at/images/products/huge/eurobehaelter-geschlossen-AUER--eg43-17.jpg`

attached on the robot(e.g. cameras, infrared sensors, sonar sensors, $\cdots$).

We assume a consistent warehouse management. This means that a transport box B_i contains a subset of $\mathbb{I}_t$, which are all of the same item type T_i. In this warehouse management the possible location of a transport box B_i is $loc_i = \mathbb{L} \cap \{R_o\} \cap \{R_a\}$. The transport box can either be on a defined level L_{ij} or the robot can manipulate it or it is arranged on the tray and the robot manipulates its content. Until the robot has not manipulated a transport box, its place is unique defined by a level $L_{i,j}$. So $\#B$ must be limited by $\#L$, as mentioned before. If the robot finishes manipulating the content of B_i, it returns the box to its previous level $L_{i,j}$
Further a management system has knowledge about how many transport boxes are in the warehouse and where to find them (on which level $L_{i,j}$ they are). It also holds information about what kind of item type and how many items are stored in each box. Further a special transport box D is defined which is not member of $\mathbb{B}$ ($D \notin \mathbb{B}$). This box stores any items which need to be delivered. So it can hold items of different item types. The pose of D is defined through the robot. It is near to R_o, so that it can place any selected items into D.

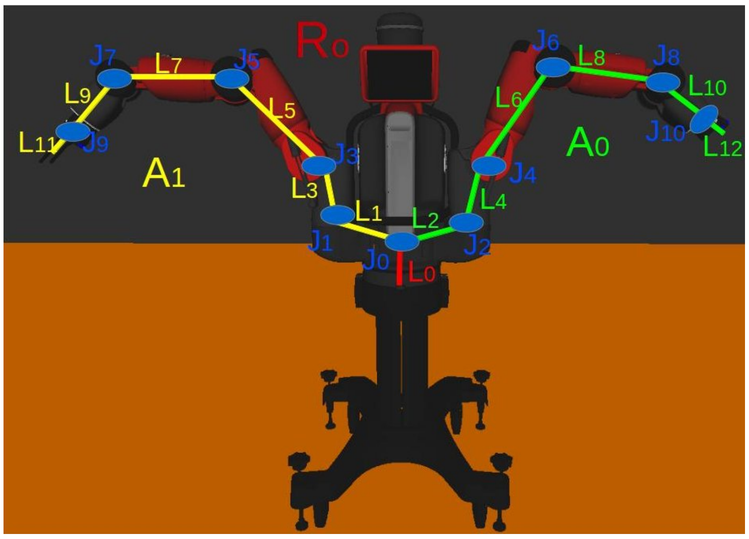


Figure 2.4: Two robotic arms A_1 and A_2 are mounted on the robot R_o. The shared base link is L_0. Arm A_1 is a kinematic chain of L_0, J_0,L_1, J_1,L_3, J_3,L_5, J_5, L_7, J_7,L_9, J_9 and L_{11}. Arm A_0 is a kinematic chain of L_0, J_0,L_2, J_2,L_4, J_4,L_6, J_6, L_8, J_8 and L_10. On arm A_1 the end effector link L_{11} is a gripping device, whereas on arm A_0 the end effector L_{12} is not a gripping device.

2.2 Use Case

In this section a typical usecase is described for a bette understanding.

1. **order**
   The order $\mathbb{O}$ is defined as set $\mathbb{O} = \{o_0, o_1, \ldots, o_n\}$. Where o_i is defined as tuple $< n_i, T_i >$. This tuple defines n_i items of type T_i . So the order $\mathbb{O} = \{o_0, o_1\}$, $o_0 = < 2, T_i >$, $o_1 = < 4, T_j >$ means: "Get two items of type T_i and four items of type T_j".
2. **warehouse management system**
   The system provides the information on which levels the transport boxes are, which contain the wanted item types. Further it has to check that enough items are in the transport box or if another transport box should be chosen which contains the same item type. It provides the information: " two items of T_i are in box B_i on $L_{r,t}$ and four items of type T_j are in box B_j on $L_{s,k}$"

3. **delivering the good**
   The robot has to move towards the shelving unit S_{Ur}, which stores the first transport box B_i. Next it must lift B_i out of S_{Ur} and bring it to the manipulating place, the tray R_a. Then it has to pick two items out of b_i and place it into the delivering box D. Further it returns the transport box B_i to its initial level $L_{r,t}$ and moves towards the next shelving unit S_{Us}. Again it takes B_j from the level $L_{s,k}$ and puts it on R_a. Now it picks four items and places it into D. The robot returns B_j too. The last step is delivering D.

3 Related Research

3.1 High-Level Planning

In literature numerous works about high-level task planning for robotic systems exist that make use of a set of simpler system capabilities. This system capabilities are called skills. Dividing a complex task in such skills has multiple benefits like a good software portability and easier reuse of software capabilities.
Pederson et al. shows in [1] this concept. Here single complex task is divided into a sequence of smaller subtasks. These subtasks are the so called skills. These are the capabilities the system has. The authors call them the "fundamental building blocks"[1], which can be easily ported to other robotic systems. Further any other complex task can be executed easily without reprogramming the whole system, but with reordering skills only. The authors define a single skill containing a precondition, which checks if the skill can be executed. This system has an execution monitoring which surveys the outcome of each primitive. The evaluation of the successful outcome of a skill is similar to our approach, although the description of a single skill is not that abstract than mine.
So this works describes how intuitive robots can be reprogrammed by also unskilled persons. This indicates a good abstraction of the skill layer. In this paper [2] the authors describe the integration of robot skills on different robot platforms and how portable a system consisting of skills is. Pederson et al. mention that a complex task can be performed, even if having only a very basic set of skills. A skill is here described as a combination of primitives, but the authors do not give a strict definition on a skill primitive. Again a program is defined as sequence of skills, which can be planned but can also be generated by a user. Within [2] the advantage of modularity and the abstraction of tasks is clearly shown.

In both papers [1], [2] they concentrate on skills and their benefits for easily reprogrammable robots. This work [3] of Rodiva et al. is about the division of a complex task into different layer. They introduce into a 4-TIER structure with the same idea of abstraction for skills and skill primitives as in the previous papers. The layers are structured as follows: the lowest level is called device layer which includes the hardware abstraction, next is the primitive layer which contains actions and perceptions, followed by the skill layer for object abstraction and the last layer is the task layer which is for planning. This structure ensures the portability through the lower two layers and the abstraction for task planning through the higher two layers. This works uses the abstraction also for easy human-robot interaction.
So all these papers show a clear division of tasks, skills and primitives and their portability and how easy new complex tasks can be executed. But the focus is on human-robot interaction. The human in the loop defines a new task through reordering skills. The work of Huckaby et al. [4] focuses on artificial intelligence (AI) planning. Within this paper they want to automate task modeling and the execution performed by a robot. So a taxonomy defines the skill primitives and robot specific skills. Higher level tasks can be build using the knowledge of this taxonomy. The authors see the advantage of AI planning methods in reducing the constraints on users and in improving the automation process. They divide the problem into a model space, which represents all the capabilities of a robot and a process space, which describes the environment and the goal. This separation refers to the planning and problem domain of the Planning Domain Definition Language (PDDL) [5]. They proposed PDDL for AI planning because it is a standard language for planning tasks. Within this work the advantages of not having a human in the loop were pointed out. Such 'programmers' need also deep knowledge on the system and intensive training. Further humans need not stick to formal specifications like preconditions or effects. So there is no guarantee if the plan succeeds or if it is the best one. Within this work information about skills and their primitives is gained from an ontology. In our work we use an inherent knowledge about skill primitives and their composition in skills. They showed that PDDL is a good choice for an abstract planner. They use trajectory constraints to ensure the temporal order of subgoals (specific in PDDL3 [6]).
So within [3] the authors focused on the high-level itself and not on skills or their primitives. But they assumed that all skills will always succeed. But we

operate within a real environment, with real hardware. Some actions will fail and so we need at least any strategy to detect failures and perform recovery. Within [7] Okada et al. introduced a high-level which controls systems in real environment using an advanced hardware/software architecture. They transfer the high-level description from an AI planner to a behavioral state machine. The detection of a failure is done by a vision system. Three different types of recovery are defined within this paper: *no recovery*, *local recovery* and *global recovery*. They describe *no recovery* behavior as follows: performing all actions, evaluate if it succeeded and if not perform all actions again until it succeeds. *Local recovery* is defined as checking after each action if the expected outcome occurs and if not, perform the action again until it succeeds. *Global recovery* means performing another action as recovery behavior and call he first action again. So within this work the authors had the opportunity to check if a single skill succeeds through a vision system. We followed a similar approach with a local recovery behaviour but at a lower level. We performed the local recovery at the skill primitive layer. We believe that it is easier to recognize a failure earlier and can perform reactive behaviors more easily. Further in [7] they proposed infinite retries for local recoveries. For our system this does not make sense because at most failures the environment changed so much that skill primitives would not be able to be performed any more (e.g. loses box during manipulation). They proposed a really good abstract way of failures of actions and how they convert the output of the planner into a state machine. We use a different state machine pattern from [8], but we could benefit a lot from this idea.
The following papers deal with AI planning. In [9] the author first discusses the GRAPHPLAN algorithm (fur further reading please see [10]) and its optimizations. The author sees the GRAPHPLAN algorithm as a basis for encoding planning problems into propositional SAT. He explains how the graph expansion needs to be adopted and how the solution is then extracted. The first part focuses on the STRIPS[1] representation where actions are limited to conjunctive and quantifier free preconditions and effects. Following he describes expansions to GRAPHPLAN, so that disjunctive precondition, conditionals effects and universal quantifiers can be used. Further the author gives an overview of the compiling of planning problems into propositional

[1]STRIPS planning is refered in literature as classical planning for restricted state-transition-systems, see page 17 in [11]

formulas, so that propositional satisfiability problem (SAT) algorithms can be used for solving them. Further he shortly summarizes SAT planners. The author shows also a nice example of AI planning using STRIPS like formalism in a NASA space shuttle.
The fast-forward planning system was the best performance planner in th artificial intelligence planning systems (AIPS) competition in 2000 and is presented in [12]. It performs a search in state-space using a heuristic which guides the search. It tries to escape local minima and plateaus through a systematic search. The planner is tested on different planning benchmarks, where it clearly outperforms other planners. However, the author describes these benchmarks as 'simple'. So he tests the planner on complexer problems too and compares it to other planners. In these scenarios the other planner perform much better. The presented one gets stuck in large local minima where it cannot escape with systematic search.
In [13] also a planner is described, which participated at the AIPS competition. This planner supports features like soft constraints and preferences, as well as fluents. The fluents feature allows to assign numbers to variables, instead of only true or false (like it is the case for predicates). Usually this feature is used to assign costs to different action. However, because of the fluent feature the planner is very interesting for our work. It provides the opportunity for counting.

3.2 Grasping

Grasping is one essential part in object manipulation. In literature different grasping strategies, grasping poses and evaluations are discussed.
In [14] three special poses during a grasping action are defined. The *pregrasp pose* is defined as final pose of the endeffector for grasping the object but being an offset away from the object. The *grasp pose* is reached, when the endeffector needs to close the gripper and reliably grasps the the object. After grasping the object and moving the endeffector to a specific pose. This pose is called *postgrasp pose*. Within this paper the grasps and pregrasps are recorded manually and saved in a database. Therefore the compliant robotic arm is moved in the *pregrasp pose* and the *grasp pose*. Only the selection of grasps is performed online. Chosen grasps and their pregrasps are then

sent to the inverse kinematic (IK) solver to move the arm first towards the *pregrasp pose* and then to the *grasp pose*. Important to mention is that more than a single grasp is sent to the IK solver. We pick up this idea because many grasps have similar quality measures, but one grasp can be easier reached than another. If not too many grasps are sent to the IK solver, they can be tested if a solution exists very fast.
Whereas in [14] human taught grasps are saved, the next work [15] also uses predefined grasps but not human taught ones. Their online selection focuses also on feasibility of a grasp but also on the stability. In order to calculate a grasp offline the geometric model of the gripper and the target object are needed. Then the object model's surface and valid endeffector parameters are sampled. From these samples a valid manipulator pose is calculated and a force closure score is computed for each grasp. A grasp scoring function is defined to rank the grasps online. This score takes into account the former computed force closure score, the local environment of the object (e.g. how far are other objects away from endeffector when performing the selected grasp) and the robot's relative pose (prefers grasps where palm faces away from robot).
Within the work of [14] and [15] a model exists of the known object which should be grasped. In [16] they describe an approach to calculate grasps for unknown objects. This planner uses the clustered point cloud of an object and computes grasps using a heuristic based on shape and local features. One important observation within [16] is, that good grasps of human designed objects are along the object's principle axis. Those grasps are ranked online. The second grasping planner which is presented in [16] is a recognition based database planner. Grasps are planned on mesh models of known objects in advance and then chosen if object is recognized. For this approach GraspIt! [17] is used, which uses the Eigengrasps from [18]. Here the basic idea is taken from human grasps. It is shown that after recording human performed grasps and running a principle component analysis (PCA) on it, that circa 85% of all grasps are a linear combination of the first and the second principle component. These components are referred as Eigengrasps within [18]. Each endeffector has predefined contact points, where the grasped object gets in contact with the endeffector. Ciocarle et al. defined an energy function which brings these contact points in contact with the object. This energy function is minimized using simulated annealing. Due to the ability of the energy function to deal with obstacles this planner

can be used to generate valid grasps online too. We appreciate the idea of online planning. All obstacles and the graspable object can be considered. The grasps can be calculated online without defining a complex ranking function. The obvious drawback in online grasp computation is the time consumption.
In [14] - [18] errors in sensed poses of objects and obstacles or inaccurate execution of motion are not taken into account. In order to overcome these errors some work suggests reactive grasping. In [19] an abstract framework for grasping is presented. This framework handles with manipulation and grasping tasks including how correction of these tasks can be achieved using finite state machines (FSM). In [16] grasping results are improved using reactive approaches. Within this work the information sensed from tactile sensors is used to perform reactive behaviors to overcome small errors in perception. But many platforms have no tactile sensors and so in [20] a vision control loop is used. Here a single camera observes both endeffector and object which should be grasped. The authors distinguish between a so called reaching phase which contains the movement of the arm towards the object and the grasping itself. The interaction phase is about the manipulation of the object. A position based visual servoing closed loop approach is used to successfully perform the reaching phase. This closed loop approach performs also very good if the robot's movements are inaccurate. We pick up the vision control loop idea from [20] to grasp the handle of the transport box. The camera which senses the environment is mounted on the endeffector. So the gripper and the object is observed simultaneously. In our case where both pose estimation of the handle and movements seem to be inaccurate, this approach works out fine. Although within this setup the relative error can only be calculated.
In [21] an approach is presented where grasp and motion planning are combined. In particular a probabilistic planning approach is presented which is a variation of a rapidly exploring random tree (RRT). A tree is built up which consists of collission free and reachable configurations for the arm. Some nodes are tested if the center point of the endeffector can be moved towards the object and a stable grasp can be performed. The advantage of this method is the combination of motion planning, IK and grasping point calculation. It was pointed out that finding grasps online has the benefit that it is not limited to the finite set of predefined grasps. Although this is a really different approach, they show that online grasp computation has

great advantages. So we decide to compute grasps online, but separate it from IK and motion planning.

3.3 Perception

Due to the introduction of cheap 3D cameras, the environment is sensed as point cloud. Point clouds of known objects can be compared with the sensed point cloud of the environment much more easily.
For object recognition and object's pose determination point clouds of existing models can be matched with the sensed environmental point cloud. One idea how they can be matched is the iterative closest point (ICP) algorithm proposed in [22]. This algorithm tries to minimize the distance between points. Therefore a distance and minimization function need to be defined. The minimization function calculates the rotation and translation of the point cloud for the next iteration. The main disadvantage of this algorithm is the convergence against local minima. One strategy to overcome local minima is proposed in the same work [22]. Creating different initial states and test all the different states for performing in the best way. The advantage of the ICP is that neither features have to be calculated, nor any surface normals or curves and their derivates. Nowadays many different variations and implementations of the ICP exist (see [23] for a comparison of different implementations). Because this algorithm gets stuck in local minima easily, it is not appropriate for our purpose in that way.
The next paper [24] follows a different approach. It extracts features and tries to fit these features and not the whole point cloud. In [24] the point feature histogram (PFH) is introduced. The local 3D features within the k-nearest neighbor (KNN) of a point are extracted. Within the PFH all neighbors are fully connected and the computational complexity of n points in a point cloud is O($n^2 \cdot k$) for k neighbors. Rusu et al. therefore presented a simplified PFH version called fast point feature histogram (FPFH) with a computational complexity of O($n \cdot k$). There the neighbors are not fully connected and some neighboring pair values can be reused. The authors also present an alignment method, namely the sample consensus initial alignment (SAC-IA). They sample an amount of correspondences and rank them. They do not try out all combinations and use a non-linear optimization

technique. Although this algorithm works quite good for objects containing many features, we cannot use it for objects with rare features. The algorithm aligns the objects not accurate enough.
In [25] a different descriptor is presented which is an extension of the FPFH. This descriptor includes also information about the view point. For this descriptor a preprocessing step is very important, so that objects are already segmented before applying the descriptor. Within [25] they cut out all planes and performed Euclidean clustering. But there need to remain enough points within the cloud in order to compute valid descriptors. The idea of pre-segmenting the objects is really good because this can safe a lot of computation time. But this is not always possible in clustered scenes or objects with flat surfaces.
Within [26] a coarse to fine approach is presented where the FPFH is used for coarse registration and the ICP for refining this registration. The authors align point clouds of human faces within in a digital face inspection framework. The intention is that dentists can monitor faces during orthodontic treatments. In the first step a preprocessing is performed. This includes removing outliers and reducing the resolution of the point cloud through down sampling. The next step is the coarse registration using FPFH features for aligning point clouds as initial alignment. The last step is fine registration which uses the alignment of the coarse registration and then uses the ICP for refining this registration. The coarse to fine approach is a good idea. We follow a similar approach to register point clouds of known objects. Using only the coarse alignment delivers too inaccurate poses. But using the coarse alignment before the ICP prevents the ICP to get stuck in a local minima. The combination of both algorithms works out fine for our object recognition.

3.4 Integrated Systems

In [27] a mobile bin picking application is presented. The task of the robot is navigating towards a transport box which includes many pipe sections, grasping one single item and returning it to a process station. The high level is implemented as FSM which contains following states: navigate towards transport box, cognition phase (transport box and items are recognized),

a pick up phase, navigate to process station and placing object. For object recognition a RGB-D camera is used and three dimensional (3D) data is reconstructed from different views on the scene. The transport box is recognized using an iterative closest point algorithm (ICP) and the items are recognized using a coarse-to-fine registration. The grasps are calculated offline through sampling shape primitives. The online selection of feasible grasps is based on a scoring function which includes information about a collision free approach and about the IK. The linear bidirectional kinematic version of motion planning by interior–exterior cell exploration (LBKPIECE) from [28] is used. The specialty about motion planning while picking items is that small collisions with other items are allowed. Otherwise no successful motion plan would have been found.
In [29] a software architecture and their implementation for grasping objects is presented. Some of these concepts are used in our work too. The collision environment is a 3D occupancy grid which is produced from a point cloud, but the robot parts are filtered out. Known and recognized objects are represented as geometric primitives or as mesh models of the objects. Here the collision scene is recorded when the arms are moved out of the work space of the robot. So the whole environment is seen and nothing can be hide behind the arms. For grasp planning two modules are used. One for known objects, which uses precalculated grasps. The second one is for unknown objects, which uses the shape and local features of the objects. A sample-based motion planner from the Open Motion Planning Library (OMPL) [30] and a collision-aware IK is used. The planned trajectory is executed and monitored by an extra controller. Due to noisy sensors and inaccurate object pose detections a reactive grasping approach is introduced. This relies mainly on information gained from tactile sensors.
An extension of this work is published in [16] and some of these concepts are extended. Here Chitta et al. present a pick and place approach where they have to deal with known and unknown objects, cluttered workspace and noisy sensor data. The environment is represented as OctoMap [31] using the data from a 3D sensor. Objects are tracked and recognized using 2D sensors. Those objects are then represented as meshes. Known objects have a semantic representation too, which is important for making high level decisions. For global planning a correct and persistent model of the environment is needed. In order to ensure a collision free motion, the moving arm is actively monitored by the 3D sensor. For local planning a

more reactive model is needed and so the information from tactile sensors is used. Again motion planners from the OMPL are used. Some of the presented concepts are implemented in open source projects called MoveIt! and GraspIt!, which we discuss in section 4.2 and section 4.3. We benefit a lot from this work and can also reuse some parts in order to not reinvent the wheel.

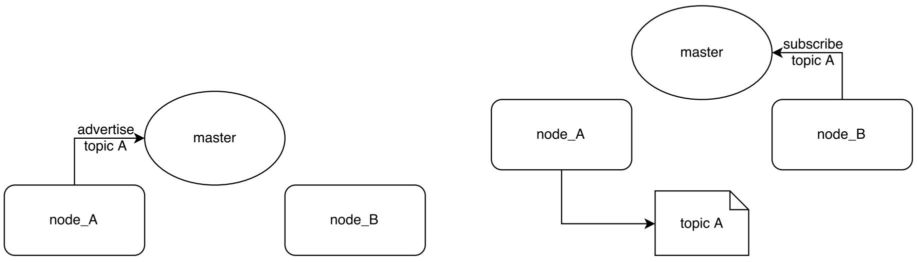
4 Prerequesites

In this chapter we describe concepts and software modules which we reuse.

4.1 ROS

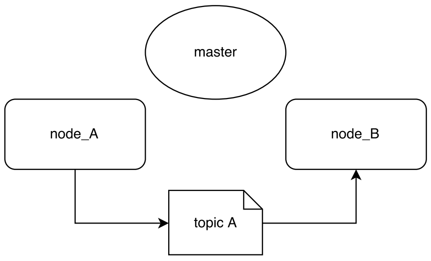
In [32] the Robot Operating System (ROS) is introduced. This is an open source robotic framework which is a middle-ware and runs on linux-based platforms. It provides modules for hardware abstraction, driver for hardware components, communication between processes and a management for different packages. Most concepts are shown in [32]. The latest documentation can be found at `http://wiki.ros.org` . In the following paragraph the basic concept of ROS are introduced.
Nodes are processes. The communication between single nodes is via messages. A **message** is a clear defined data structure. This communication between processes is based on the publisher/subscriber pattern. Nodes sends messages via topics. A **topic** can be understood as named communication channel. Nodes publish messages on topics and any other node could receive those messages when subscribing to the topic. This communication is a way of broadcasting messages within the whole system. There can be multiple nodes publishing on the same topic. Sometimes the processing flow of two nodes need to be synchronized via a request of one and a reply the other node. A **service** provides this core functionality. A node can provide a service under a unique name. Another node sends the request and waits blocking until the service replies. The **ROS master** is the core of the running system. It needs to be started before any other node. It provides naming and registration services, keeps track of publishing and subscribing messages and surveys service requests and replies. In Figure 4.1 the concept

of publishing and subscribing to a specific topic is shown. In Figure 4.2 the service request and service reply scheme is shown.



(a) A node is the publisher if it advertises a topic. At the beginning it registers the topic at the master.

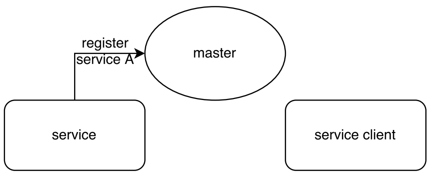
(b) A node which wants to subscribe to a specific topic is called subscriber. First it looks up the topic at the master.



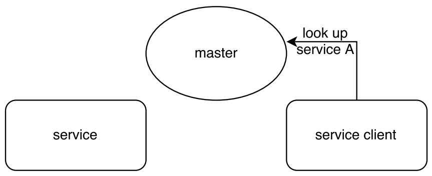
(c) The master takes care of directing the advertised topic to the subscriber

Figure 4.1: This figure describes the publishing and subscribing to a topic. Concept taken from `http://wiki.ros.org`

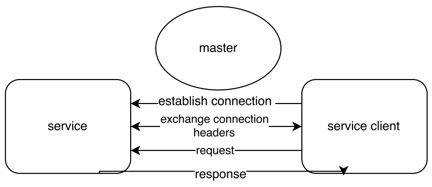
The **parameter server** is used to store static values like configuration parameters. This server can be accessed during runtime by all nodes. All **bold features** are the core functionalities of ROS. A preemptable task is still missing. This is implemented in another library called actionlib. One node implements the server application and any other node can implement the client application, which calls the server and starts the execution of the task. In Figure 4.3 the concept of actions is visualized.



(a)At the beginning the node registers the service at the master.



(b)A node which wants to use this service is called service client. First it looks up the service at the master.



(c)Then client establishes a connection with the service. After a handshake the client sends the request and waits for the response of the service

Figure 4.2: This figure describes how a service client establishes connection to a service. Concept taken from `http://wiki.ros.org`

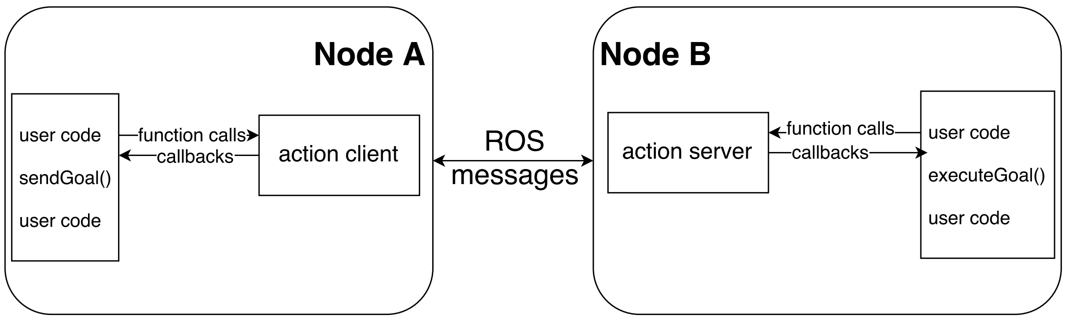


Figure 4.3: The concept of actions in ROS. Within Node B the action server callbacks (cb) are implemented within the user code. The callbacks are called by the action server when the action client communicates via the ROS communication channels. The user code interacts with the server directly using function calls. In Node A the user can start an action using function calls from the action client. Information about the state of the action can be accessed using function calls or implemented as callbacks. Concept taken from `http://wiki.ros.org`

4.2 MoveIt!

In [33] and on `http://moveit.ros.org/` MoveIt! is presented. It is a framework for manipulating robotic arms. It also wraps some external libraries and ROS packages, so that users can easily access the whole manipulation pipeline through defined interfaces. It provides an environmental representation, includes libraries for motion planning and monitors motion execution. The **Planning Scene Monitor** monitors both: the robot and the environment. The later builds upon the octomap package [31]. It contains a voxel grid representation for unknown obstacles and a representation for recognized objects based on geometric primitives. In figure 4.4 the concept of the Planning Scene Monitor is shown. The main input of the motion planning

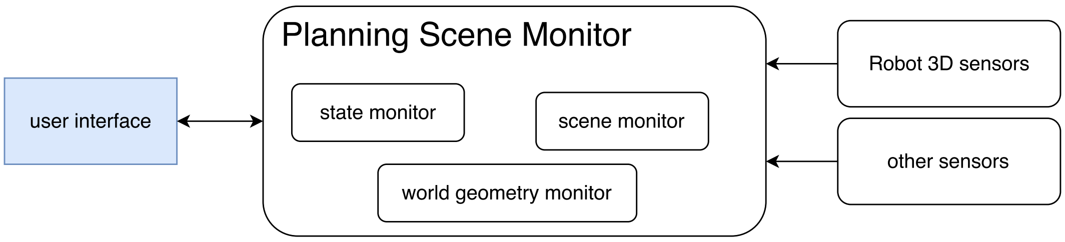


Figure 4.4: The planning scene monitor consists of the state monitor, the scene monitor and the world geometry monitor. The state monitor keeps track of the current pose of robot and the poses of the arm links. It uses the information which are provided by the robot sensors. The scene monitor gains information from the robot's 3D sensors. It builds up the voxel grid and refreshes it continuously. The world geometry monitor maintains the information from all geometric primitives which were registered at the planning scene. The user can access the planning scene through defined interfaces (e.g. introducing new geometric primitives or attaching it to a robot link). The concept is taken from `http://moveit.ros.org/`

module are the planning scene, a kinematic and semantic description of the robot. The kinematic information is stored as Unified Robot Description Format (URDF) and the semantic information as Semantic Robot Description Format (SRDF) on the ROS parameter server. With this information the motion planning module computes collision free motion plans. On default OMPL is loaded via a plugin system, but other libraries can be used as well. After generating a motion plan, the plan gets smoothed and a trajectory is generated, which is then forwarded to a controller. This controller executes and monitors the execution of the trajectory. If an error occurs the controller aborts the execution. In figure 4.5 the concept of MoveIt! is visualized. The move group node is the core of the MoveIt! interface. It provides the user interface, fuses the information from sensors in modules like the planning scene monitor and interacts with the trajectory execution module. For example the move group node provides interfaces to define the pose goal for a specific link. Then it handles the trajectory planning as well as the trajectory execution. So if the environment would change during the execution in the area of the arm, it aborts the movements. MoveIt! provides also the

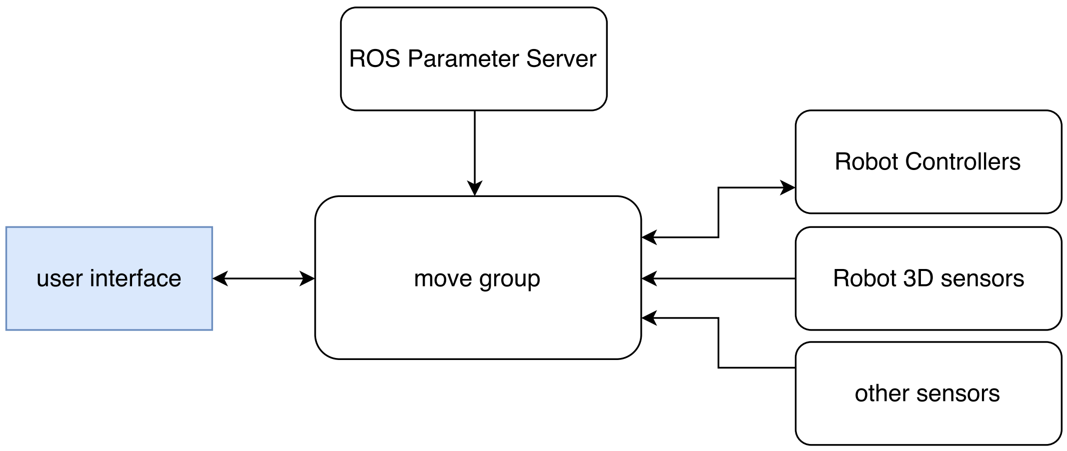


Figure 4.5: The move group node loads all necessary configuration parameters from the ROS parameter server. It holds also the planning scene monitor and the motion planning module. Therefore it gains information from the robotic sensors and the current robotic state. It provides planned trajectories to the robot controller. Through the user interface it informs the user about a successful execution of the trajectory. If an error occurs it reports the error from the controller. Concept taken from `http://moveit.ros.org/`

opportunity to integrate other robotic frameworks. If the robot's description is in an URDF odr Collada file format, it can be loaded by a MoveIt! setup assistant, which the sets up the MoveIt! package for the specific robot. A screenshot can be seen in Figure 4.6

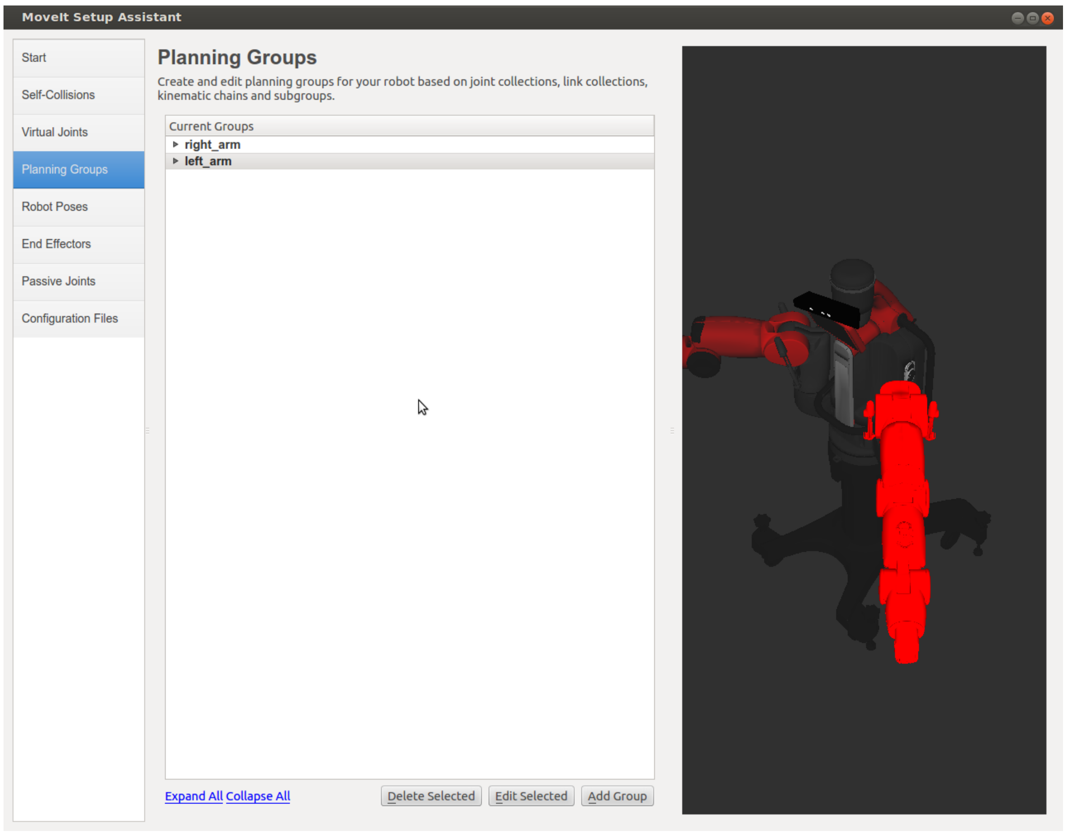


Figure 4.6: This figure shows the setup assistant for the Baxter robot. This assistant loads the robot's description and generates a MoveIt! package.

4.3 GraspIt!

GraspIt! [17] is a simulation tool for grasp analysis and grasp planning. Within GraspIt! it is possible to load obstacles, graspable objects and different kind of endeffectors. Because all bodies are defined within files, it is quite easy to introduce new objects and endeffectors.
If a grasp is simulated, GraspIt! provides real time collision and contact information. The unit grasp wrench space (GWS) is constructed for determining the quality of the grasp (grasp analysis). Two quality measures are introduced: the ϵ and v measurement. The ϵ measurement corresponds to

the minimal ball which fits into the convex hull of the GWS space. The radius of this ball is the ϵ measurement. The v corresponds to the volume of the convex hull. GraspIt! also provides a visualization of the GWS. Due to the fact that the GWS is a six dimensional space, the user has to choose three fix axis so that the space can be visualized.
GraspIt! has not only the ability to evaluate grasps but also to plan new grasps. Within GraspIt! the Eigengrasp [18] approach is implemented. So new grasps can be planned and immediately simulated as well as evaluated.

4.4 PCL

The point cloud library (PCL) is presented in [34]. A very good documentation and tutorial can be found on `http://pointclouds.org/documentation/`. This library collects algorithms and functionalities for processing point clouds fast and efficiently. Further it is fully integrated in ROS. PCL provides the the possibility for recording and loading point clouds but also to process them in real time. Visualization tools from ROS can be used to show point clouds online. Nevertheless PCL provides some tools for visualizing point clouds too.
The following overview is taken from `http://pointclouds.org/documentation/`. PCL provides a viewer named *PCD viewer*. It displays point clouds as can be seen in Figure 4.7. PCL provides different kind of algorithms and efficient data structures. For example the octree and kd-tree data structures is implemented.
PCL provides filtering algorithms for noisy data and offers interfaces to load point clouds from hardware devices like laser scanner or 3D cameras. When point clouds are not generated artificially, but recorded by hardware devices they are often very noise. PCL provides different algorithms and strategies to remove noise or outliers. Points are called outliers if they do not fit in the statistical characteristics of their neighbors.
In order to compare point clouds, PCL provides different feature extractors for point clouds. Due to the nature of a point cloud, these feature are 3-dimensional and computed using information of the k next neighbors (also referred as k-neighborhood). Beside the feature extractors they also provide

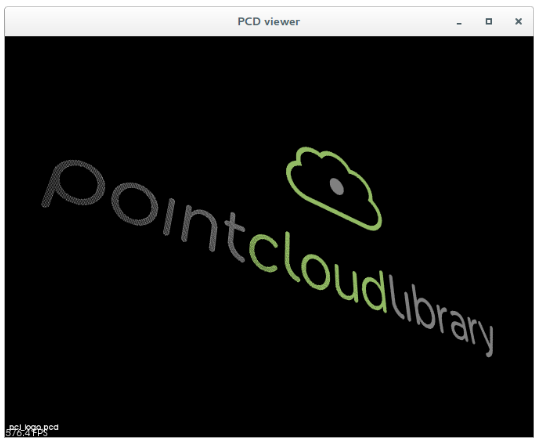


Figure 4.7: This figure shows the visualization tool of PCL. It shows the point cloud of the PCL logo.

so called keypoint calculation. Keypoints are special points in point clouds which have different features than other points. Such interesting points can be used as sparse features to represent the point cloud in a compact form. There are also algorithms for point cloud alignment implemented (registration), as well as sample consensus algorithms for fitting different models into point clouds. Such models can be loaded, but there are already computed ones like lines, planes, cylinders and spheres.
In the PCL different kinds of segmentation algorithms are implemented. So point clouds can be split into sub-point clouds and only sub clouds are further processed to save computation time. One well known algorithm is the Euclidean clustering algorithm. In Figure 4.8 a sample point cloud from the PCL is shown and in Figure 4.9 one of the sub-point clouds produced by the clustering algorithm.

4.5 Baxter

Baxter [35] is a two arm, collaborative robot built by Rethink Robotics. Each arm has seven degrees of freedom (DOF). In addition each arm has one RGB camera and one infra red (IR) sensors to supervise performed grasps.

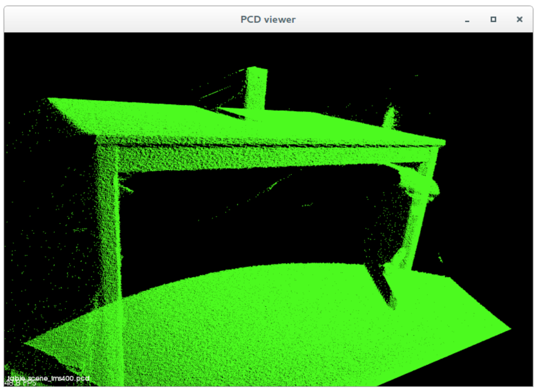


Figure 4.8: This figure shows the visualization of the sample point cloud of PCL

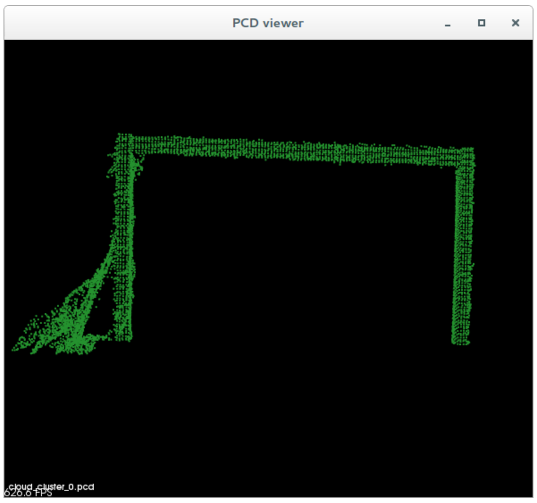


Figure 4.9: This figure shows one cluster which was generated by the Euclidean clustering algorithm of the sample point cloud shown in Figure 4.8

Different types of gripper can be mounted, like an electrical parallel gripper, vacuum cup gripper or custom made grippers.

4.5.1 ROS support

Baxter is fully ROS supported. Figure 4.10 shows an overview of its software development kit (SDK). Baxter is delivered with an in-built-computer. On this computer embedded controller software is running, as well as the ROS master, motor controller nodes, ROS diagnostics, the transform library (tf) node and the robot state publisher. The transform library node keeps track of all frames in the system over time. It can calculate relative transformations between the different frames [36]. The robot state publisher subscribes to topics which are related to the robot state (e.g. joint angles) and publishes this information as tf. Another computer can be used as workstation if ROS is installed. It communicates with Baxter over ROS networking connection using defined application programming interfaces (API). So own nodes can be run on the workstation, or other ROS tools can be started.

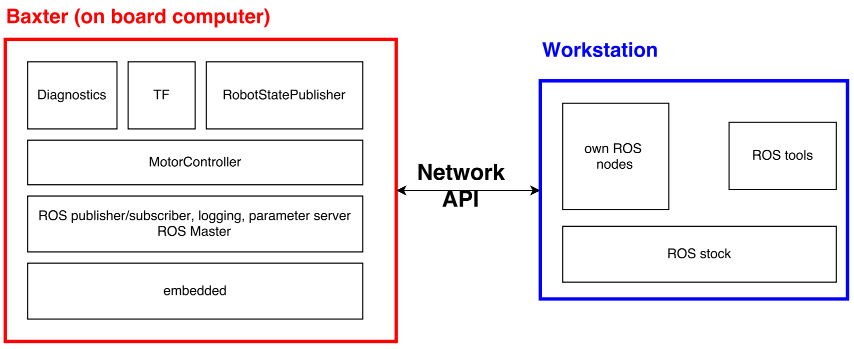


Figure 4.10: Overview of Baxter's software development kit (SDK) and its ROS nodes. Concept taken from `http://sdk.rethinkrobotics.com/wiki/Baxter_Research_Software_Developers_Kit_(SDK)`

4.5.2 Sensors, Inputs and Outputs

Baxter provides a number of different sensors. On its head a ring of sonar sensors is mounted, it has two RGB cameras on its end effectors and one RGB camera in its head mounted display. We installed an additional RGB-D camera on top of its head display to gain 3D information about the environment. In Figure 4.11 the mounted Asus can be seen from two perspectives. Buttons and scrolls on Baxter's arms and buttons on Baxter's back provide

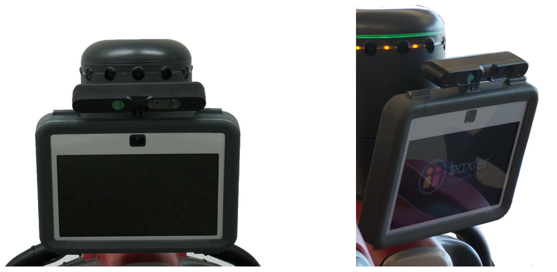

(a) Asus camera, frontal perspective. (b) Asus camera, side perspective.

Figure 4.11: This figure shows the Asus on Baxter's head from front and side position

the opportunity to use them as user input for applications. Information or user interfaces (UI) can be presented on Baxter's head display. An additional output is the LED ring on Baxter's head. It can be colored green, red or anything in between. In figure 4.12 Baxter is shown. For this task an electrical parallel jaw gripper is mounted on Baxter's end effector. A detailed view is shown in figure 4.13.

4.5.3 Arm controlling via ROS interfaces

Baxter's SDK provides a joint trajectory action server (JTAS) node which is started on the workstation. This action takes a joint trajectory, which can be send via a ROS action call from another node (e.g which is using the move group interface). This joint trajectory provides information of each joint's position with a time stamp. Optionally it provides information about velocity or acceleration. The JTAS interpolates these points and sends a command to a controller which is running on Baxter. In turn the controller provides information about Baxter's state. The JTAS monitors this state and if any value runs out of its constraints it aborts the trajectory execution. The

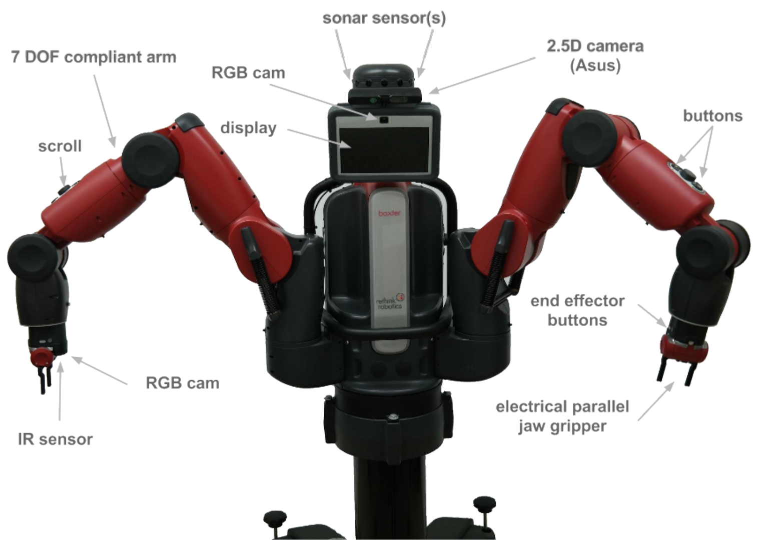


Figure 4.12: Within this figure Baxter is shown. It can be seen that it has two 7DOF arms with the scrolls and buttons on the upper arms. On Baxter's head the sonar sensor ring, the display with the RGB camera and the 2.5D camera is marked. The endeffectors are equipped with electrical parallel jaw grippers, a RGB camera and a IR sensor.

controller on Baxter forwards the commands to a real time motor controller to guarantee precise execution. This scenario is shown in figure 4.14.

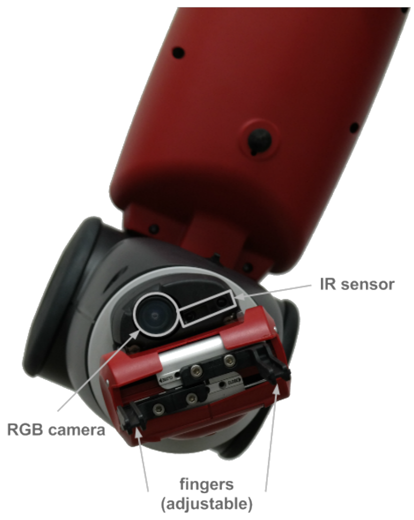


Figure 4.13: The RGB camera as well as the IR sensor are integrated in the endeffector. The electrical parallel jaw gripper could be exchanged by another gripper. On the gripper two fingers are mounted. The position of the fingers is adjustable for each finger. The fingers can be exchanged by wider or longer fingers too. The additional fingers are delivered with the electrical parallel jaw gripper.

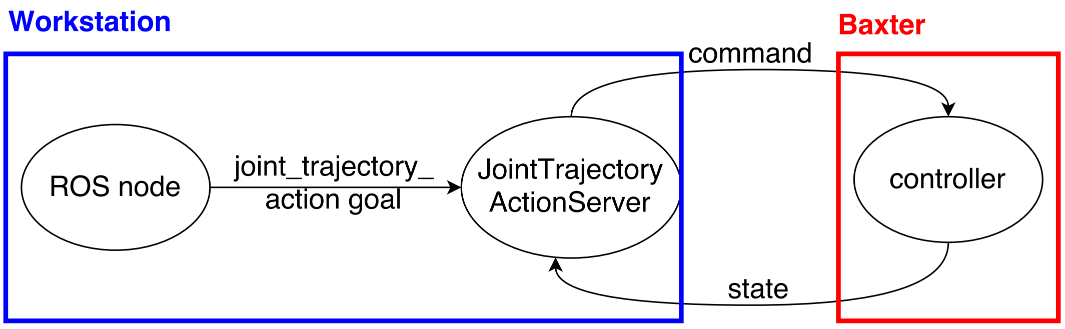


Figure 4.14: Overview of joint trajectory action server from Baxter's SDK and how provided joint trajectories are executed. Concept taken from `http://wiki.ros.org/joint_trajectory_action`

5 System Overview

Within this chapter an overview of the Industrial Grasping system is given. We start with describing breifly a general 3-TIER architecture. Then we discuss the planning layer. This layer receives information about the skills, as well as information about the start and the goal state of the environment. It outputs a list of skills, which leads to successful execution of the given task, if the skills are executed successfully. Next the execution layer is discussed, which handles the execution of the list of skills. It relies on the decomposition of skill into skill primitives. Finally we talk about the skill primitives in detail.

5.1 Overview of the architecture for the robotic order picking

Within this secttion a general 3-TIER architecture is described. Moreover the specialization of this architecture to solve the problem described in chapter 2 is presented. In Figure 5.1 the conceptional overview of a general 3-TIER architecture is shown. The 3 layers for planning, executive and behavioral control are clearly separated. The communication is clearly defined. The top layer represents the planning layer. The planner uses the information of the domain and the problem to generate a plan. The plan is a sequence of skills that have to be executed to reach a given goal. The plan is forwarded to the next layer. The executive layer takes care of the execution of each skill. It knows about the composition of the skill primitives. The primitives are located in the behavioral layer. The advantage of this model is its clear structure and its modularity. For for further reading about the 3-TIER architecture please see [37, p. 244–277].

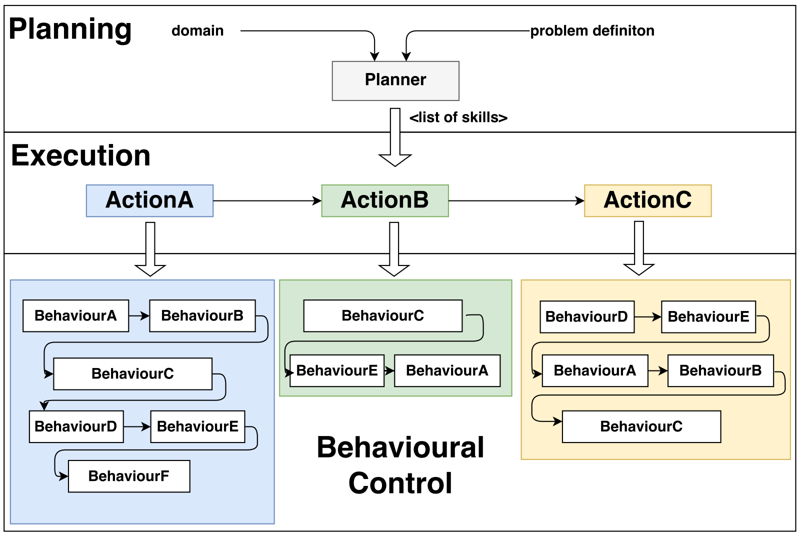


Figure 5.1: This figure illustrates the concept of a 3-TIER architecture. It consists of a planning layer, executive laer and behavioural control layer.

In the following sections each layer is explained in detail and how this concept can be implemented.

5.2 Planning Layer

The top layer of a 3-TIER architecture is the planning layer. The planning layer contains a domain and problem description of the given environment and task. The planner itself is also in this layer and takes the domain and problem as input.
We use the classical representation for planning problems which is based on first order logic. It is described in [11, Chapter 2.3].

5.2.1 Domain

The domain contains information about all the objects and classes which can appear in the environment. Further it holds information about all the skills (= actions) the robot is able to perform. A skill is defined through its name and the it's parameters. A skill has a conjunctive precondition and a conjunctive effect. The precondition is the state of the environment which needs to be fulfilled before the action can be performed. The effect of a skill is the description of the environment after the skill is performed. In table 5.1 a general skill is described. Further the domain contains predicates and functions which describe the environment.
Types offer the opportunity to group different classes. Classes are any

Table 5.1: General Skill

name	**precondition**	**effect**
name of skill and ⟨Params⟩ in use	any condition that need to be fulfilled to perform skill	how the skill changes the environment

Within this table a general skill is shown. The parameters of the skill are marked with pointed brackets (⟨, ⟩)

objects that appear in the environment. Within the industrial grasping domain following types exists: boxes, locations and places. A class can belong to more than a single type. The classes *DeliveryBox*, *Box*, *Tray* and *Level* are needed. Predicates and functions can belong to types, or to single classes. The predicate *free* is defined for the places type, to model if a place is free or occupied. The functions *currentLoad* and *hasType* are defined for boxes. The *currentLoad* function returns the amount of stored items within a box. The function *hasType* returns the type of item which are stored in the box. The function *location* is defined for the locations type. It returns the current location of the class. The function *hasLevel* is only defined for the class *Box*. This function returns the *Level* where the *Box* should be stored. These relations are visualized in figure 5.2.

In order to solve the problem, the robot must have the following skills:

- move⟨ Box ⟩From⟨ Level ⟩To⟨ Tray ⟩

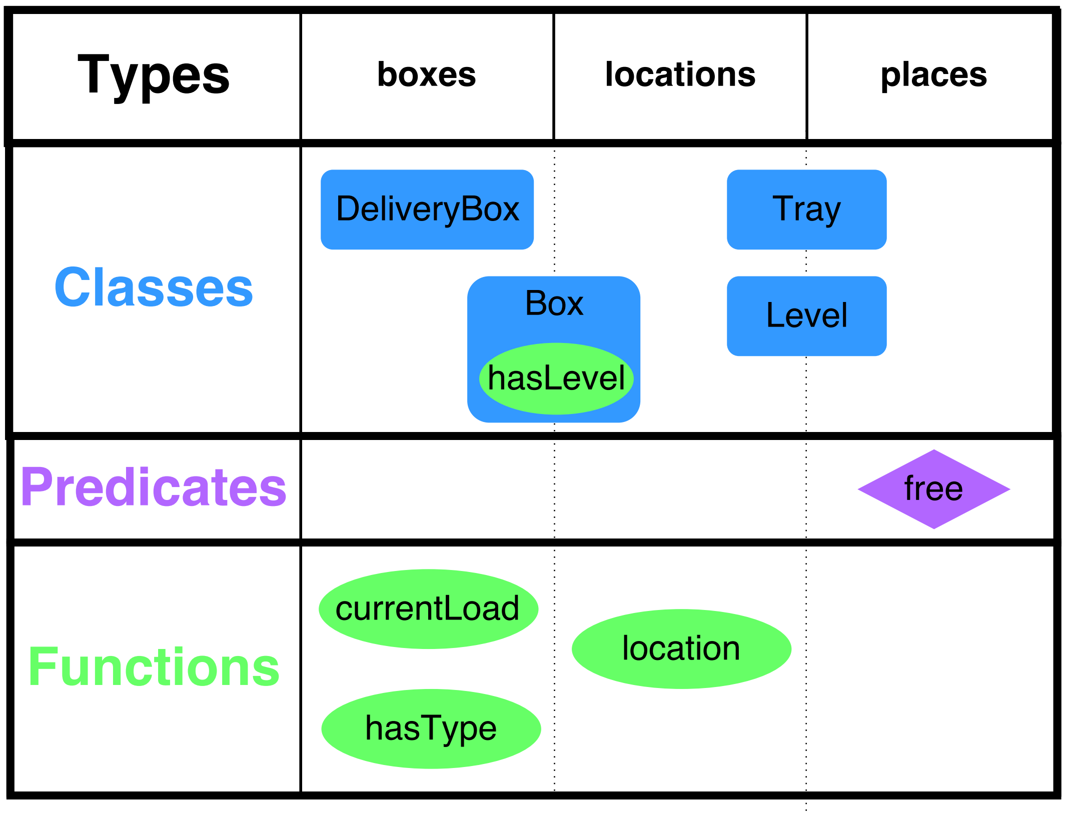


Figure 5.2: In this figure the different types, classes, predicates and functions are listed and their relations are visualized. Classes can belong to more than a single type. Predicates and function can either belong to types or single classes.

- move⟨ Box ⟩ From ⟨ Tray ⟩ To⟨ Level ⟩
- graspItemFrom⟨ Box ⟩PlaceTo⟨ DeliveryBox ⟩On⟨ Tray ⟩

They are explicitly listed in Table 5.2.

The preconditions and effects for each skill are intuitive. For the skill *move⟨ Box ⟩ From ⟨ Level ⟩To⟨ Tray ⟩* the box needs to be on the level and tray must be free. The effect is that the box is on the tray, so the tray is obviously not free any more, but the level is. More or less the same holds the other way round. To perform *move⟨ Box ⟩ From ⟨ Tray ⟩To⟨ Level ⟩*, the box needs to be on the tray, the level needs to be free and it must be allowed to store

Table 5.2: Skills for Order Picking Problem

name	**precondition**	**effect**
move⟨ Box ⟩ From ⟨ Level ⟩To⟨ Tray ⟩	• ⟨ Box ⟩ On⟨ Level ⟩ • ⟨ Tray ⟩ Free	• ⟨ Box ⟩ On ⟨ Tray ⟩ • ⟨ Tray ⟩ NotFree • ⟨ Level ⟩ Free
move⟨ Box ⟩ From ⟨ Tray ⟩ To⟨ Level ⟩	• ⟨ Box ⟩ On⟨ Tray ⟩ • ⟨ Level ⟩ Free • ⟨ Box ⟩ BelongsTo⟨ Level ⟩	• ⟨ Box ⟩ On ⟨ Level ⟩ • ⟨ Tray ⟩ Free • ⟨ Level ⟩ NotFree
graspItemFrom⟨ Box ⟩ PlaceTo ⟨ DeliveryBox ⟩On⟨ Tray ⟩	• ⟨s Box ⟩On⟨ Tray ⟩ • #Items(⟨ Box ⟩) > 0 • ItemType(⟨ Box ⟩) = ItemType(⟨ DeliveryBox ⟩)	• Decrease #Items(⟨ Box ⟩) • Increase #Items(⟨ DeliveryBox ⟩)

Within this table the skills are described which are needed to solve the industrial grasping problem.

this specific box in the level. The last constraint is not intuitive on the first glance, but due to a order management system specific boxes are stored at specific levels so that they can be found easily again (by a robot or human). This action's effect is that the box is on the level, the level is not free any more but the tray. The *graspItemFrom⟨ Box ⟩ PlaceTo ⟨ DeliveryBox ⟩On⟨ Tray ⟩* requires that the delivery box holds the same items as the box. Further the box needs to be on the tray and it must not be empty. The effect is that inside the box the amount of items is decreased by one and the amount of picked items in the delivery box is increased by one.

5.2.2 Problem

The problem describes the initial state s_0 and the goal g, which is a set of propositions [11]. The goal state s_g is a state, that satisfies g. Further the problem holds information about additional constraints, which are taken into account by the planner (e.g. costs which are minimized). First object instances are defined, which occur in the environment and their initial properties are stored. In Figure 5.3 a possible initial state is shown and in Figure 5.4 a reachable goal state is shown. For this example we define following constants:

- box_A
- box_B
- box_C
- $dbox\_A$
- $dbox\_B$
- $dbox\_C$
- $type\_A$
- $type\_B$
- $type\_C$
- $level\_0$
- $level\_1$
- $level\_2$
- $level\_3$
- $tray$

In the initial state s_0 three boxes are stored respectively on the levels $level\_1$, $level\_2$, $level\_3$. box_A holds five items of $type\_A$, box_B holds one item of $type\_B$ and box_C holds three items of $type\_C$. The $tray$ is free and the delivery boxes ($dbox\_A$, $dbox\_B$, $dbox\_C$) are empty . Each delivery box can load only a specific item type. The different delivery boxes can be understood as part of one single large box. Of course this can be a single box too. For the initial state s_0 following holds:

- $location(box\_A) = level\_1$
- $location(box\_B) = level\_2$
- $location(box\_C) = level\_3$
- $currentLoad(box\_A) = 5$
- $hasType(box\_A) = type\_A$
- $currentLoad(box\_B) = 1$
- $hasType(box\_B) = type\_B$
- $currentLoad(box\_C) = 3$
- $hasType(box\_C) = type\_C$
- free($tray$)

- $currentLoad(dbox\_A) = 0$,
- $currentLoad(dbox\_B) = 0$
- $currentLoad(dbox\_C) = 0$
- $hasType(dbox\_A) = type\_A$
- $hasType(dbox\_B) = type\_B$
- $hasType(dbox\_C) = type\_C$

When the goal state s_g is reached, in the *dbox_C* should be one item. In *delivery_box_A* should be two items. The *tray* is free again and so all boxes must be returned to their level. Following holds for the goal state s_g:

- $currentLoad(dbox\_C) = 1$
- $currentLoad(dbox\_A) = 2$
- free($tray$)

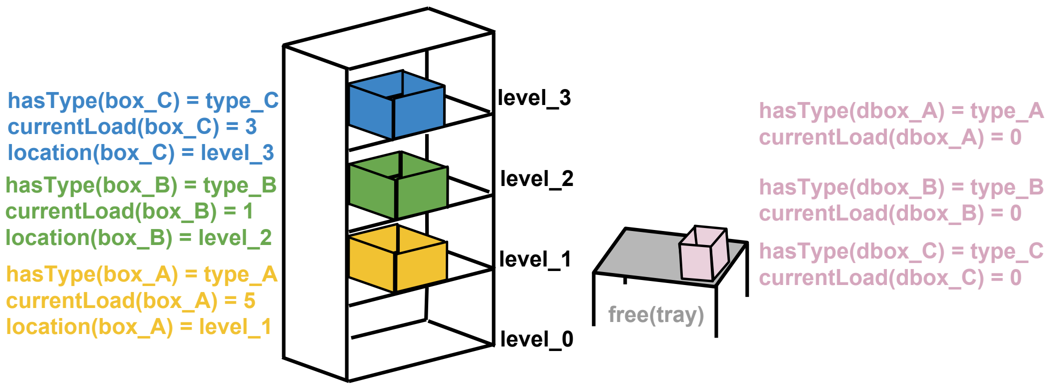


Figure 5.3: This figure shows an example of a possible initial state s_0.

5.2.3 Planner

The planner uses the information from the domain and the problem and generates a plan. A plan is basically a list of skills which need to be performed to solve the given problem. This list of skills is forwarded to the executive layer of the 3-TIER architecture.
Among a huge space of different planning languages and planners I decided

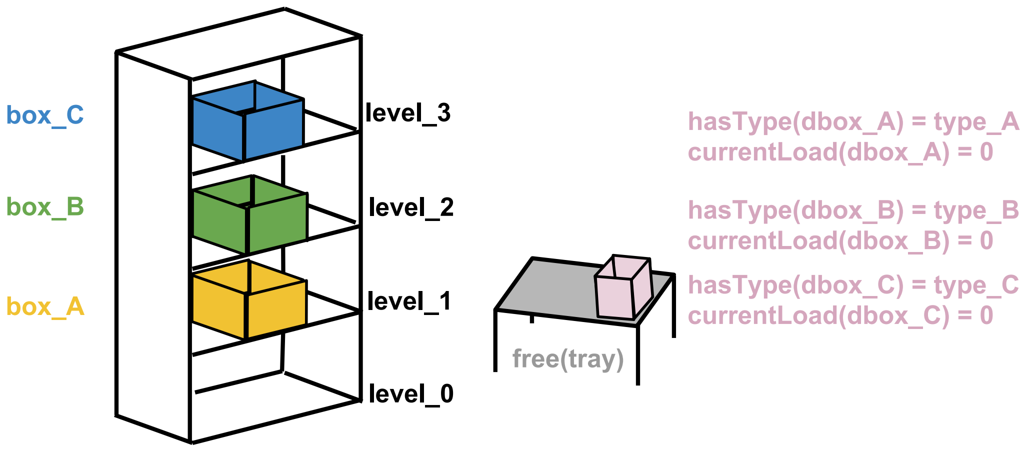


Figure 5.4: This figure shows an example of a possible goal state s_g.

to use the PDDL language. The authors of [4] showed that it is an appropriate choice for robotic tasks. Further PDDL is a standard language for planning problems and so very fast open source planners are available (like sgplan5 from Hsu et al. [13]). PDDL requires the definition of the robotic skills and functions within a domain file. The skill *graspItemFrom⟨ Box ⟩ PlaceTo ⟨ DeliveryBox ⟩On⟨ Tray ⟩* (short form *graspItem* used in) is defined in PDDL as shown in Listing 5.1.

Listing 5.1: The graspItem skill in the PDDL domain file.

```
(:action graspItem
:parameters (?b - transport_box ?d - delivery_box ?r - tray)
:precondition (and (= (has_type ?b) (has_type ?d))
                   (> (current_load ?b) 0)
                   (= (location ?b) (location ?r)))
:effect (and (increase (current_load ?d) 1)
             (decrease (current_load ?b) 1))
)
```

The definition of initial and goal state are defined in an extra problem file. For example all propositions that are used are defined at the beginning. For the example initial and goal state presented before in Section 5.2.2 the initial state is shown in Listing 5.2 and the goal state is in Listing 5.3.

Listing 5.2: The definition of propositons used in PDDL problem file.

```
(:objects
   box_A box_B box_C - transport_box
   dbox_A dbox_B dbox_C - delivery_box
   tray - tray
   level_0 level_1 level_2 level_3 - level
   typ_a typ_b typ_c - item_typ
)
```

Listing 5.3: The definition of a goal state used in PDDL problem file.

```
(:goal (and  (= (current_load dbox_A) 2)
             (= (current_load dbox_C) 1)
             (free tray)
        )
)
```

For the example initial and goal state presented before in Section 5.2.2 the planner could produce following output (see Listing 5.4)

Listing 5.4: Output of planner for example domain and problem. The name of the action is the first parameter

```
0 (MOVEBOXTOTRAY BOX_C LEVEL_3 TRAY)
1 (GRASPITEM BOX_C DBOX_C TRAY)
2 (MOVEBOXTOLEVEL BOX_C LEVEL_3 TRAY)
3 (MOVEBOXTOTRAY BOX_A LEVEL_1 TRAY)
4 (GRASPITEM BOX_A DBOX_A TRAY)
5 (GRASPITEM BOX_A DBOX_A TRAY)
6 (MOVEBOXTOLEVEL BOX_A LEVEL_1 TRAY)
```

5.3 Executive Layer

As the name already suggests, the executive layer handles the execution of single skills. Each skill is composed of skill primitives which are the fundamental building blocks of each skill. The executive layer knows about this composition and ensures that primitives are executed in right order to

guarantee a successful skill execution. The skill primitives theirselves are part of the lowest layer of the 3-TIER architecture.
The executive layer receives a list of skills from the planner. The list looks like the Listing 5.4. The executive layer executes each single skill by calling its skill primitives. Further it surveys the outcome of each primitive, aborts if an error occurs and reports an error to the planning layer. The only exception is the skill *inspectEnvironment* which is shown in figure 5.1. This skill is not contained in list received from the planner but added by the executive layer. It has no preconditions or effects. With adding the skill it is ensured that the environment around the robot is sensed before any other action is performed.
The composition of skill primitives for each skill is intrinsic knowledge of this layer. For the given skills following skill primitive composition is defined in Table 5.3.

Table 5.3: Skill primitive composition

skills	**moveBoxToRack**	**graspItem**	**moveBoxToLevel**
skill primitives	detectHandle graspHandle moveArmToSupportPose pullBox moveBox deliverBoxOnTray	detectItem graspItem deliverItem	detectHandle graspHandle moveArmToSupportPose pullBox moveBox deliverBoxOnLevel

Within this table the skill primitive composition of all skills are listed.

The executive layer is not only responsible for the execution of the primitives but also to verify their correct execution. So if one skill primitive fails, it must report this failure. The decomposition and verification procedure for all skills is shown in figure 5.5, figure 5.6 and figure 5.7. The primitives are visualized in all figures as rectangular boxes. The verifications are visualized as diamonds. The executive layer has an intrinsic knowledge about all skill compositions. So if the skill should be executed, the executive layer calls the skill primitives in right order. This is the decomposition of skills. Further the executive layer monitors the outcome of all skill primitives. If one skill primitive fails it aborts the skill and reports the failure to the high level. This is called verification.

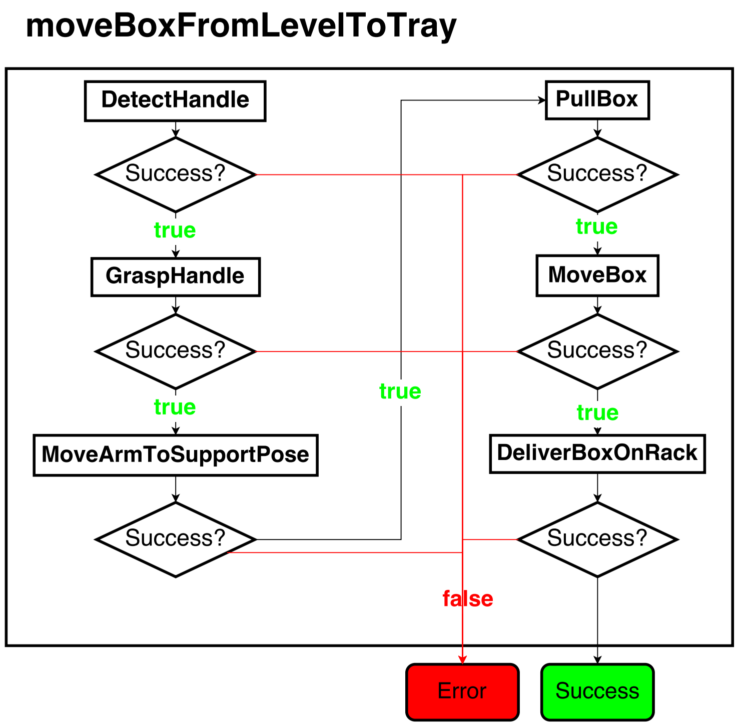


Figure 5.5: In this figure the skill moveBoxFromLevelToTray is visualized containing its primitives and verifications if everything works out. It can only report success or error.

5.4 Behavioral control

This layer holds all skill primitives. Within this layer the primitives are defined and programed for a specific robotic platform. All upper layers are still portable to other platforms. The primitives have a defined interface but are programed separately for all platforms.
A skill holds information about its execution and reports any error. But if a local recovery can be performed it can be done at this stage. Skill primitives usually try to recover from errors if possible. But if it is impossible skill primitives report their error. A skill primitive can be used by different skills. Our approach with skills and their decomposition in skill primitives is comparable to hierarchical task networks (HTN) presented in [11, Chapter

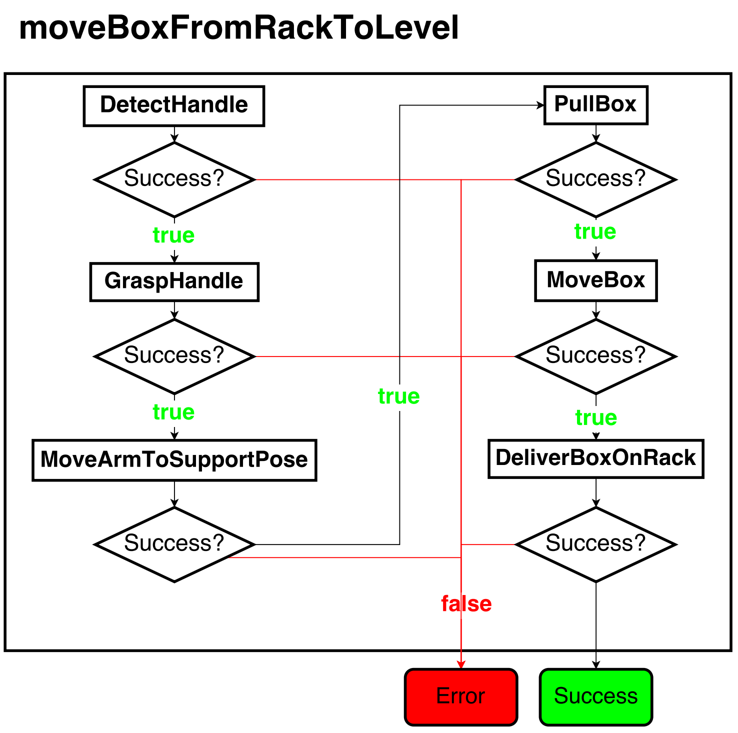


Figure 5.6: The skill moveBoxFromTrayToToLevel is shown. It looks similar to the skill moveBoxFromLevelToTray. Most primitives can be reused in this skill.

11]. Within this HTNs a problem is also reduced to different tasks. These tasks are then also decomposed into subtasks, in a similar manner as we do the decomposition of skills into skill primitives. These are following skill primitives:

- DetectHandle
- GraspHandle
- MoveArmToSupportPose
- PullBox
- MoveBox
- DeliverBoxOnTray
- DeliverBoxOnLevel
- DetectItem
- GraspItem

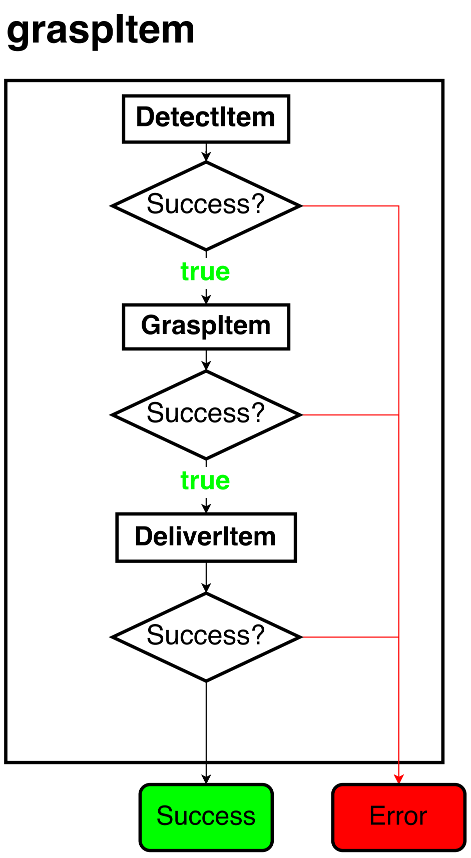


Figure 5.7: This graphic shows the graspItem skill. Its structure is the same like the one of the other two skills. Due to the purpose of this skill, it has different primitives.

- DeliverItem

5.4.1 Detect Handle

The aim of this skill primitive is to return the measured pose of the handle of a transport box. The primitive contains two main parts visualized in Figure 5.8. This primitive receives the expected place of the transport box and returns the measured poses. So the primitive searches at the expected place or the handle, this can be also understood as heuristic search. The expected place is sent form the executive layer and depends on which skill uses this primitive. For example, if the skill move⟨ Box ⟩From⟨ Level ⟩To⟨ Tray ⟩ is performed, then the place ⟨ Level ⟩ is forwarded to the primitive. Both parts are implemented as ROS action.

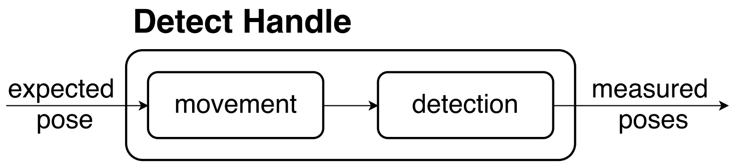


Figure 5.8: This figure shows the main parts of the skill primitive detect handle. It receives the expected place of the transport box. First the arm is moved there. Then the handle is detected and the measured poses are returned. If the detection fails for any reason, the poses are set invalid.

5.4.1.1 movement

It plans and executes a collision free path for the grasping arm from its current pose to the received pose. The grasping arm is defined as the hand or endeffector which should grasp the handle later on (in this system left or right arm). The choice if the grasping arm is the left or the right arm depends on the were the gripper is mounted (at first there was only one gripper available for both hands).

5.4.1.2 detection

This part detects the handle using the camera and the IR range sensor of the endeffector of the grasping arm. This step is shown in Figure 5.9. This part detects the handle within the image of the camera in the end effector. Simultaneously it measures the IR range distance to the transport box. Because the detection of the handle lays outside the scope of this work, this task is simplified and so the handle is marked with a yellow dot. So the detection itself is a simple color detection.

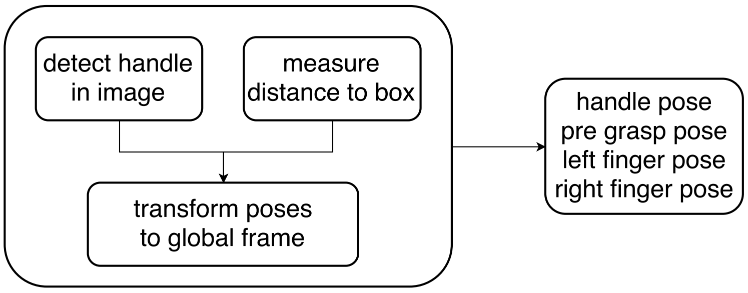


Figure 5.9: The detection part of the primitive detect handle is shown. It contains two measurements. One is performed in the image taken by the camera of the end effector. The other one is the measurement of the IR range sensor. Both are used to transform the poses in the global frame.

A pinhole camera model is used to transform the measured poses in the image to the camera frame. Further the poses in the camera frame are transformed to the global coordinate system. This transformation is possible due to the knowledge about the current robot state. The robot state publisher provides this transformation. The pinhole camera model is shown in Figure 5.10. The origin of the camera frame is named O_C. The origin of the global frame is named O_G. The origin of the image plane is in the upper left corner marked with vectors in x and y direction. The principle point P is shown in the image plain at the coordinates (px, py). The point $H_i(h_ix, h_iy)$ in the image plane is first transformed formed to the camera frame $H_C(h_Cx, h_Cy, h_Cz)$ and afterwards to the global frame $H_G(h_Gx, h_Gy, h_Gz)$. Transformation information from the GlobalFrame O_G to the CameraFrame O_C is provided by the robot state publisher. So the transformation $H_C \rightarrow H_G$ can be applied. If the pose H_i is detected within the image with the coordinates (h_ix, h_iy) and the IR sensor measured the range of r meters, the pose H_C in the camera frame can be calculated as follows:

$$h_Cx = \frac{H_ix - px}{f_x} \cdot r \tag{5.1}$$

$$h_Cy = \frac{H_iy - py}{f_y} \cdot r \tag{5.2}$$

$$h_Cz = r \tag{5.3}$$

The intrinsic camera parameters focal length f_x and f_y as well as the principle point $P(px, py)$ are known. So the coordinates of H_C can be calculated using equation 5.1, equation 5.2 and equation 5.3. The output of the de-

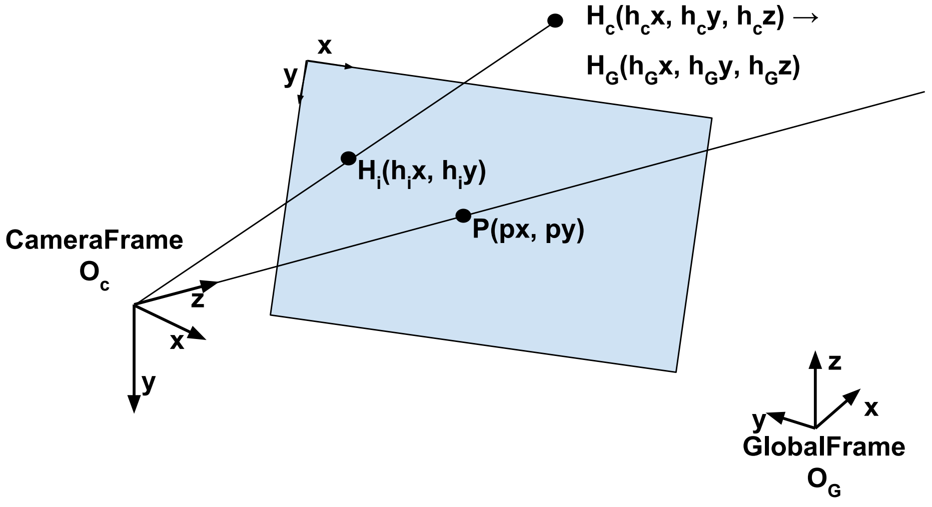


Figure 5.10: This figure shows the pinhole model.

tection step are following poses in the global frame: *handle pose*, *pre-grasp pose*, *left finger pose* and *right finger pose*. The *handle pose* is the actual pose of the handle. The *pre-grasp pose* is the pose where the endeffector should be, before it moves forward and closes the gripper. This pose is an offset in z-direction away from the *handle pose* in the camera frame. Further both fingers are detected in the image plane and their pose is also returned. These poses within the camera frame are shown in Figure 5.11. In the last step the poses are transformed in the global coordinate system.

5.4.2 Grasp Handle

This primitive performs the grasping of the handle. It uses the camera in the endeffector to track both fingers and the handle. The arm controls towards the *pre-grasp pose* using visual servoing. The visual servoing is needed, because first of all the detection of the handle could have some small

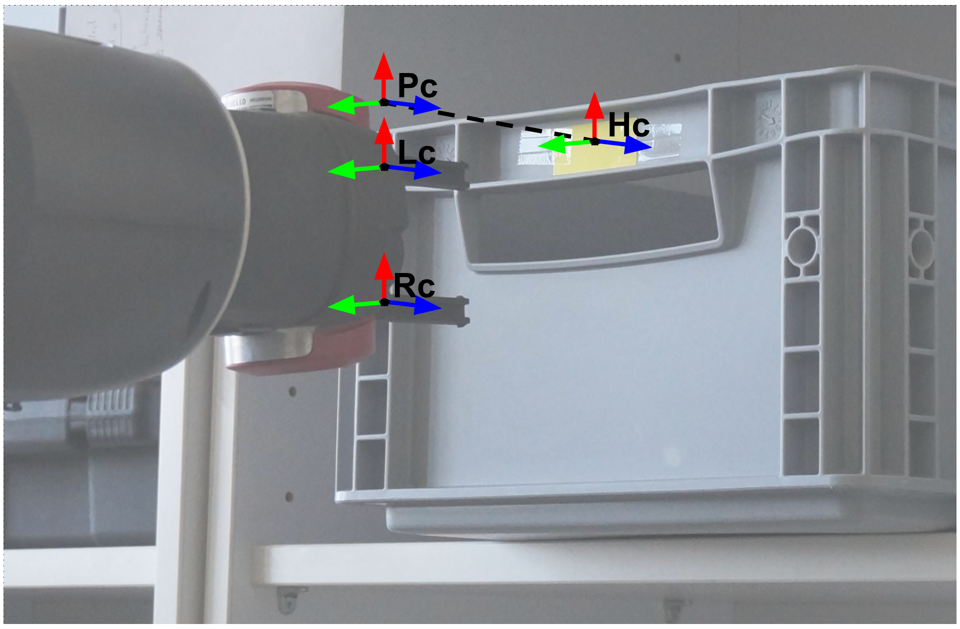


Figure 5.11: Here all detected poses are visualized within the camera frame. These are the *handle pose* H_C, the *pre-grasp pose* P_C, the *left finger pose* L_C and the *right finger pose* R_C

measurement errors. But the main reason for performing visual servoing is the inaccurate execution of trajectories. Once *pre-grasp pose* is reached, the endeffector is moved to the handle and gripper closes. Then it is validated if the gripper grasps the handle and if it does the success is returned. If it fails a local recovery is performed. After a finite amount of trails without success the skill returns the failure of execution. This procedure is shown in Figure 5.12.

5.4.2.1 Initial Measurement

This block measures the pose of the handle in the same way as described in section 5.4.1.2. If the handle cannot be detected, the error is reported. This inital measurement is very important after the local recovery was performed. Because the robot could have lost the box (fall to the ground)

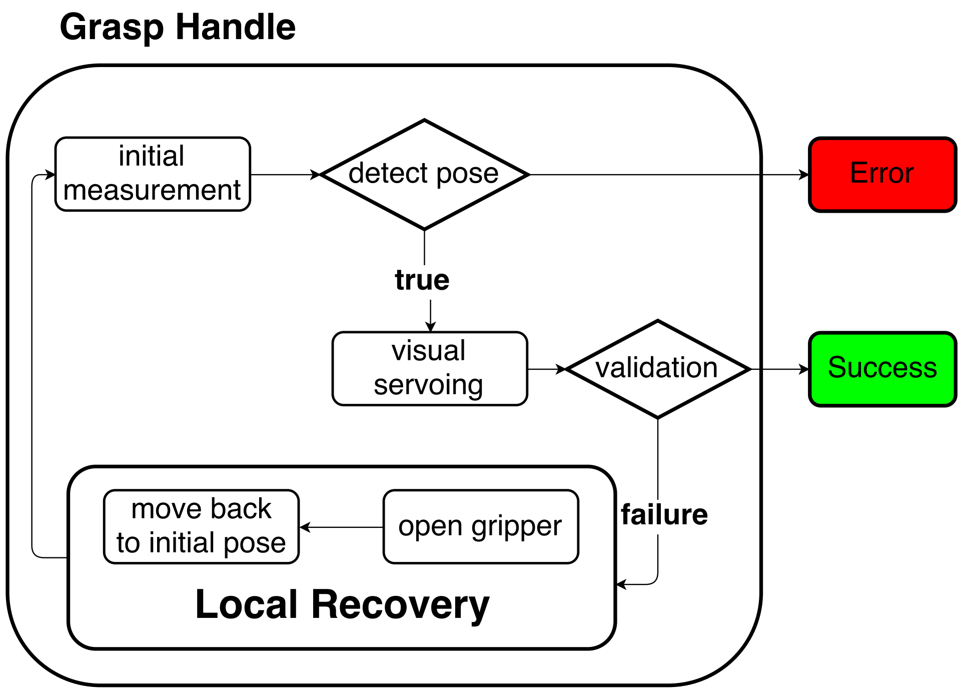


Figure 5.12: Within this figure the routine of the grasp handle skill primitive is shown.

and it is not where it is expected to be. After the initial measurement is successful completed, the visual servoing is performed.

5.4.2.2 Visual Servoing

The visual servoing routine is shown in detail in Figure 5.13. It measures the *handle pose* H_G, *pre-grasp pose* P_G and the current pose of the endeffector M_G with respect to the base coordinate frame. M_G is the midpoint between the *left finger pose* L_G and the *right finger pose* R_G with respect to the global coordinate system. The error e between the *pre-grasp pose* and the current pose of the endeffector is defined as follows:

$$e = \|M_G - P_G\| \tag{5.4}$$

If the error is smaller than a threshold ϵ ($e < \epsilon$), then the gripper moves towards the handle and closes the gripper. If $e \geq \epsilon$, the endeffector is moved along the correction vector M_G - P_G. This is repeated until $e < \epsilon$ holds.

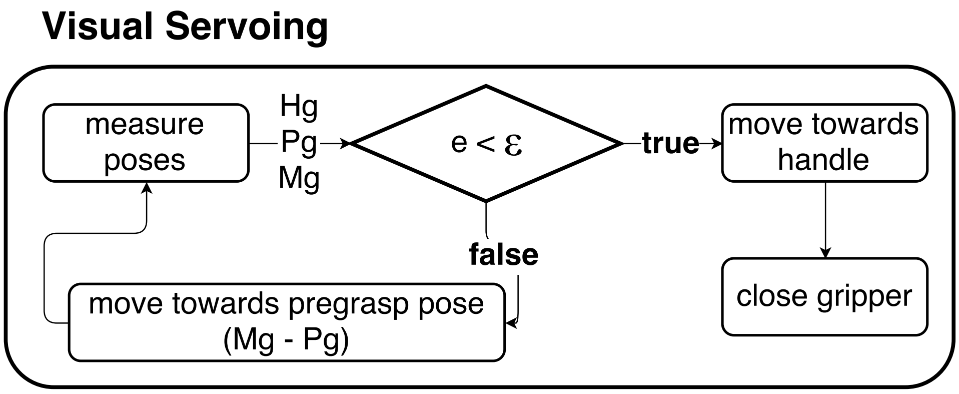


Figure 5.13: Here the visual servoing block of the grasp handle skill primitive is shown. First the *midpoint pose* M_G, the *pre-grasp pose* P_G and the *handle pose* H_G, with respect to the global coordinate system are measured. If M_G and P_G are approximately the same, the endeffector moves towards the handle and closes the gripper. Otherwise the endeffector is moved to P_G

5.4.2.3 Validation

The validation gives the primitive the opportunity to report about execution status and the execution layer can react to it.

5.4.2.4 Local Recovery

The local recovery is performed if the visual servoing part fails to grasp the handle. This recovery consists of two parts. First it opens the gripper. Then the movement of the arm to its initial pose is planed and executed. The initial pose is where the primitive was started. The movement back to the initial pose should always be possible. However if the movement cannot be planned or executed the primitive returns with an error. Through performing the initial measurement it is guaranteed that the box is still standing on the same place and is not lost. Afterwards the visual servoing is performed again.

5.4.3 Move to Support Pose

In Figure 5.16 the box manipulation with both arms is shown. It is not possible for Baxter to manipulate the box with a single arm, because it is too weak. So it needs to manipulate it with both arms. This primitive moves the second arm in a pose where it can support the grasping arm with manipulating the box. The scheme of this primitive is shown in Figure 5.14. First the pose is calculated. Then the trajectory towards this pose is computed and executed. If it is executed successfully the success is reported. If the trajectory cannot be planed or executed the planner is called again. Because a sample based planner with a random sampling strategy and random projection is used (LBKPIECE described in [28]) it makes sense calling it multiple times. After a finite times of trials the primitive reports the error.

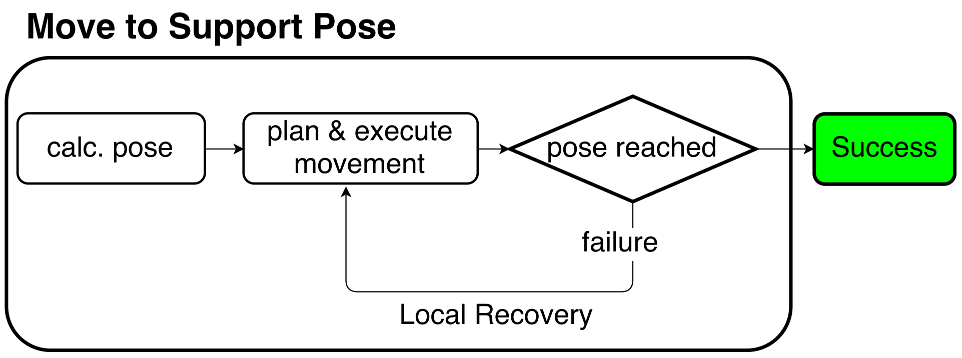


Figure 5.14: Here the scheme of the move to support pose skill primitive is shown. It returns success if the support arm reaches the previous calculated support pose.

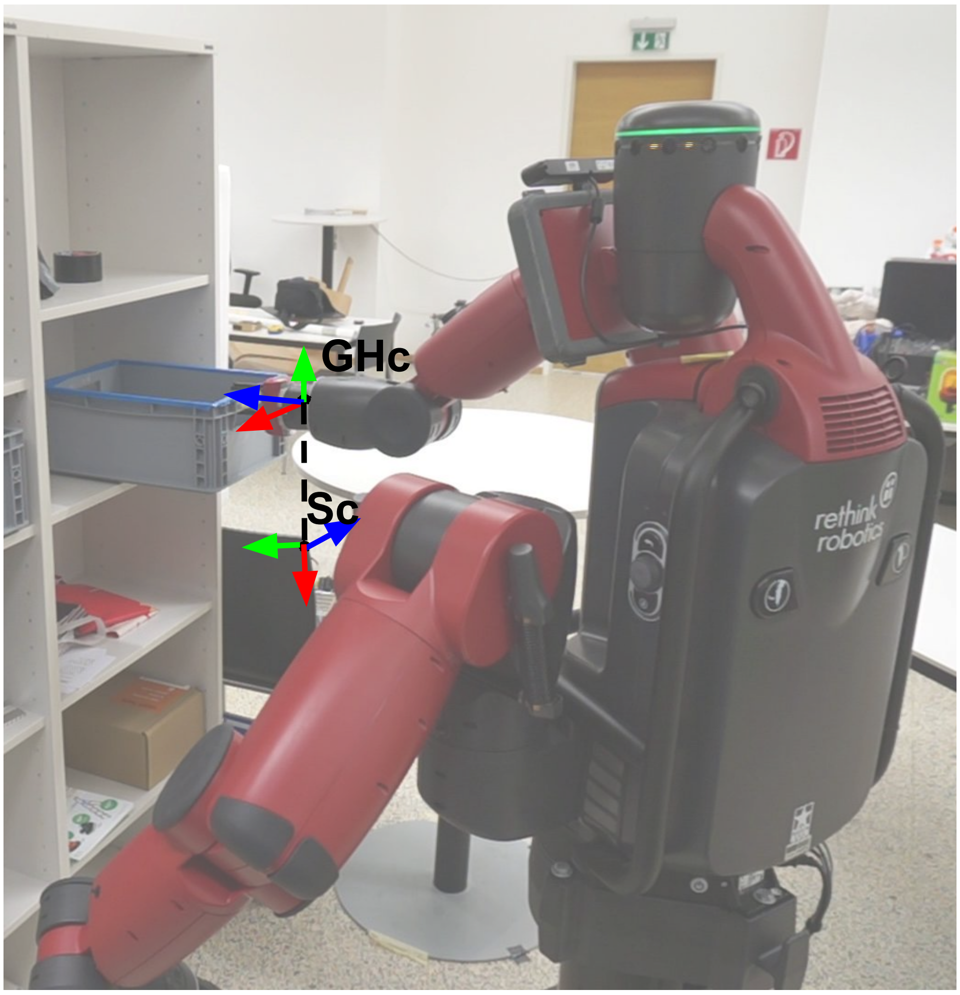


Figure 5.15: Here the current *grasp hand pose* (GH_C) with respect to camera frame is shown. The *support pose* S_C with respect to the camera frame is the GH_C rotated by 90°around the x-axis.

5.4.3.1 Calc. Pose

An example support pose is shown in Figure 5.15. The current support S_C with respect to the camera frame is calculated rotating the current grasp pose GH_C and translating it. First we discuss the rotation. α is the rotation around the x-axis, β is the rotation around the y-axis and γ is the rotation around the z-axis. Setting $\alpha = \frac{\pi}{2}$, $\beta = 0$ and $\gamma = \frac{-\pi}{2}$ gives the orientation shown in Figure 5.15. This orientation guarantees that the support arm is normal to the grasping arm. So the box movement can be supported by the support arm. The rotations around the axis are notated as $\mathbf{R_x}$, $\mathbf{R_y}$ and $\mathbf{R_z}$. The rotation matrices are defined as follows:

$$\mathbf{R_x}(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & cos(\alpha) & -sin(\alpha) \\ 0 & sin(\alpha) & cos(\alpha) \end{pmatrix} \tag{5.5}$$

$$\mathbf{R_x}\left(\alpha = \frac{\pi}{2}\right) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{pmatrix} \tag{5.6}$$

$$\mathbf{R_y}(\beta) = \begin{pmatrix} cos(\beta) & 0 & sin(\beta) \\ 0 & 1 & 0 \\ -sin(\beta) & 0 & cos(\beta) \end{pmatrix} \tag{5.7}$$

$$\mathbf{R_y}(\beta = 0) = \mathbf{I} \tag{5.8}$$

$$\mathbf{R_z}(\gamma) = \begin{pmatrix} cos(\gamma) & -sin(\gamma) & 0 \\ sin(\gamma) & cos(\gamma) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{5.9}$$

$$\mathbf{R_z}\left(\gamma = \frac{-\pi}{2}\right) = \begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{5.10}$$

The final rotation matrix $R_{S_c\_GH_c}$ is defined as:

$$\mathbf{R_{S_c\_GH_c}} = \mathbf{R_z}\left(\frac{-\pi}{2}\right) \cdot \mathbf{R_y}(0) \cdot \mathbf{R_x}\left(\frac{\pi}{2}\right) = \begin{pmatrix} 0 & 0 & -1 \\ -1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \tag{5.11}$$

The translation $\mathbf{t}_{\mathbf{S_c\_GH_c}}$ is just a movement in the negative y-coordinate. The length of this translation is the box height h_b:

$$\mathbf{t}_{\mathbf{S_c\_GH_c}} = \begin{pmatrix} 0 \\ -h_b \\ 0 \end{pmatrix} \tag{5.12}$$

So the transformation of the support pose $\mathbf{S_c}$ with respect to the grasping arm camera frame can now be written in homogeneous coordinates:

$$\mathbf{T}_{\mathbf{S_c\_GH_c}} = \begin{pmatrix} \mathbf{R}_{\mathbf{S_c\_GH_c}} & \mathbf{t}_{\mathbf{S_c\_GH_c}} \\ \underline{\mathbf{0}} & 1 \end{pmatrix} \tag{5.13}$$

Because the transformation for the grasping hand with respect to the global coordinate system is known $\mathbf{T}_{\mathbf{GH_c\_GH_G}}$, the transformation for the support pose with respect to the global coordinate system can be calculated:

$$\mathbf{T}_{\mathbf{S_c\_S_G}} = \mathbf{T}_{\mathbf{S_c\_GH_c}} \cdot \mathbf{T}_{\mathbf{GH_c\_GH_G}} \tag{5.14}$$

The position of the support pose is now given through the translation vector in the transformation $\mathbf{T}_{\mathbf{S_c\_S_G}}$. The orientation is gained from the rotation matrix.

5.4.4 Pull Box

This primitive moves the arm along the axis of the lower arm. The distance is given as input. After the movement it is checked if the gripper still holds the box or if the box was lost during the movement. The information if the gripper is still holding the object can be received from the Baxter SDK. When the gripper receives the signal to close, but the force which needs to be applied to close the gripper is larger then usual when closing it. So it is concluded that an object is grasped. In this primitive no local recovery is performed. Because if the box is lost during pulling, it is quite likely that it fall to the ground. There is no possibility to recover from that due to the action radius of Baxter. The scheme of this primitive is shown in Figure 5.17.

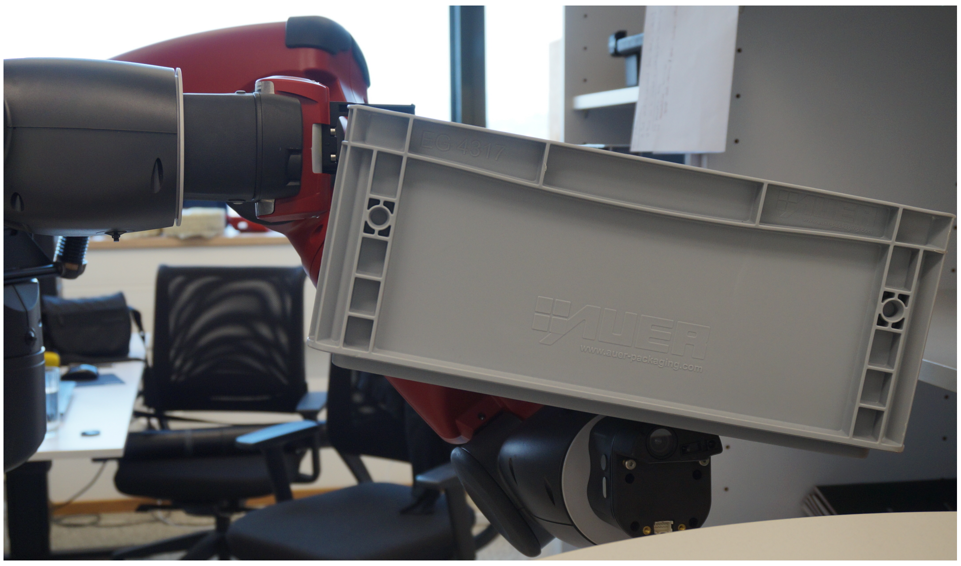


Figure 5.16: This figure shows Baxter carrying a box, where its right arm grasps the box. Its left arm supports supports the manipulation.

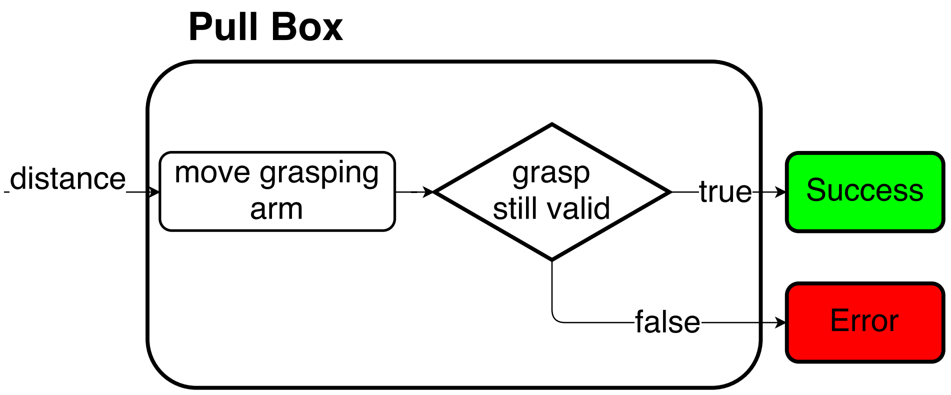


Figure 5.17: The scheme of the pull box skill primitive is shown. It moves the grasping arm for the given distance. It reports success if Baxter still holds the box after the movement. Otherwise error is returned.

5.4.5 Move Box

This skill primitive handles the manipulation of the box by both hands. It makes sure that both arms move in a way that the box is not lost. First it ensures that the arms keep the same distance and orientation from each other. Second it executes the movements time shifted. It calculates a single movement because both arms should perform the same movement. The routine is shown in Figure 5.18.

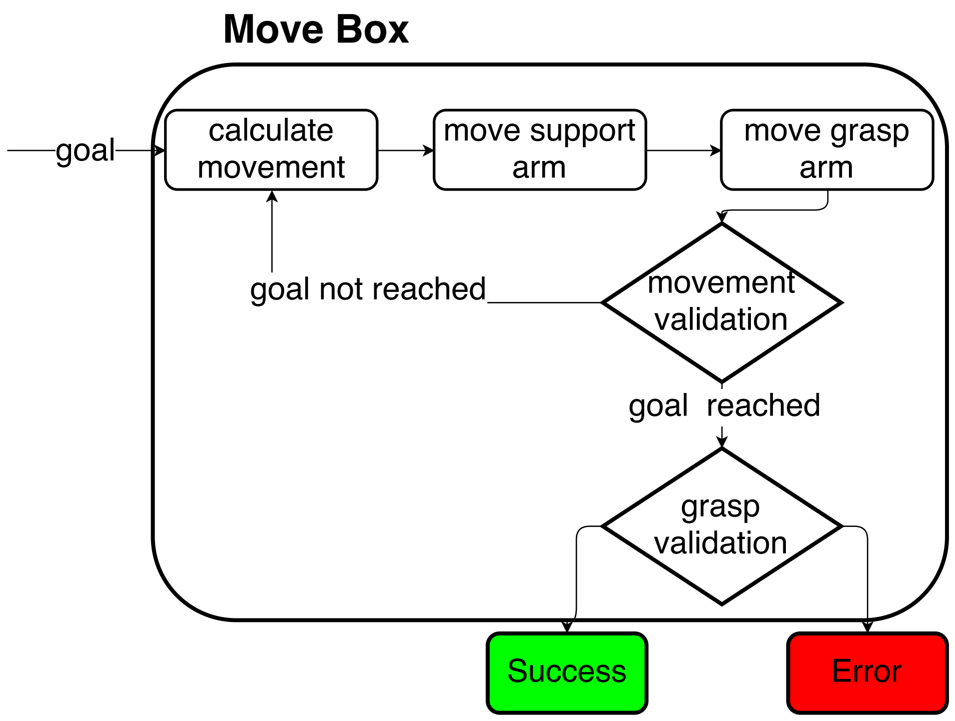


Figure 5.18: The scheme of the move box skill primitive is shown. It receives the goal as input. It calculates the movements for both arms. Then the support arm is moved first and the grasping arm afterwards. If the distance to goal vanishes and the box is still hold the primitive returns success. If the distance towards the goal is not approximately zero and both arms are still able to move, the new movement is calculated and the routine is executed again.

5.4.5.1 Calculate Movement

Within this block the next movement is calculated according to the current poses of the arms and the goal. The trajectories the arm should follow are visualized in Figure 5.18. First the trajectory for the support arm is calculated. It starts at the current position of the support arm, when the primitive is started. This position is marked as $\mathbf{S_S}$ (start position support arm). The goal position for the support arm ($\mathbf{S_G}$) is directly in front of the tray and known in advance. The trajectory the support arm should follow is marked in bright blue ($trajectory_S$). The grasping arm follows the same trajectory but starting at the current grasping arm position ($trajectory_G$). The $trajectory_G$ is marked in yellow. Because the arms move one after each other, starting with the support arm, one single movement is not allowed to be longer the box length. Otherwise the box would be dropped. So the arms move iteratively, along this trajectory. The first movement for the support arm is to the intermediate point $\mathbf{I_{S1}}$. The grasping arm follows to its intermediate point $\mathbf{I_{G1}}$. These points are calculated the following way. First the direction vector $\mathbf{d}$ id defined:

$$\mathbf{d} = \frac{\mathbf{S_G} - \mathbf{S_S}}{\|\mathbf{S_G} - \mathbf{S_S}\|} \tag{5.15}$$

The next intermediate point for the support arm is calculated using the current position of the support arm $\mathbf{S_C}$ and the box width w_b:

$$\mathbf{I_{Si}} = \mathbf{S_C} + \mathbf{d} \cdot w_b \tag{5.16}$$

The same holds for the next intermediate point for the grasping arm. Here the current position of the grasping arm is used $\mathbf{G_C}$:

$$\mathbf{I_{Gi}} = \mathbf{G_C} + \mathbf{d} \cdot w_b \tag{5.17}$$

The reason why the current position of the arms are used, instead of the previous intermediate point $\mathbf{I_{Gi-1}}$, or $\mathbf{I_{Si-1}}$ is that the execution is not that accurate to ensure that arm will exactly move to intermediate point. Further it is important to mention that for both arms their path is constrained in all orientations of the endeffector. So only translation is allowed, otherwise the box would be dropped. The maximum path length depends on the box proportions. It is not allowed to move too far with one arm only. It is an iterative movement.

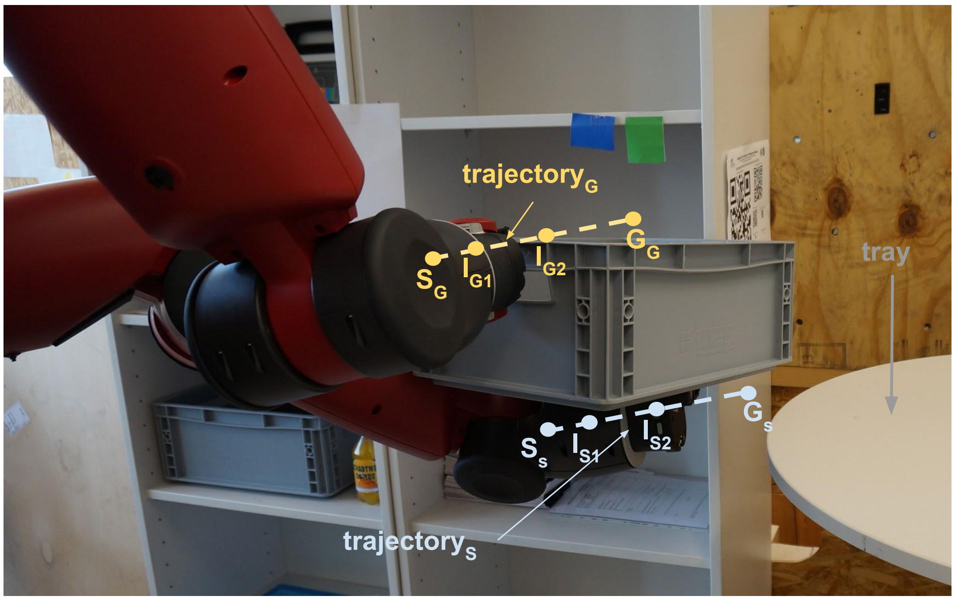


Figure 5.19: This figure shows the trajectories both arm should follow in order to get from the level to the tray.

5.4.5.2 Movement Validation

The validation consists of two parts. First it is checked if the goal has already been reached. This check is similar to the check in Section 5.4.2.2. if the error to the calculated pose is smaller than a threshold, it is assumed that the pose is reached. If the pose is not yet reached, it is checked if another movement is still possible and the next movement is calculated. If the pose is reached the grasp validation is performed.

5.4.5.3 Grasp Validation

If the box is still grasped after the movement success is returned. Otherwise the error is returned.

5.4.6 Deliver Box on Tray

This primitive handles the box delivery on the tray. First the box is moved towards the center of the tray. The grasp validation checks if the box is still grasped after moving it to the center. If not an error is returned. Then the box is turned towards the robot. When the handle is released, the pose is returned to the executive layer for later usage when the box's handle is grasped again. This scheme is shown in Figure 5.20. .

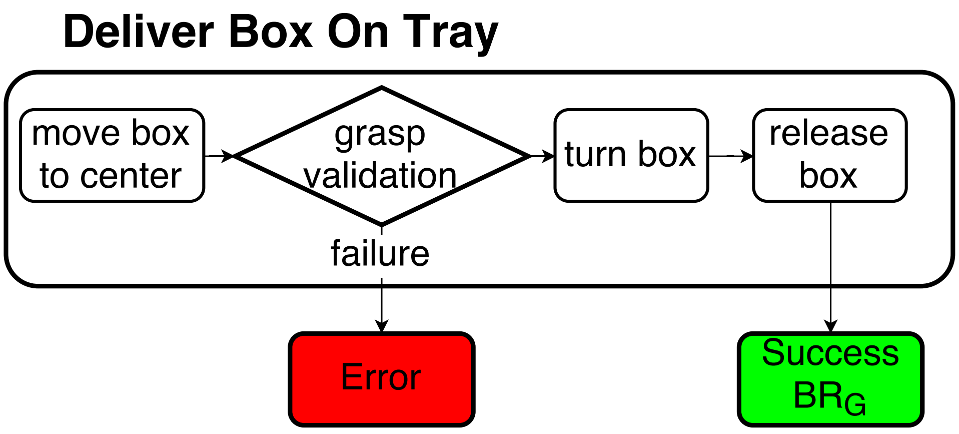


Figure 5.20: Here the different blocks of the deliver box on tray skill primitive are shown. First the box is moved towards the center, then turned and finally the handle is released. If the box gets lost an error is returned. Otherwise the *box release pose* BR_G with respect to the global coordinate system is returned.

5.4.6.1 Turn Box

In Figure 5.21 the box pose is shown schematically, after the box is moved towards the rack center. Due to friction the end where handle is not grasped does not move that far. This results in a box pose, where it is not easy to grasp the handle again. So the box is ”turned” towards the robot. This is not a real rotation of the box, but a small translation towards the origin of Baxter. This movement **m** is visualized as yellow vector in Figure 5.21. Again, due to friction, the box's end, where the handle is grasped, moves further which results in twisting the box. This movement is small but it ensures that the handle can be grasped again. The handle pose H_G with respect to the global coordinate system is known, because Baxter grasps the handle. Because the should be turned only and not lifted, only the x $h_G x$ and y $h_G y$ coordinates are considered. The origin of the global coordinate system is marked in Figure 5.21 with O_G. The movement has the length l, which is only about a few centimeters (0.02m). So the movement is calculated as follows:

$$\mathbf{m} = \begin{pmatrix} mx \\ my \\ 0 \end{pmatrix} = \frac{\begin{pmatrix} -h_G x \\ -h_G y \\ 0 \end{pmatrix}}{\left\| \begin{pmatrix} -h_G x \\ -h_G y \\ 0 \end{pmatrix} \right\|} \cdot l \tag{5.18}$$

5.4.6.2 Release Handle

First the gripper is opened. Then the arm is moved back for half of the box length along the movement shown in Figure 5.22. This movement is represented in the grasping arm camera frame. As shown in Figure 5.11 the z-axis is along this movement. So in the camera coordinate system frame the movement for releasing the box **mr** is represented as follows:

$$\mathbf{mr} = \begin{pmatrix} 0 \\ 0 \\ \frac{-w_b}{2} \end{pmatrix} \tag{5.19}$$

The current pose of the arm is now the box release pose BR_G, which is returned.

5.4.7 Detect Item

The aim of this primitive is the detection of the item pose with the hand camera. Due to the environmental setup it is not possible to detect it with

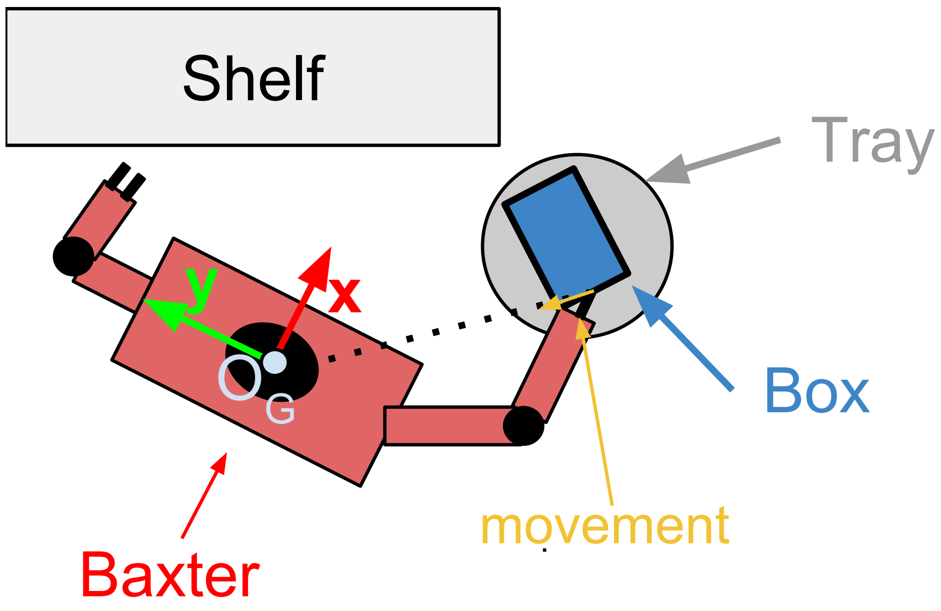


Figure 5.21: The box is moved in the middle of the rack. Due to friction the box end, where the handle is grasped is moved further than the the other end of the box.

the head mounted camera due to perspective occlusions like the box borders. The box's approximate pose is given as input. The pose is uncertain due to some uncertainties within the arm's movement. So first the exact pose of the box need to be determined. Based on this pose the arm can be moved over the box in a way that the hand camera can look into the box. If the arm is over the box it detects the item with the hand camera. If it cannot detect the item it purely turns the wrist and tries to detect it again. The item could be occluded by one of the fingers within the camera image. The output of this primitive is the exact box pose and the item pose with respect to global coordinate system. This procedure is shown in Figure 5.23. If either the box or the item cannot be detected, the primitive returns the failure.

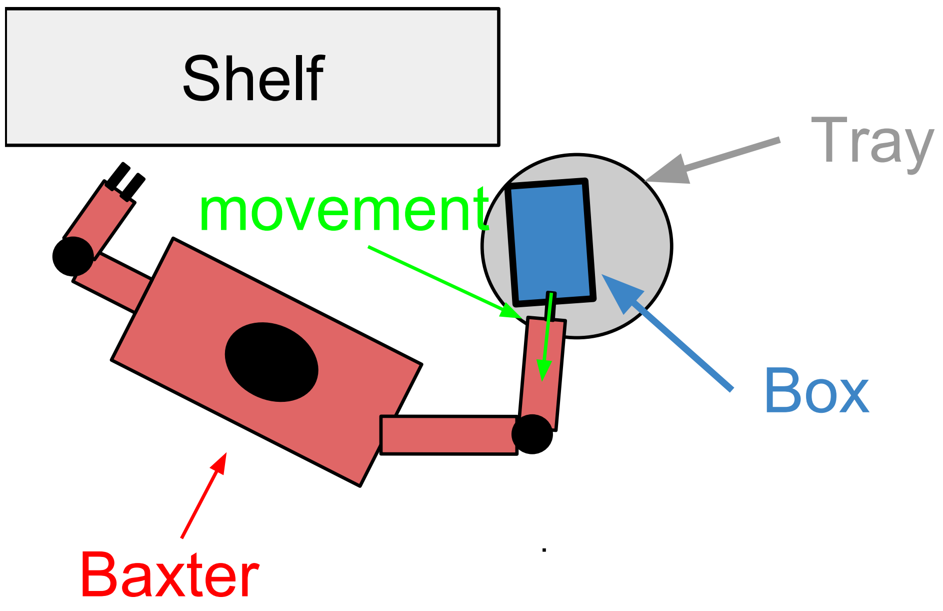


Figure 5.22: After the box is turned the box's handle is released and the grasping arm moves away from the handle.

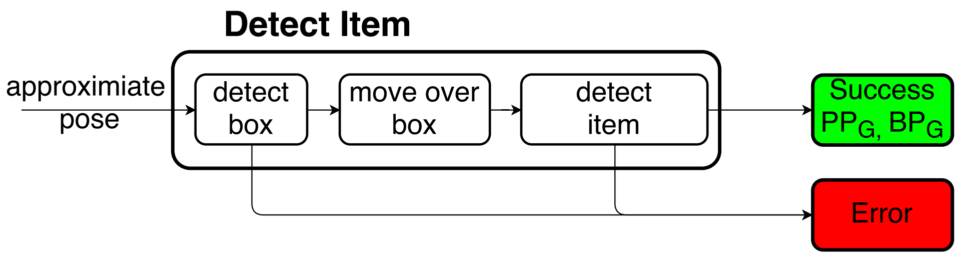


Figure 5.23: In this figure the different blocks of the detect item skill primitive are shown. First it detects the transport box, then it moves the arm over the box and detects the item using the hand camera. If the skill succeeds the *item pose* PP_G and the *box pose* BP_G ares returned.

5.4.7.1 Detect Box

In order to detect the box the information from the head mounted 3D camera is used. In particular only the point cloud is used. A coarse to fine template matching approach is used for detection.

1. Preprocessing
   Because the approximate pose of the box is known, all points which are too far away are cut out. Further it is known that the box should be standing on the table. So all large planes are also cut out. Euclidean clustering is performed on the remaining points. The output of the preprocessing step are the different clustered point clouds.
2. Coarse Alignment
   A template of the box is aligned using the fast point feature histogram (FPFH) and the sample consensus initial alignment (SAC-IA) described in [24]. The alignment is performed on each single clustered point cloud from the preprocessing step. The point cloud which gives the highest score and the detected pose there is forwarded to fine alignment step.
3. Fine Alignment
   The fine alignment receives a single point cloud and an approximate pose. An iterative closest point (ICP) algorithm is performed.

The last two steps are repeated 15 times and the best result is taken so a very robust detection is guaranteed. Because the SAC-IA alignment contains random sampling, it makes sense to repeat this two steps more often. We show in Section 7.4 that this chain outperforms a single ICP or FPFH and SAC-IA algorithm in our setup.

5.4.7.2 Move Over Box

Once the box pose is known, possible poses over the box for the grasping endeffector are calculated. These poses ensure that the box content will be visible in the camera image of the endeffector's camera. The poses are in increasing height above the box. Beginning with the closest pose to the box, it is tested , if a motion plan can be found. If a plan is found the trajectory is executed, else the next pose is tested. Some example poses can be seen in

Figure 5.24. In Figure 5.26 Baxter is shown while inspecting the box.
For the inspect box pose calculations the box pose with respect to global coordinate system is given with BP_G. The transformation of the box frame with respect to global coordinate system is denoted with $\mathbf{T_{B\_G}}$. The first inspection pose IP_1 is directly over the box origin (see Figure 5.25), z-direction pointing towards the origin (content is visible in camera frame). This transformation can be presented as $\mathbf{T_{IP_1\_B}}$

$$\mathbf{T_{IP_1\_B}} = \begin{pmatrix} \mathbf{R_{IP_1\_B}} & \mathbf{t_{IP_1\_B}} \\ \underline{\mathbf{0}} & 1 \end{pmatrix} \tag{5.20}$$

The translation $\mathbf{t_{IP_1\_B}}$ is only a translation in z-direction, because the first pose is directly over the box:

$$\mathbf{t_{IP_1\_B}} = \begin{pmatrix} 0 \\ 0 \\ 2 \cdot h_b \end{pmatrix} \tag{5.21}$$

The rotation is a 180°rotation around the x-axis of the box, so the rotation matrix $R_{IP_1\_B}$ is defined as:

$$\mathbf{R_{IP_1\_B}} = \mathbf{R_x}\left(\alpha = \pi\right) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix} \tag{5.22}$$

The inspection poses are sampled starting from IP_1 in direct line to IP_{end}. The position of IP_{end} can be calculated with respect to the box frame (for right grasping hand), given the box depth d_B, box width w_B and box height h_B:

$$\mathbf{t_{IP_{end}\_B}} = \mathbf{t_{IP_1\_B}} + \begin{pmatrix} -d_B \\ -w_b \\ 1.5 \cdot h_B \end{pmatrix} \tag{5.23}$$

For the inspecting with the left hand the translation would look like following:

$$\mathbf{t_{IP_{end}\_B}} = \mathbf{t_{IP_1\_B}} + \begin{pmatrix} -d_B \\ w_b \\ 1.5 \cdot h_B \end{pmatrix} \tag{5.24}$$

So possible poses are now chosen in uniform distance from each other on the direct line between the positions of IP_1 and IP_{end}. The direction of the direct line is described with the normed vector $\mathbf{dir}$. The translation from IP_1 to the n^{th} point, with a distance d is calculated the following:

$$\mathbf{dir} = \frac{\mathbf{IP_{end}}}{||\mathbf{IP_{end}}||} \tag{5.25}$$

$$\mathbf{t_{IP_1\_IP_n}} = d \cdot \mathbf{dir} \tag{5.26}$$

If the n^{th} position with respect to the first pose IP_1 is chosen, the orientation is calculated as described now. First the connecting vector $\mathbf{c}$ to the origin position of the box BP_{IP_1} from $IP_{n_{IP_1}}$ position is calculated:

$$\mathbf{c} = \mathbf{BP_{IP_1}} - \mathbf{IP_{n_{IP_1}}} \tag{5.27}$$

The rotation around the y-axis is given:

$$\beta = atan\left(\frac{c_x}{c_z}\right) \tag{5.28}$$

and the rotation around the x-axis is given with:

$$\alpha = atan\left(\frac{c_y}{c_z}\right) \tag{5.29}$$

So the rotation matrix for the n^{th} pose is defined as following:

$$\mathbf{R_{IP_1\_IP_n}} = \mathbf{R_y}(\beta) \cdot \mathbf{R_x}(\alpha) \tag{5.30}$$

and the transformation is given with:

$$\mathbf{T_{IP_1\_IP_n}} = \begin{pmatrix} \mathbf{R_{IP_1\_IP_n}} & \mathbf{t_{IP_1\_IP_n}} \\ \underline{\mathbf{0}} & 1 \end{pmatrix} \tag{5.31}$$

and the transformation to the global coordinate system:

$$\mathbf{T_{IP_1\_G}} = \mathbf{T_{IP_1\_IP_n}} \cdot \mathbf{T_{IP_n\_IP_G}} \tag{5.32}$$

The desired position and orientation for IP_n can now be gained from the transformation $\mathbf{T_{IP_1\_G}}$.

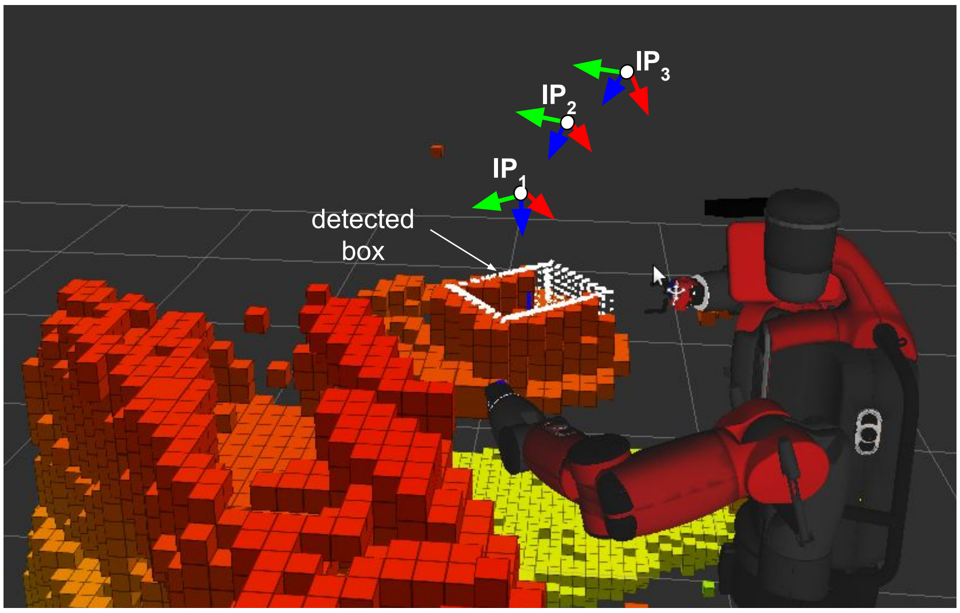


Figure 5.24: In this figure some example inspection poses (IP_1, IP_2, IP_3) are marked. This poses are orientated in a way, that ensures the box content being on the camera image of the endeffector's camera. The detected box is shown as white point cloud. The orange voxels indicate the collision scene.

5.4.7.3 Item Detection

Because the item detection itself is out of the scope of this work, the item is marked with an augmented reality (AR) tag. An AR tag is shown in Figure 5.27. Because the AR tags and the dimensions of the tags in real world are known the object pose can be detected using the camera image only. No additional information like depth need to be measured, because the pose can be reconstructed from the tag template and the knowledge of the dimensions. AR tags are also used in mobile phone AR applications. So there are already existing open source libraries for the detections of AR tags. This eases the detection also.

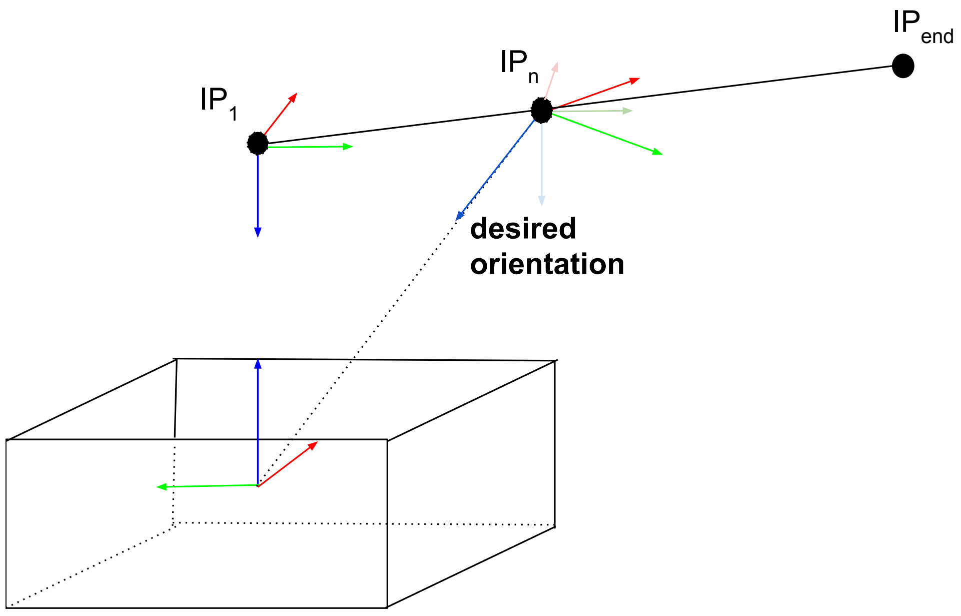
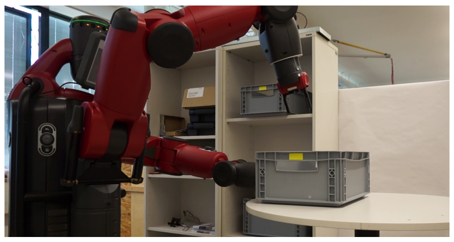


Figure 5.25: This figure shows the inspection points in more detail.


(a)Baxter inspecting the box with it's endeffector camera.

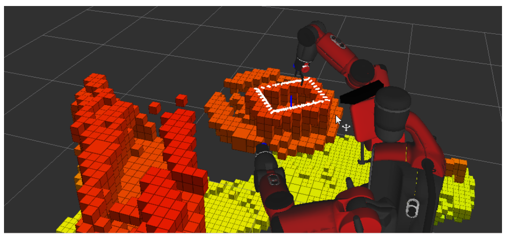

(b)Baxter inspecting the Box visualized in RViz. The white point cloud indicates the detected box. The orange voxels visualize the collision scene.

Figure 5.26: This figure shows Baxter inspecting the box in reality as well as visualized in RViz.

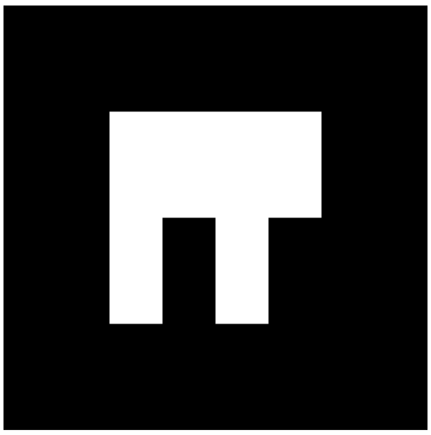


Figure 5.27: An example AR tag.

5.4.8 Grasp Item

This primitive receives the *box pose BP_G* and the *item pose PP_G*. Based on those poses it calculates good grasping points. It grasps the item and checks if the movement is correctly executed and the item grasped. If not it return an error. Then the item is placed to the delivery box. Again the movement is checked. If all works out the primitive succeeds, otherwise it returns a failure. This procedure is shown in Figure 5.28.
It calculates grasping points using the Eigengrasp approach from [18]. Using the ten best grasping points to calculate a *pre-grasp pose* to the corresponding grasping points. It plans a trajectory for all grasping points and takes the first one which is feasible. It validates the grasp and finally places the item. If no grasping points can be found, or no valid trajectory can be planned or the grasp validation fails the skill primitive returns an error.

5.4.8.1 Calculate Grasping Points

The planner plans within the Eigengrasp space, which leads to a reduction of the search space [18]. The reduction to the Eigengrasp space is useful for manipulators with high degree of freedom (e.g. humanoid hands). Grasp postures can be represented as linear combination of Eigengrasps. A single

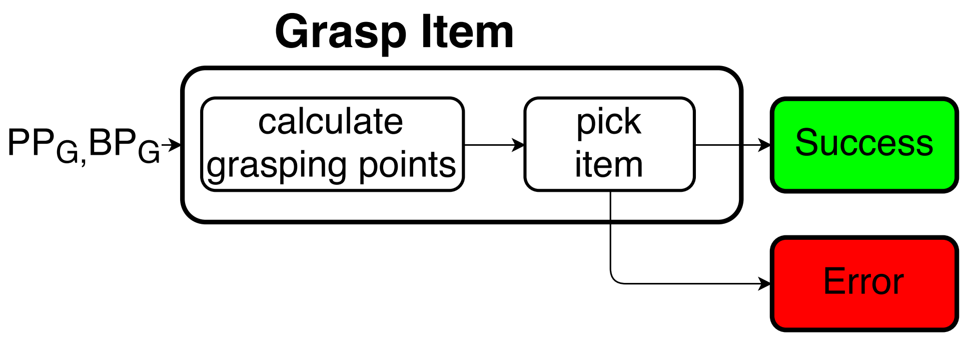


Figure 5.28: In this figure the different blocks of the grasp item skill primitive are shown.

Eigengrasp can be understood as direction vector of the grasping motion in joint space. However this should not effect a parallel jaw gripper that much, but this approach plans good grasps also for this kind of grippers. For planning grasps first regions on the gripper are sampled. These sampled points will be considered as good contact points for the gripper. The energy function in the Simulated annealing optimization strategy is designed to bring those presampled contact points in contact with the object, that should be grasped. So if points on the fingertips and the palm are sampled, then the planner plans enclosing grasps. The planner minimizes the distance of these points towards the object and optimizes the angle between the surface normals and the the contact regions of the gripper. The planner has also a feasibility check, if the gripper is in collision with the environment. So only feasible grasps are considered. Due to the nature of Simulated Annealing in early steps nearly all feasible grasps are considered. The algorithm converges towards good grasps in later simulation steps.
There was no parallel jaw gripper modeled within GraspIt!, so we modeled Baxter's gripper and lower arm. Because Baxter can nearly turn its wrist around 360°, we also added the lower arm model to the gripper. So feasible grasps can be computed which will not result in a collision with the environment. And it does not effect the execution the grasps, because Baxter can turn the wrist to nearly all arbitrary angles. In Figure 5.29 the GraspIt! graphical interface is shown. GraspIt! provides the possibility to run own physics simulation which calculates the forces applied by the gripper on

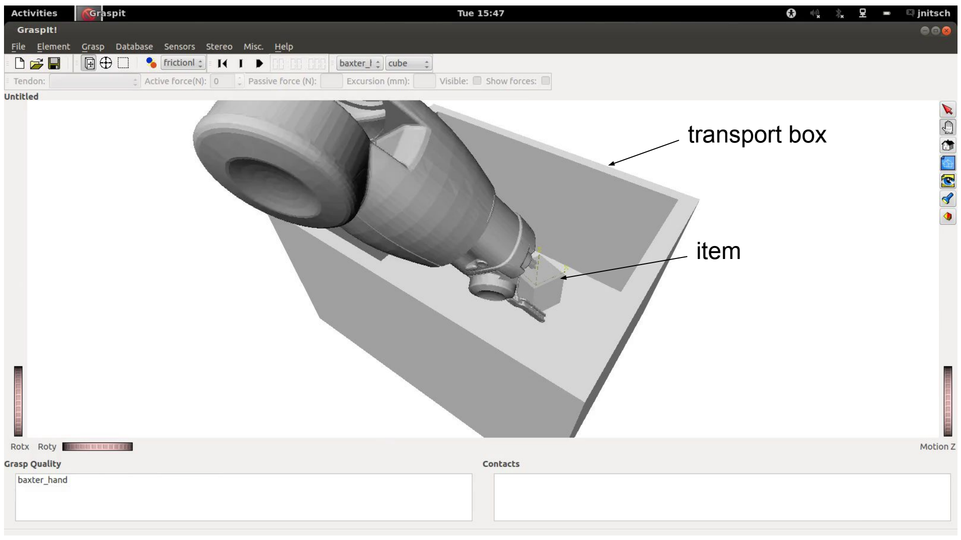


Figure 5.29: In this figure one possible grasping pose for Baxter is shown. The transport box model, as well as the item model are loaded. The planner works on this environment to ensure grasps which are not in collision with the box.

an object. We visualize within Section 7.6 planned and executed grasps on different objects.

5.4.8.2 Pick Item

First the transport box borders are included into the collision scene as collision objects. This ensures that the transport box is not moved or touched be the grasping arm. Then for the corresponding grasping points the arm motion is calculated to reach this grasping posture. If for one grasp a feasible arm motion is found, the motion is executed and the pick is performed. The pick is closing the gripper.

5.4.9 Deliver Item

This skill primitive consits of only one block shown in Figure 5.30. In order to be able to place the item, the gripper needs to hold the item. In the place item block the trajectory for the grasping arm to the delivery box is planned and executed. The box borders are still recognized as collision objects. So the arm does not touch the box while moving over the delivery box. The placing of the package is simply opening the gripper while being over the box. So the item drops into the delivery box. After this movement the box borders are removed from the collision scene again. If no trajectory can be planned or the execution is aborted an error is returned. Otherwise the primitive returns the success.

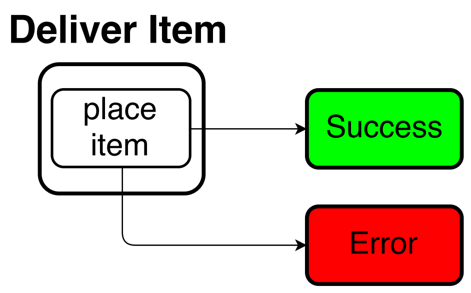


Figure 5.30: In this figure the only block of the eliver item skill primitive is shown.

5.4.10 Deliver Box On Level

This skill primitive delivers the box back into the level. Therefore it is first pushed into the level of the shelf, so that it securely stands inside the level. After this step a grasp validation is performed to make sure the box was not lost during the first step. If the box is lost the primitive returns the error. Otherwise it can be assumed the box stands now securely on the level of the shelf. Now the grasp is released and the arm is moved back the same way as it is described in Section 5.4.6.2. The primitive returns the successful execution. This is flow is visualized in Figure 5.31.

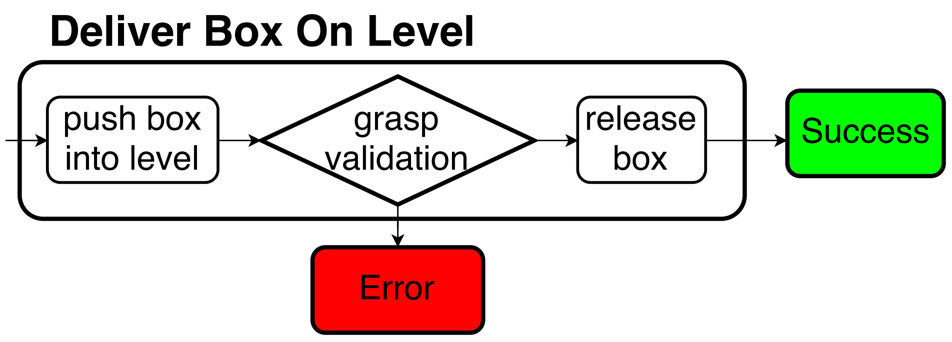


Figure 5.31: This figure shows the deliver box on level primitive. First the box is pushed into the level, then it is turned so that it stands normal towards the shelf. Finally the grasp is released and the arm is moved back.

5.4.10.1 Push Box Into Level

For a better understanding of this part in Figure 5.32a the robot is shown before pushing the box into the level and after it pushed into to level (see Figure 5.32b). So a trajectory needs to be calculated from the current grasping pose (see Figure 5.32a) to the end pose (see Figure 5.32b). First the orientation of the end pose is calculated. This calculation is the inverse computation from Section 5.4.3. Now the support pose is given and the grasping arm pose is calculated. So this can be thought of a rotation around z-axis by 90°($\gamma = \frac{\pi}{2}$) and a rotation around the y-axis by 90°($\beta = \frac{\pi}{2}$) and no rotation around the x-axis ($\alpha = 0$). So the rotation matrices are:

$$\mathbf{R_x}(\alpha = 0) = \mathbf{I} \tag{5.33}$$

$$\mathbf{R_y}\left(\beta = \frac{\pi}{2}\right) = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{pmatrix} \tag{5.34}$$

$$\mathbf{R_z}\left(\gamma = \frac{\pi}{2}\right) = \begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{5.35}$$

So the final rotation matrix from the support orientation to the grasping orientation $\mathbf{R}_{\mathbf{GH_c\_S_c}}$ is:

$$\mathbf{R}_{\mathbf{GH_c\_S_c}} = \mathbf{R_z} \cdot \mathbf{R_y} \cdot \mathbf{R_x} = \begin{pmatrix} 0 & -1 & 0 \\ 0 & 0 & 1 \\ -1 & 0 & 0 \end{pmatrix} \tag{5.36}$$

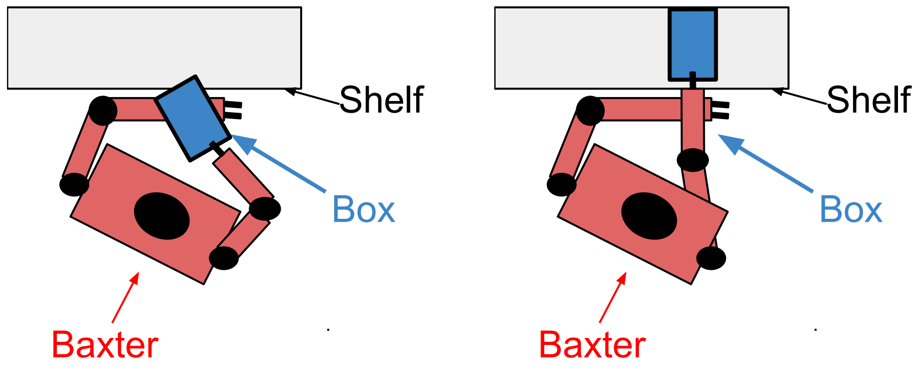
This rotation matrix $\mathbf{R}_{\mathbf{GH_c\_S_c}}$ is the inverse of $\mathbf{R}_{\mathbf{S_c\_GH_c}}$ (see Section 5.4.3):

$$\mathbf{R}_{\mathbf{GH_c\_S_c}} = \mathbf{R}_{\mathbf{S_c\_GH_c}}{}^{-1} \tag{5.37}$$

The translation is same as the movement $\mathbf{m}$ in Section 5.4.6.2. But now it is not moving away from the box, but pushing the box. So the z-coordinate has a positive sign. Because the box is pushed in the level, length of the translation vector $\mathbf{t}_{\mathbf{GH_c\_S_c}}$ is w_b.

$$\mathbf{m} = \begin{pmatrix} 0 \\ 0 \\ w_b \end{pmatrix} \tag{5.38}$$

The pose GH_G can be retrieved the same way as in Section 5.4.3.



(a)Baxter before pushing the box into the level. (b)Baxter after pushing the box into the level.

Figure 5.32: This figure shows Baxter pushing the box into the level.

5.5 Layout for 3-TIER high-level architecture as HSM

A general mission control for a 3-TIER high level can be implemented as hierarchical state machine (HSM). If the plan gets started, first a planner (representing the planning layer) is called with a current environment description and the goal which should be achieved and a domain description. It creates a list of skills and forwards this list to an instance of a planning execution. This represents the executive layer. This is a nested state machine. The skills are called in the order of the received plan. If the skills succeed, the success is reported. If an error occurs the planner is informed about the error. An overview of the HSM is shown in Figure 5.33. Each single skill is decomposed into its skill primitives. So the skill itself is also implemented as state machine. Each primitive represents a state which is followed by a monitor state. These monitor states report if the skill primitive aborts with an error. Then the skill is aborted and the error is reported to the *PlanExecution* state machine. The state machine of the skills is represantively shown in Figure 5.34.

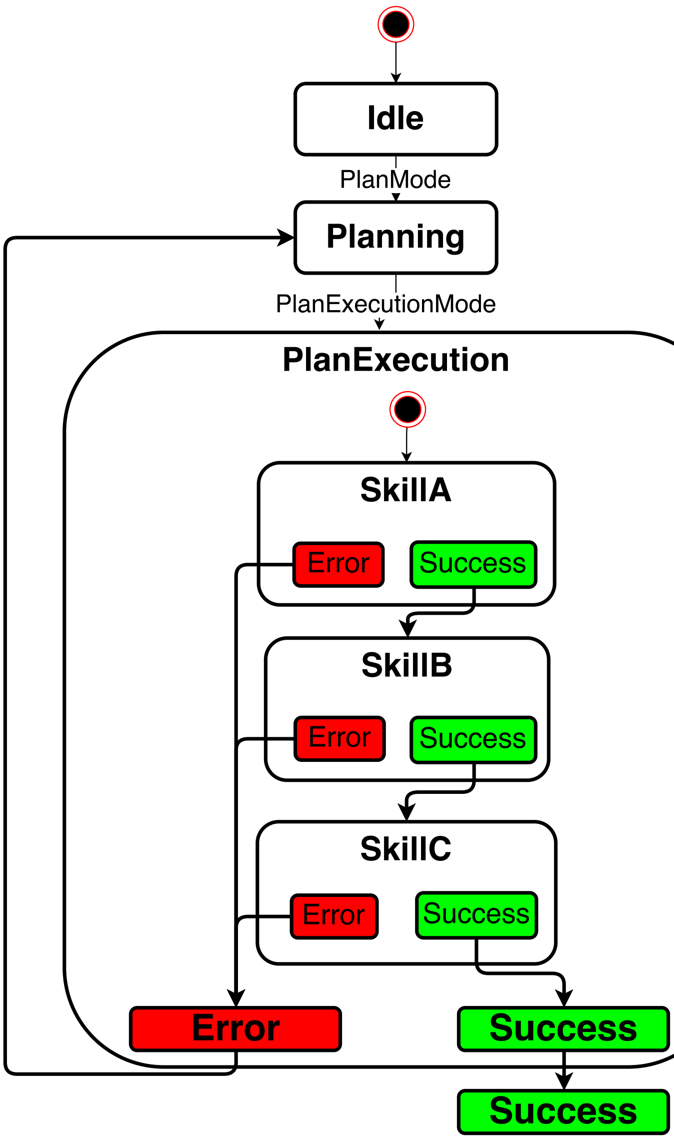


Figure 5.33: This figure shows a general HSM of a 3 TIER architecture. Once started it enters the planning instance, which gets a domain and problem description as input. It delivers a plan (list of skills) to the executive instance. Within the *PlanExecution* state, which is itself a state machine. Each skill represents a state of this state machine.

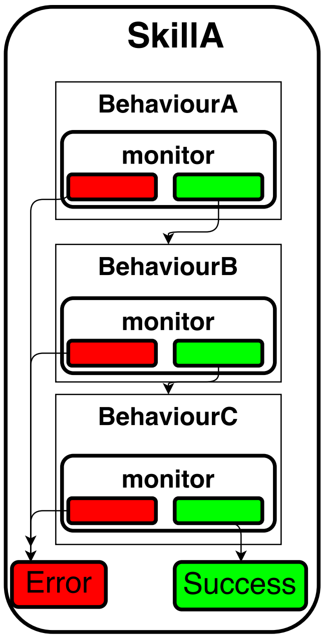


Figure 5.34: This figure shows a general skill of the *PlanExecution* state machie. Each skill is state machine for its self. Each primitive it consists of represented as own state followed by a monitor state.

6 Implementation

In this chapter the implementation of the system design, presented in Chapter 5, is explained in detail. This system is developed in C++ utilizing ROS.

6.1 Environment

All poses of fixed objects in the environment like the one of the shelf or the tray are known in advance and stored in rosparam files. Also geometric specifications (e.g: box dimensions w_b, d_b and h_b) are stored in such files. These parameter files are loaded at system start up to the ROS parameter server. There the parameters can be directly accessed by the nodes, which need the information about it. Further we implemented a library *config_envrionment* which provides convenient functions to load multiple parameters, which are often used together, at once with one function call. This includes loading the box or tray dimensions. Further this library provides one function to load information about the tf frame names of the grasping and the support arm.

6.2 High Level Node

Within this section 6.2 the node (*baxter_control* node) which implements the high level is discussed in detail. The HSM for the order picking problem looks like shown in Figure 5.34. The top level state machine is implemented

following the pattern proposed in [38]. We introduce a base state *BaxterControlState* which defines the interface for transitions between different states, so called *modes*. Further this base state defines the interface for the arguments a state takes. It also defines a *timedExecution* function. This function is called periodically, if the state is active. All states (*Idle*, *Planning*, *PlanExecution*) in the top level state machine are derived from this base state. Further all states hold the same instance of the *BaxterControl* class. This class manages all resources. The ressources can be accessed through interfaces of this class. All skill primitives are implemented as ROS actions. So the *BaxterControl* class holds the clients for these actions. So this actions clients need not be implemented in all states and nested states, but simply accessed through this resource class. Further this class holds information about the current state and calls the *timedExecution* from the current active state periodically.
The hierarchical state machines are implemented straight forward and access also the resources from the *BaxterControl* class.

6.2.1 Class Structure

The class diagram of the top level state machine is visualized in Figure 6.2. The ressource class holds action clients from following functionalities: *Detect Handle*, *Grasp Handle*, *Pull Box*, *Move To Support Pose*, *Move Box*, *Deliver Box On Tray*, *Deliver Box On Level*, *Scan Environment*, *Link Follower*, *Move To Level*, *Detect Box*, *Move Over Box*, *Grasp Item*, *Place Item*, *Move To Pose*. The implementation details of these functionalities are discussed in Section 6.3. During a successful execution, the state machine starts with the *Idle* state, as the name already suggests it idles during its *timedExecution* function until the start of the order is invoked. Then the state machine transitions to the *Planning* state. Within this *timedExecution* function, the planning itself is performed. The domain and problem descriptions are saved as plain text files. The problem description grounds the objects which occur in this setup. Some of this objects are marked in Figure 6.1. This setup contains two boxes (*box_0* and *box_1*) which are stored respectively on *level_0_0* and *level_0_1*. The dimensions of the box are stored in the $transport\_box\_config.yaml$ file. The poses of the levels in the world are stored in the $levels.yaml$. Further the tray defined in the problem file is described by its position and the radius (*environment_config.yaml*) because it is a round table. The current load of the transport boxes is also defined in the problem description as well as the different item types which occur in this setup (*type_A* and *type_B*). The parameters which are stored in different *∗.yaml* files can be accessed as described in Section 6.1. So for the current implementation only one order

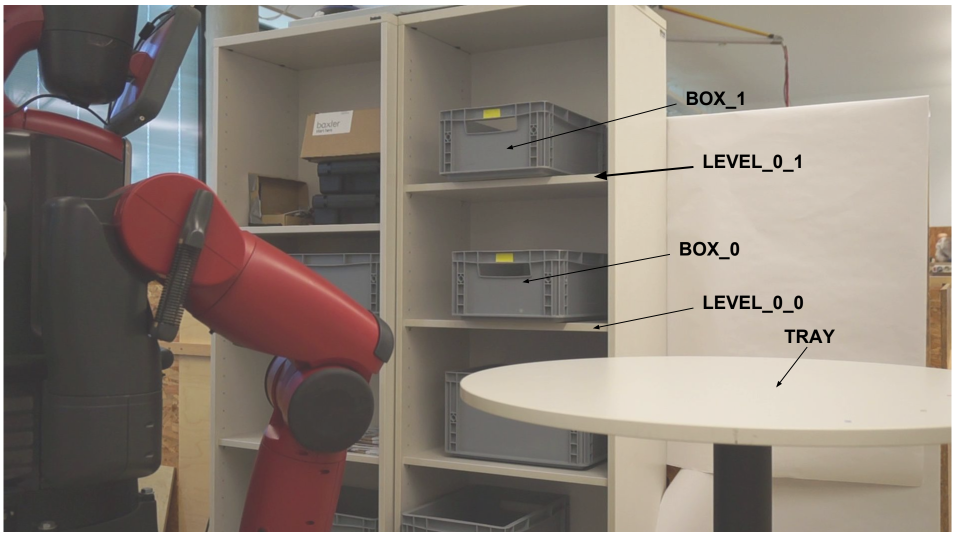


Figure 6.1: In this figure objects in the system are visualized with their corresponding name in the domain description file.

is considered. This order is saved in the problem description file. This order asks for one item of type *type_A*. If this order is finished, the next order can only be executed if the problem file is replaced with a new problem file containing the next order. The planner is a pre-compiled executable. So if the planning is invoked, this high level process is forked. In the child process the planner is executed with the problem and domain description as input. The output of the planner is piped into a file descriptor. The main process waits for the child process to terminate. Then it parses the output of the planner and saves the list of skills as *std::vector<command>*. A *command* is a struct and defined as shown in Listing 6.1. It holds the name of the command and its arguments as strings.

Listing 6.1: *command* struct

```
struct command
{
  std::string command_str;
  std::vector<std::string> command_args;
};
```

Planning state transitions to the *PlanExecution* state and passes the list of planned skills to this state. Within the *PlanExecution* state the hierarchical state machines for the skills and furthermore their primitives are called. As long as no error occurs, the *PlanExecution* state remains active. If an error occurs the *PlanExecution* reports this error and transitions back to the *Idle* state.

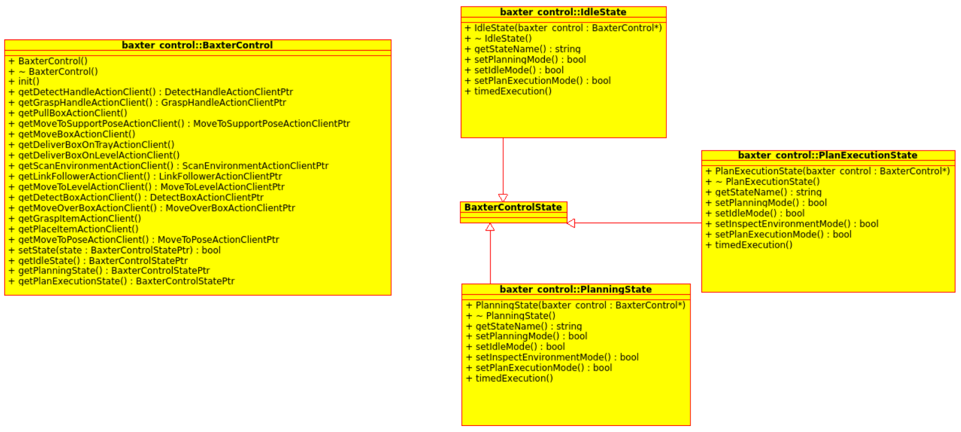

baxter control::BaxterControl
+ BaxterControl()
+ ~ BaxterControl()
+ init()
+ getDetectHandleActionClient() : DetectHandleActionClientPtr
+ getGraspHandleActionClient() : GraspHandleActionClientPtr
+ getPullBoxActionClient()
+ getMoveToSupportPoseActionClient() : MoveToSupportPoseActionClientPtr
+ getMoveBoxActionClient()
+ getDeliverBoxOnTrayActionClient()
+ getDeliverBoxOnLevelActionClient()
+ getScanEnvironmentActionClient() : ScanEnvironmentActionClientPtr
+ getLinkFollowerActionClient() : LinkFollowerActionClientPtr
+ getMoveToLevelActionClient() : MoveToLevelActionClientPtr
+ getDetectBoxActionClient() : DetectBoxActionClientPtr
+ getMoveOverBoxActionClient() : MoveOverBoxActionClientPtr
+ getGraspItemActionClient()
+ getPlaceItemActionClient()
+ getMoveToPoseActionClient() : MoveToPoseActionClientPtr
+ setState(state : BaxterControlStatePtr) : bool
+ getIdleState() : BaxterControlStatePtr
+ getPlanningState() : BaxterControlStatePtr
+ getPlanExecutionState() : BaxterControlStatePtr
baxter control::IdleState
+ IdleState(baxter control : BaxterControl*)
+ ~ IdleState()
+ getStateName() : string
+ setPlanningMode() : bool
+ setIdleMode() : bool
+ setPlanExecutionMode() : bool
+ timedExecution()
BaxterControlState
baxter control::PlanExecutionState
+ PlanExecutionState(baxter control : BaxterControl*)
+ ~ PlanExecutionState()
+ getStateName() : string
+ setPlanningMode() : bool
+ setIdleMode() : bool
+ setInspectEnvironmentMode() : bool
+ setPlanExecutionMode() : bool
+ timedExecution()
baxter control::PlanningState
+ PlanningState(baxter control : BaxterControl*)
+ ~ PlanningState()
+ getStateName() : string
+ setPlanningMode() : bool
+ setIdleMode() : bool
+ setInspectEnvironmentMode() : bool
+ setPlanExecutionMode() : bool
+ timedExecution()

Figure 6.2: This figure shows the top level state machine. *BaxterControl* holds resources needed in all different layers of the state machine so that they can be accessed easily and spread this information. The abstract *BaxterControlState* class defines all the transitions and the *timedExecution* function for the implemented states. The implemented top level states are *IdleState*, *PlanningState* and the *PlanExecutionState*

6.3 Skill Primitives and System Functionalities

In this section the skill primitives and basic functionalities of the system are described in detail. We start with the general functionalities skill primitives can use. They are either provided by Baxter or other frameworks. We continue with a deeper look in the implementation of the primitives itselves.

In Figure 6.3 functionalities and hardware parts are visualized that skill primitives can directly access. The skill primitives can access these functionalities. They directly read out status information published by Baxter or can directly control different Hardware parts. Orange colored parts are provided by Baxter, the yellow colored part visualizes MoveIt! component, the purple colored part belongs to the GraspIt! framework, the green part represents the Asus 3D camera using the OpenNI driver and blue parts belong to our Industrial Grasping system.
The first hardware component which can be controlled via a topic is Baxter's head. In order to control the head a *baxter_core_msgs/HeadPanCommand* message is published on the topic */robot/head/command_head_pan*. Within the *HeadPanCommand.msg* the target angle of Baxter's head and the speed how fast it should move there is defined. The data of the IR sensor in the grasping (in our case right) endeffector can be accessed through listening to the */robot/range/right_hand_range* topic. There the standard *sensor_msgs/Range* message is published which includes the currently measured range. The camera image is also transmitted via a ROS image message (*sensor_msgs/Image*) on the topic */cameras/right_hand_camera/image*. Further the cameras intrinsic parameter (see *sensor_msgs/CameraInfo*) are also published on a separate topic named */cameras/right_hand_camera/camera_info*. This parameter are important for calculating the back projection from image pixel into world coordinates. The next very important part on the endeffector is the gripper. With listening to the gripper state (*/robot/end_effector/right_gripper/state*) it can be easily evaluated if the gripper currently grasps an object or is empty. On this topic a *EndEffectorState.msg* message is published. This message contains the boolean flag *gripping*. This flag is *true* if the gripper grasps an object and false otherwise (described in Section 5.4.4). The gripper state topic can also be used to control the gripper. When publishing the 'closing' or the 'opening' command, the gripper immediately moves.
The arms are not accessed directly, but using the MoveIt! interface as described in Section 4.2. Further the MoveIt! node is also connected to the point cloud provided by the Asus 3D camera. Therefore the MoveIt! node keeps track of the environment and its changes, as well as on recognized objects.

MoveIt! stores the information about the environment and the robot state in the *planning scene*. The *Planning Scene Monitor* manages the *planning scene*, like handling sensor updates and registers objects. All in all the *planning scene* contains objects, which are allowed to be touched and manipulated. It also contains objects not allowed to be touched, which are called collision objects. So all the collision objects are further referred as collision scene. A visualization of the collision scene can be seen in Figure 6.4.
Skill primitives can also access the point cloud (see *sensor_msgs/PointCloud2*) delivered by the Asus 3D camera via the topic */camera/depth/points*. Alternatively not the raw point cloud need to be used, but a filtered one by the MoveIt! node (*move_group/filtered_cloud*). MoveIt! excludes all visible robot parts in the Asus 3D camera image from the depth map. Therefore it uses information about the current robot state as well as the meshes of the robotic parts. Further skill primitives have the opportunity to access the GraspIt! framework via ROS services. Therefore the GraspIt! framework was wrapped into a ROS node and basic functionalities like loading a gripper, collision and grasping objects can be used through service calls. Further the Eigengrasp planner, can also be started through a service and the planned grasps are returned. In particular the gripper can be loaded via the service */ros_graspit_interface/load_local_gripper* and the Eigengrasp planning is invoked with the */ros_graspit_interface/generate_grasps* and the response of this service returns the generated grasps.

Skill primitives and core functionalities in our system are grouped in four modules: perception, manipulation, grasping and arm movements. In the following sections we describe the primitives and functionalities in the former described order. All nodes provide their functionalities as ROS actions, or ROS services.

6.3.1 Perception Module

Here all nodes are described, which handle perceptional tasks.

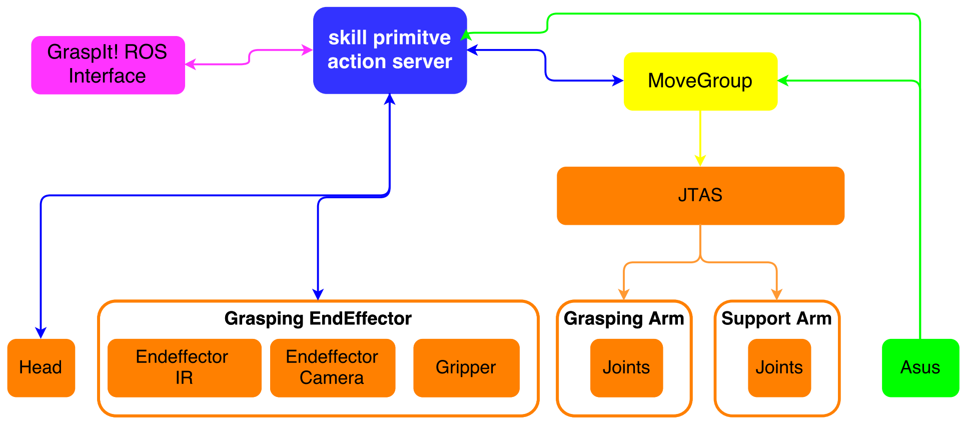


Figure 6.3: This figure shows the basic functionalities and information that is provided by the system.

6.3.1.1 Detect Handle Node

This node implements the handle detection functionality described in Section 5.4.1.2. Therefore it subscribes to the camera image topic for detecting the handle. Further it subscribes the camera info topic and the IR range sensor topic to back project the detected pose in the global frame. As soon as the box's handle gets detected a cuboid of (a little bit larger than) box size is published to the MoveIt! *Planning Scene Monitor*. It is also marked as an object, which is allowed to be touched by Baxter. This ensures that afterwards a motion plan can be generated to grasp the handle. The box published to the *Planning Scene Monitor* is larger than the real box. This ensures the the visual servoing routing afterwards encounters no problems with false detections of the collision scene around the box. In Figure 6.4 the published box can be seen. When the box is published, the collision scene in this area is removed. It is important to notice, that these voxel are removed when the collision scene updates this region, this happens only if a new point cloud in this area is captured. So in order to really update the collision scene, where the box is inserted, the Asus 3D camera(or any other sensor) must deliver a new point cloud in that region.

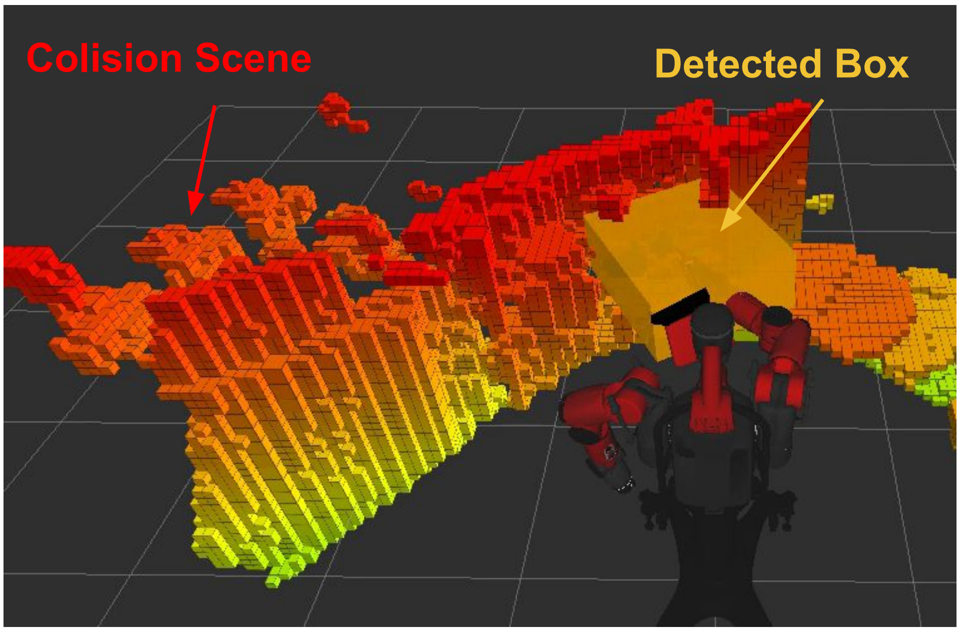


Figure 6.4: This figure shows the detected collision scene with the published box. Baxter is allowed to touch the box but not the collision objects

6.3.1.2 AR Tag Detection Node

As described in Section 5.4.7.3 the item is marked with an AR tag. In order to detect the AR tag the ROS wrapped version[1] of the library ARToolKit[2] (version 2.72.1) is used. This is a commonly used library for detecting AR tags. The ROS wrapper subscribes to the camera image and info topic and publishes detected tags on the topic *ar_pose_marker*.

6.3.1.3 Detect Item Node

This node implements the functionalities described in Section 5.4.7. When this action is invoked, it listens to to *ar_pose_marker* topic. If no tag is detected

[1]please see `http://wiki.ros.org/artoolkit`

[2]for more information about this library please see the documentation on `http://artoolkit.org/`

it utilizes the MoveIt! interface to turn the wrist of the grasping arm around 90°. Therefore the current pose of the endeffector is taken and the rotated pose is calculated. This calculated pose is packed into a trajectory message and send via the MoveIt! interface to the JTAS. The JTAS is described in detail in Section 4.5.3. This is the only node, which does not use any trajectory planner. If the AR tag is detected it returns the pose as ROS action result and continuously publishes the pose as ROS tf. This is an easy way to steadily publish the pose of an object and utilizing the transformation functionalities of the tf package. This item is then presented as cubic object in the world. This object is also registered in the MoveIt! framework as object which is allowed to be touched and not a collision object anymore. This is visualized in Figure 6.5.
For a better understanding of the interaction of this node with the AR tag detection a sequence diagram is shown in Figure 6.6. First the detect item node requests an AR tag pose from the AR tag detection. Because it has not detected any tags yet, it returns no pose. Meanwhile the detection is running on current camera images. If one tag would be detected this pose is stored and returned on request. The detect item node turns the wrist to capture another view of the content of the box. It requests an AR pose and again non has been detected. The wrist is turned for the third time and a tag pose is requested. Now a tag has been registered. The detect item node returns the success.

6.3.1.4 Detect Box Node

Here the coarse to fine approach described in Section 5.4.7.1 is implemented. This node does not subscribe to the raw point cloud published by the OpenNI driver but uses the already preprocessed point cloud from MoveIt! framework. This point cloud has the big advantage, that the parts belonging to the robot are already filtered out. It uses the PCL interfaces for the Euclidean clustering (*pcl::EuclideanClusterExtraction*), FPFH extraction (*pcl::FPFHEstimation*), SAC-IA alignment (*pcl::SampleConsensusInitialAlignment*) and the ICP (*pcl::IterativeClosestPoint*). For this approach previous generated box template is needed. This template is generated using the open source

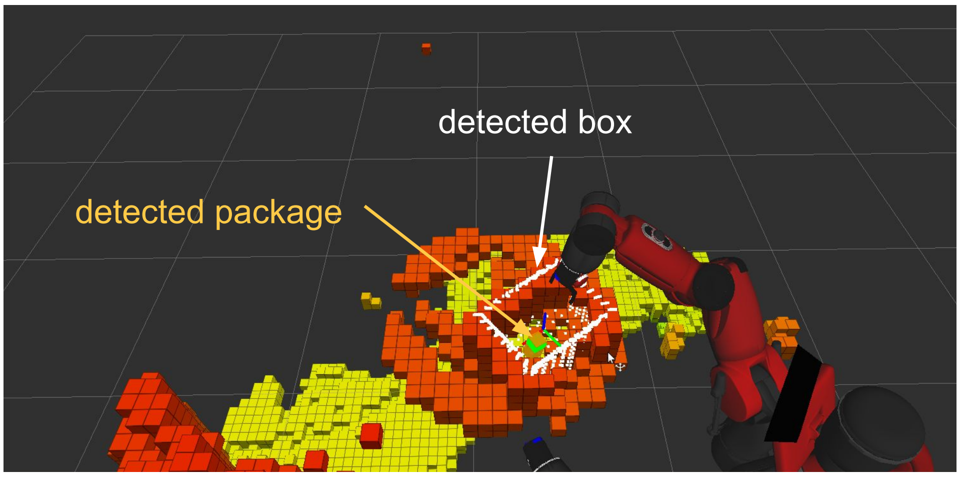


Figure 6.5: This figure shows the detected item published as cuboid object in orange. The former detected box is published as white point cloud. The collision scene are red colored voxel.

software Blender[3]. Therefore five cuboid objects, which represent the box borders and the box bottom, are arranged. This model is shown in Figure 6.7. Because the box has obviously a smooth surface, a surface subdivision is performed to get more points on the surface. Then the box can be stored as point cloud (*.ply) file. PCL can load these files. This box model is now used as template for the box detection.
The action returns the detected box pose in the ROS action result. The detected box can be seen as white point cloud in Figure 6.5. The white points represent the points which were exported by Blender.
The general *detectBox* routine is presented in Algorithm 1. The *preprocessing* routine is shown in Algorithm 2. This preprocessing routine makes use of the *pcl::PassThrough* function to crop the point cloud around the approximate pose, which is passed in via an argument. The *pcl::EuclideanClusterExtraction* functionality returns possible pointcloud candidates. These pointclouds are used by the *coarseFitting* presented in Algorith 3. It uses the *pcl::FPFHEstimation* for estimating the features and the *pcl::SampleConsensusInitialAlignment* function to align it. The *fineFitting* performs the final alignment in Algorithm 5 using the *pcl::IterativeClosestPoint* functionality. Finally it is verified that

[3]see `https://www.blender.org/`

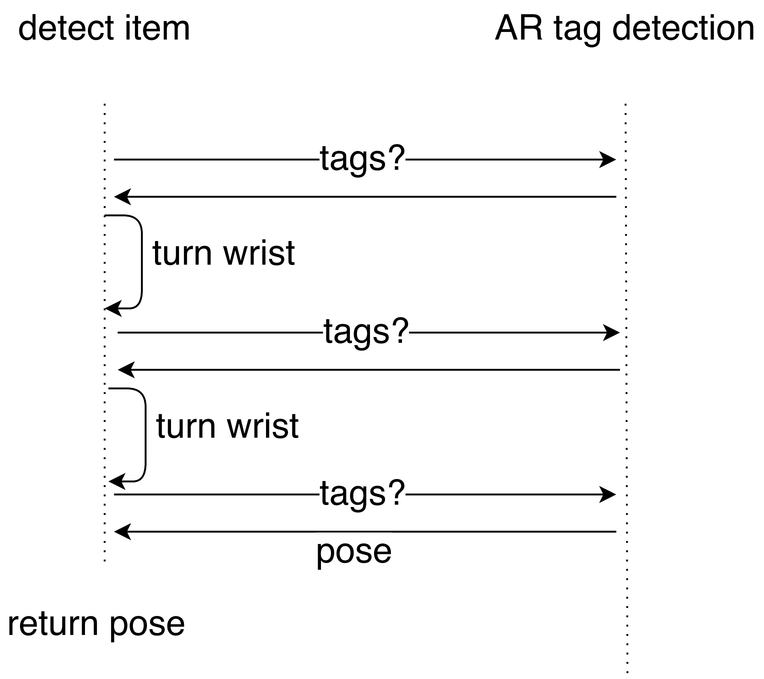


Figure 6.6: This figure shows an example sequence of the detect item nodel

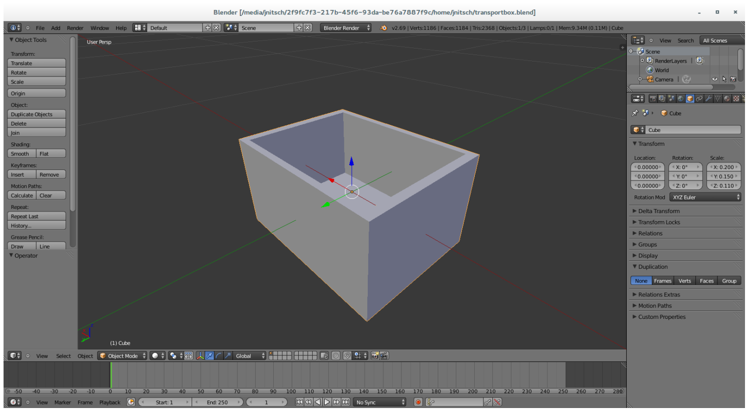

Blender [/media/jnitsch/2f9fc7f3-217b-45f6-93da-be76a7887f9c/home/jnitsch/transportbox.blend]
User Persp
(1) Cube

Figure 6.7: Here the transport box model in Blender is visualized.

z-direction of the pose points in the same direction as the z-direction of the global coordinate system. If it would not point in the same direction, that means that the box would have a wired orientation, but we know that it should stand on the tray. So either the detection failed, or the box does not stand on the tray. In this case the failure is returned, the box pose is empty. This check is computed the following way. First a translation $\mathbf{t\_B_z\_B}$ in z-direction of the box (w.r.t the box frame) is performed.

$$\mathbf{t\_B_z\_B} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \tag{6.1}$$

Because there is no rotation the transformation $T\_B_z\_B$ is given with:

$$\mathbf{T\_B_z\_B} = \begin{pmatrix} \mathbf{I} & \mathbf{t\_B_z\_B} \\ \mathbf{0} & 1 \end{pmatrix} \tag{6.2}$$

So the resulting pose B_z with respect to the origin O is given with:

$$\mathbf{T\_B_z\_O} = \mathbf{T\_B_z\_B} \cdot \mathbf{T\_B\_O} \tag{6.3}$$

We now take a closer look at the translation $\mathbf{t_{zB}}$ from the box B to B_z with respect to the origin:

$$\mathbf{t_{zB}} = \mathbf{t\_B_z\_O} - \mathbf{t\_B\_O} \tag{6.4}$$

This vector should ideally be $\mathbf{t_{zO}}$:

$$\mathbf{t_{zO}} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \tag{6.5}$$

Due to the special shape of the vectors it is tested if their dot product is between $1 - \epsilon$ and $1 + \epsilon$. Then our check evaluates with true otherwise it is not valid.

$$valid = \begin{cases} true, & \text{if } 1 - \epsilon < \mathbf{t_{zB}} \cdot \mathbf{t_{zO}} < 1 + \epsilon \\ false, & \text{otherwise} \end{cases} \tag{6.6}$$

The poses are schematically visualized in Figure 6.8. Please note that this verification, only checks one axis of the box pose. Because we cannot make a statement about the others. But the one tested is already a good indicator for a false detection.

Algorithm 1: detectBoxPose

```
   Data: cloud_in, template, approximate_pose
   Result: box_pose
 1 box_pose ⟵ emptyPose;
 2 clusters ⟵ preprocessing(cloud_in, approximate_pose);
 3 i ⟵ 0, score_best ⟵ 0, pose_best;
 4 while i <#ATTEMPTS do
 5     p_coarse, cloud ⟵ coarseFitting(clusters, template);
 6     p_fine, score ⟵ fineFitting(cloud, p_coarse, template) ;
 7     if score >score_best then
 8         pose_best ⟵ p_fine;
 9         score_best ⟵ score;
10     end
11     i ⟵ i + 1;
12 end
13 origin ⟵ globalcoordinatesystem ;
14 valid ⟵ checkPose(pose_best, origin);
15 if valid then
16     box_pose ⟵ pose_best;
17 end
```

Algorithm 2: preprocessing

```
   Data: cloud_in, approximate_pose
   Result: clusters
 1 cloud ⟵ cutAroundApproximiatePose(approximate_pose);
 2 clusters ⟵ euclideanClustering(cloud);
```

6.3.1.5 Head Node

This node implements two functionalities namely the *Scan Environment* and the *Link Follower* functionality.
The first functionality ensures that the whole collision scene is scanned before any other detections or arm movements are performed. Therefore this functionality is invoked immediately after the system start. It sends

Algorithm 3: coarseFitting

```
   Data: clusters, template
   Result: p_coarse, cluster
 1 $fpfh\_template \longleftarrow computeFPFH(template)$;
 2 $score\_best \longleftarrow 0, pose\_best, cluster\_best$;
 3 forall the cluster in clusters do
 4     $fpfh \longleftarrow computeFPFH(cluster)$;
 5     $pose, score \longleftarrow sacIA(fpfh, cluster, fpfh\_template, template)$;
 6     if score >score_best then
 7         $pose\_best \longleftarrow pose$;
 8         $score\_best \longleftarrow score$;
 9         $cluster\_best \longleftarrow cluster$;
10     end
11 end
12 $cloud \longleftarrow cloud\_best$;
13 $p\_coarse \longleftarrow pose\_best$;
```

Algorithm 4: fineFitting

```
   Data: cloud, p_coarse, template
   Result: p_fine, score
 1 $alignment \longleftarrow icp(cloud, p\_coarse, template)$ ;
 2 $score \longleftarrow alignment.getScore()$;
 3 $p\_fine \longleftarrow alignment.getFinalPose()$;
```

target positions to Baxter's head joint reaching from $\frac{-\pi}{2}\pi$ to $\frac{\pi}{2}$ in incremental steps, of 0.2 radiants, so that the Asus 3D camera, that is mounted on the head, can capture the whole workspace. After each incremental step, it stops for a short time to ensure that MoveIt! Planning Scene Monitor captures the new received point cloud. The collsion scene is represented as Octomap[4]. Such a collision scene is visualized in Figure 6.9.

The *Link Follower* functionality let the head follow any tf link so that the

[4]see `http://octomap.github.io/`

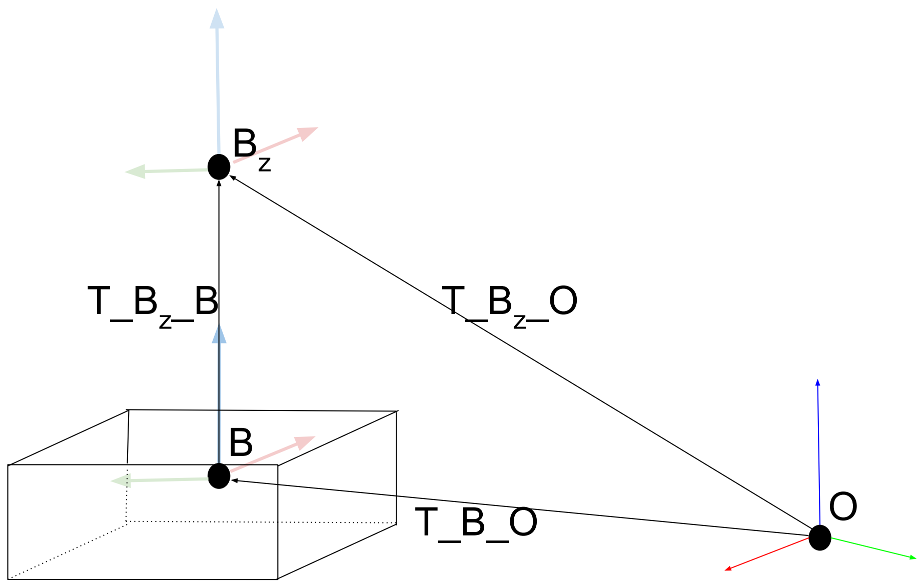


Figure 6.8: This figure shows the poses of the box and the pose translated in z-direction. The poses are needed to verify the orientation of the detected box.

Algorithm 5: check

```
   Data: p_best, origin
   Result: valid
1  $T\_B_z\_B.rotation \longleftarrow IdentityMatrix;$
2  $T\_B_z\_B.translation \longleftarrow Vec3(0,0,1);$
3  $T\_B_z\_O \longleftarrow T\_B_z\_B \cdot T\_B\_O;$
4  $t_{zB} \longleftarrow T\_B_z\_O.translation - T\_B\_O.translation;$
5  $t_{zO} \longleftarrow Vec3(0,0,1);$
6  if $(t_{zB} \cdot t_{zO} > 1\text{-}\epsilon)$ AND $(t_{zB} \cdot t_{zO} < 1\text{+}\epsilon)$ then
7      $valid \longleftarrow true;$
8  else
9      $valid \longleftarrow false;$
10 end
```

Asus 3D camera can observe this link continuously. This functionality is used to observe the grasping arm with the Asus 3D camera, while the arm

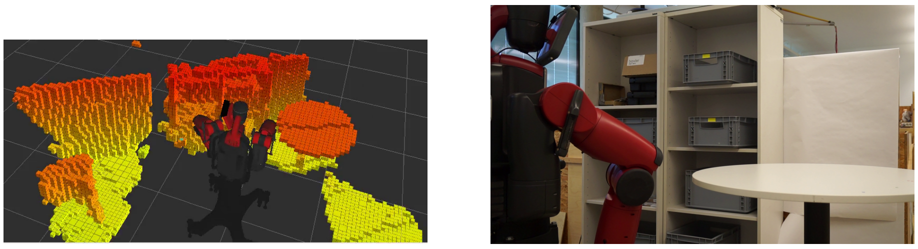

(a)Baxter before pushing the box into the level.
(b)Baxter after pushing the box into the level.

Figure 6.9: The left figure shows the sensed collision scene by the Asus 3D camera. The Planning Scene Monitor in MoveIt! keeps track of the environment. It is stored as Octomap. The right figure shows a picture taken of the real collision scene.

is moving or manipulating objects. This offers the opportunity to detect, if any object is moving into the workspace while manipulating objects. For example, the trajectory execution can be aborted, if a person walks into the workspace of Baxter. This node therefore uses the position of the grasping endeffector published as tf and calculates the corresponding head angle and publishes this angle as *HeadPanCommand*.

6.3.2 Arm Movements Module

The following nodes handle the arm movements and make use of the MoveIt! interface.

6.3.2.1 Move To Pose Node

The action provides a basic functionality for the system. It receives a pose the grasping arm should move to. Therefore it utilizes the MoveIt! interface to plan and execute a collision free trajectory. For planning the trajectory the LBKPIECE algorithm of the OMPL is used. The planned trajectory is sent to the JTAS, which handles the execution. If any errors occur, it is reported in the result of the action.

6.3.2.2 Move Over Box Node

This node implements the skill primitive described in Section 5.4.7.2. It receives the box pose as input and calculates the inspection poses as described in Algorithm 6. This algorithm receives the box pose BP_G, the box dimensions (h_b, d_b, w_b) and the distance between the single inspection poses $dist$. It calculates the first inspection pose IP_1 and last inspection pose IP_end as described in Section 5.4.7.2. The algorithm calculates new inspection poses as long as their distance to IP_1 is not longer as the distance from IP_{end} to the starting position. The orientation of each inspection pose is computed as described in Section 5.4.7.2.
This primitive also utilizes the MoveIt! framework to plan and execute a trajectory for the inspection poses. Again the LBKPIECE planner is utilized for planning and the JTAS handles the execution.

6.3.2.3 Move To Level Node

Within this node the movement part of the detect handle skill primitive (see Section 5.4.1.1) is implemented. It receives a semantic level name (e.g "Level_0_1"), looks up this name from the ROS parameter server, to which pose in the global coordinate system it belongs to, and moves the arm there, also using the MoveIt! framework. Like before the LBKPIECE plans the trajectory and the JTAS executes it.

6.3.2.4 Move To Support Pose Node

Here the calculations are implemented described in Section 5.4.3. Like in the previous described nodes, the LBKPIECE planner is used and the JTAS handles the execution.

6.3.3 Grasping Module

The following described functionalities cope with grasping objects.

Algorithm 6: cacluateInspectionPoses

```
Data: $BP_G, h_b, w_b, d_b, dist$
Result: inspection_poses
1  $inspection\_poses \longleftarrow []$;
2  $IP_1 \longleftarrow calculateFirstInspectionPose(BP_G, h_b)$;
3  $IP_{end} \longleftarrow calculateEndInspectionPose(BP_G, w_b, h_b, d_b)$;
4  $max_l \longleftarrow (IP_{end} - IP_1).length()$;
5  $dir \longleftarrow (IP_{end} - IP_1).norm()$;
6  $i \longleftarrow 1$;
7  while $true$ do
8      $vec \longleftarrow dir \cdot i \cdot dist$ if $vec.length() < max_l$ then
9          $IP \longleftarrow newinspectionpose$;
10         $IP.position \longleftarrow IP_1 + vec$;
11         $IP.orientation \longleftarrow calculateOrentation(IP.position, box_pose)$;
12         $inspection\_poses \longleftarrow inspection\_poses.pushBack(IP)$;
13         $i \longleftarrow i + 1$;
14     else
15         $inspection\_poses \longleftarrow inspection\_poses.pushBack(IP_{end})$;
16         $break$ ;
17     end
18 end
```

6.3.3.1 Grasp Handle Node

Here the functionalities of the primitive described in Section 5.4.2 are implemented. The visual servoing makes use of the functionalities implemented in the detect handle node (see Section 6.3.1.1). Therefore it just calls the ROS action to receive the measured poses. The repositioning is performed following the servoing strategy from Section 5.4.2. For a better understanding of the grasp handle primitive's routine and its interaction to the perception module as well as to the gripper, a sequence diagram is visualized in Figure 6.10. First it requests the poses P_G,H_G,L_G and R_G from the the detect handle module. It then evaluates this poses. The arm is not on at the desired pre-grasp pose. It moves the arm correspondingly. Meanwhile the detection module is sensing and updating the poses. The grasp handle node again

requests the poses and evaluates it. Because it still has not reached P_G it moves the arm again. The poses are requested and evaluated for the third time now. This time the arm reached P_G. It moves the arm to grasp the handle. Now the validation from the gripper module is requested. it returns a valid grasped. The primitive returns the success.
When performing the local recovery behavior it calls the actions implemented in the Move To Pose Node (see Section 5.4.3). As soon as the box is grasped securely, the box is *attached* to the grasping endeffector. This means that MoveIt! stores the information, that the box is now held by this arm and moves this collision object whenever the arm is moved. This is especially useful when the box is manipulated. So that the sensed point cloud of the box does not interfere as collision object with any planned trajectories of the arms.

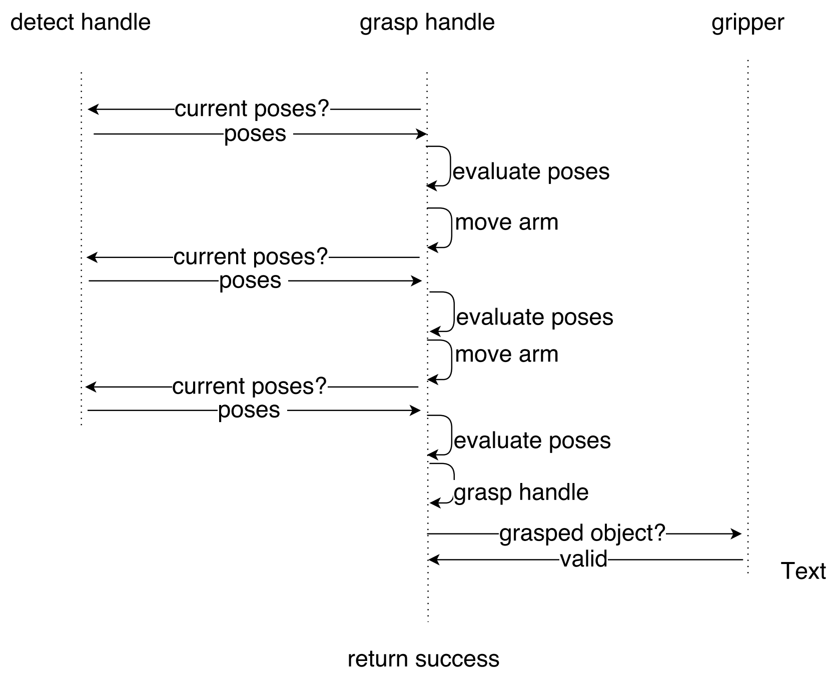


Figure 6.10: This figure shows an example sequence of the grasp handle node

6.3.3.2 GraspIt!

6.3.3.2.1 GraspIt! ROS msgs These messages define the ROS services which can be called to interact with GraspIt! in ROS. For our project we need two services: *LoadLocalGripper.srv* and *GenerateGrasps.srv*. As the name already suggests the *LoadLocalGripper.srv* loads a local gripper. A local gripper is stored in the GraspIt! folders and not loaded from any database. The service is defined in Listing 6.2. The gripper_name consists of the folder and the name of the loaded model (e.g: BaxterHand/baxter_hand.xml) and needs to be passed via this service. The service returns if the gripper can be loaded or if it fails.

Listing 6.2: *LoadLocalGripper.srv*

```
std_msgs/String gripper_name
---
int32 LOAD_SUCCESS = 0
int32 LOAD_FAILURE = 1
int32 result
```

The *GenerateGrasps.srv* service is defined in Listing 6.3. Please note that float32[][] datatypes are not a valid datatype in ROS. Instead *std_msgs/Float32MultiArray.msg* are used. We use the notation float32[][] for a clarification reasons. The model_name is the name of the object that should be grasped and the model_pose the corresponding pose. The flag reject_fingertip_collision indicates if fingertips are allowed to be in collision with the object to be grasped. This can be useful if the fingertips are made of a soft materials, which deform when touching an object. Because Baxter's fingertip are rather inflexible, we set that flag to false. Although the name of the request_tabletop flag is misleading, this flag indicates if an collision object should be load with the name passed by the tabletop_file_name. The service returns about any errors in the result variable. If the service succeeded it returns the grasping poses. One grasping pose contains of:

- bool hand_object_collision
- geometry_msgs/Pose grasp_pose
- float32[] grasp_joint_angle
- float32 grasp_energy
- float64 gripper_tabletop_clearance

- float64 gripper_object_distance

If the flag hand_object_collision is false then the hand is in collision with a collision object and the grasp should be ignored. If the flag is true, the grasp is valid. The grasp_pose contains the position and orientation of the wrist with respect to the object which should be grasped. The grasp_joint_angle holds the joint angle of the gripper. In the case of the parallel jaw gripper we use this is a single value, because it has only one degree of freedom. A quality measure is returned with the grasp_energy variable. The gripper_tabletop_clearance holds the distance to the next collision object and the gripper_object_distance holds the distance to the object which should be grasped. This is one grasp. Hopefully more then one grasp could be planned. And all the grasping informations are stored one after each other in the corresponding vectors. So the i^{th} grasp consists of following values: hand_object_collision[i], grasp_joint_angle[i], grasp_energy[i], gripper_tabletop_clearance[i] and gripper_object_distance[i].

Listing 6.3: *GenerateGrasps.srv*

```
std_msgs/String model_name
geometry_msgs/Pose model_pose
bool reject_fingertip_collision
bool request_tabletop
string tabletop_file_name
---
int32 GENERATE_SUCCESS = 0
int32 GENERATE_FAILURE = 1
int32 result

bool[] hand_object_collision
geometry_msgs/Pose[] grasp_pose
float32[][] grasp_joint_angle
float32[] grasp_energy
float64[] gripper_tabletop_clearance
float64[] gripper_object_distance
```

6.3.3.2.2 GraspIt! ROS package The GraspIt! framework itself is wrapped into a ROS package. This package does not include any source or header files, but a *CMakeLists.txt* and *package.xml* file only. During building this package, it invokes the external build command for the GraspIt! framework, which is a QT4 project. Further it copies the compiled executables to the build directory of the current ROS workspace. So the library *graspit* can be handled as a ROS library. Other packages can depend on it and the ROS build chain handles the linking.

6.3.3.2.3 GrapIt! ROS wrapper package This node provides a ROS wrapper for the GraspIt! library. It implements all the service servers for the services defined in the GraspIt! ROS messages package. It calls the appropriate GrapIt! functions provided by the *graspit* library. So if the *LoadLocalGripper.srv* service is called via ROS the *loadLocalGripperCB* callback is executed. It loads the gripper model using the function *loadGripper*. This calls functions from the *graspit* library to load the gripper model. In the case of the *GenerateGrasps.srv* service the *generateGraspsCB* call back is invoked. It first loads the collision object *loadCollisionObject* and the object to be grasped (*loadObject*). Then it invokes the Eigengrasp planner. The *invokeEigenGraspPlanner* initalizes and starts the planner. When it is finished, the planned grasps are extracted. The results are returned via the ROS service interface. The ROS interfaces of this node are used by the *Pick and Place Node*. The relation between these packages is shown in Figure 6.11.

6.3.3.3 Pick and Place Node

This node implements two actions: *Grasp Item* and *Place Item* (see description in Section 5.4.8).
The *Grasp Item* action receives the box and the item pose. Because the collision object defines the origin in the later used GraspIt! ROS wrapper, the item pose is first transformed into the box coordinate system. Then the services described in Section 6.3.3.2 are used to load the box as collision object, the item and the gripper. The grasps are planned using the *GenerateGrasps.srv*. The proposed grasping poses are returned via the ROS service. The generated grasps are returned with respect to the object pose. So they are first transformed to global coordinates and then packed into a

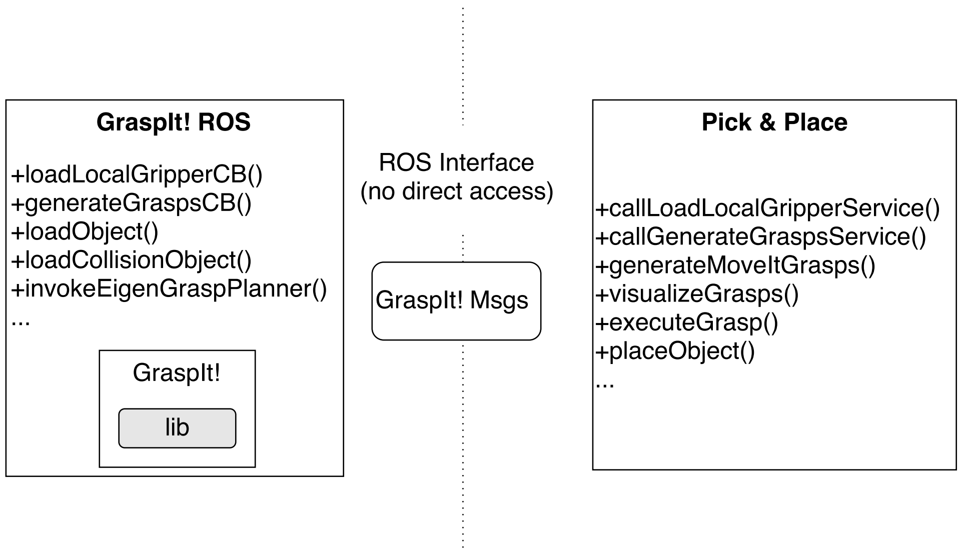


Figure 6.11: This figure shows the different nodes which use either GraspIt! libraries directly or access it via ROS interfaces

moveit_msgs::Grasp message (see Listing 6.4). The id sets the name of the current grasp. The pre_grasp_posture and the grasp_posture represent the joint angles of the gripper in the pre-grasp pose and the grasp pose. In our system these values are the same for all grasps. In the pre_grasp_posture the gripper should be open as much as possible and in the grasp_posture the gripper is closed. If a more sophisticated gripper is used, the grasp_joint_angle from the graspit message should be set. The grasp_pose is set to the pose retrieved from the service, as well as the grasp_quality. The pre_grasp_approach defines how the gripper should approach towards the object. Because in our setup we have a table top grasp we set this for all grasps to the direction **pre_grasp_approach_direction**:

$$\textbf{pre\_grasp\_approach\_direction} = \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix} \tag{6.7}$$

This ensures that the gripper performs a table top grasp. The **post_grasp_retreat** defines the direction of the gripper after the grasp, which is in our case the negative direction of the pre-grasp approach direction:

$$\textbf{post\_grasp\_retreat} = -\textbf{pre\_grasp\_approach\_direction} \tag{6.8}$$

So the arm is basically lifted after the grasp. We do not define the *post_place_retreat*. This would define the place movement of the gripper, which is immediately opened after the movement. We decided to separate this behavior to the *Place Item* action. The max_contact_force is set to a positive number so that the gripper closes, but we do not mind to much about a too strong grasp, because our item cannot break. We add our item collision object, visualized in Figure 6.12 as orange box, to the allowed_touch_objects. In the top left corner of the Figure 6.12 the grasp is visualized again. Motion trajectories are planned now in a way, that the gripper is allowed to touch the item. All the different grasps are stored in a vector (*std::vector<moveit_msgs::Grasp>*). The generated grasps are then also visualized in RViz as markers (see Figure 6.12). For the visualization of the grasps the *moveit_visual_tools* package is used. The grasps can be passed to MoveIt! via an interface. It then plans and executes the grasps. So it provides basically an interface to hand over different possible grasps and to perform a "pick", when a valid motion plan for one of those grasps can be found.

The *Place Item* action just moves the arm to the delivery place using MoveIt! and opens the gripper directly.
For both described actions the LBKPIECE planner is invoked and JTAS handles the execution.

Listing 6.4: *moveit_msgs/Grasp.msg*

```
string id
trajectory_msgs/JointTrajectory pre_grasp_posture
trajectory_msgs/JointTrajectory grasp_posture
geometry_msgs/PoseStamped grasp_pose
float64 grasp_quality
moveit_msgs/GripperTranslation pre_grasp_approach
moveit_msgs/GripperTranslation post_grasp_retreat
moveit_msgs/GripperTranslation post_place_retreat
float32 max_contact_force
string[] allowed_touch_objects
```

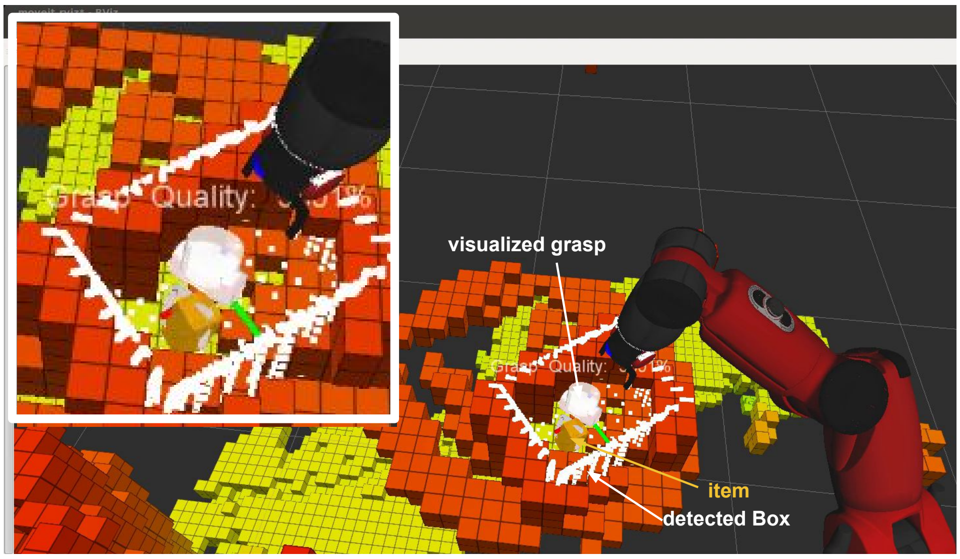


Figure 6.12: This figure shows one visualized grasp in RViz as marker. In the top left corner the grasp itself is visualized in more detail (zoomed in)

6.3.4 Manipulation Module

The following nodes which are described, all implement functionalities to manipulate the transport box.

6.3.4.1 Pull Box Node

This node implements the concept described in Section 5.4.4. In order to ensure that the wrist does not turn during the movement, this joint is constraint in its orientation. MoveIt! already provides a planner that handles this constraint: *ComputeCartesianPath*. This interface expects the waypoints the endeffector should follow without turning the wrist. It also returns an estimation, if the path can follow the waypoints or not. So the pull box action uses this interface to plan the motion and executes it using the standard MoveIt! interface (sending the trajectory to the JTAS). If no plan can be found it returns with an error.

6.3.4.2 Move Box Node

Within this node the concept of Section 5.4.5 is implemented. Therefore MoveIt! interfaces for both arms are initialized. Again the *ComputeCartesian-Path* interface is used. Because none of the arms is allowed to turn the wrist, otherwise the box is lost during manipulation. Like in the other movements and manipulation nodes the planned trajectory is sent to the JTAS, which handles the execution.

6.3.4.3 Deliver Box On Level Node

This node implements the action for delivering the box on the level as described in Section 5.4.10. Therefore it also implements a MoveIt! interface. When the box is finally delivered on the level, the box is *detached* from the endeffector. Now MoveIt! registers, that the box is no longer hold by the gripper any more and handled as normal collision object.

6.3.4.4 Deliver Box On Tray Node

In comparison to the node before, this node implements the action to deliver the box on the tray (see Section 5.4.6). When the box is delivered on the tray it is also *detached* from the gripper.

7 Evaluation

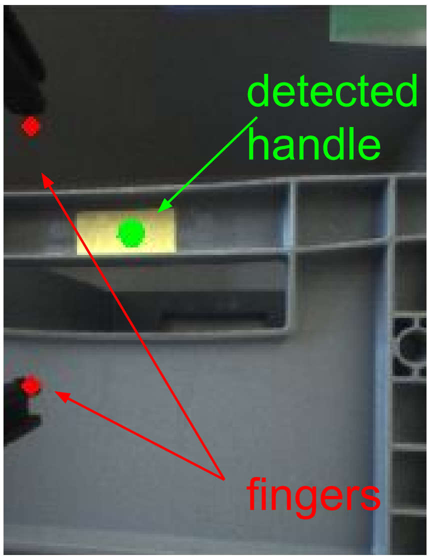
Within this chapter the different parts of the system are evaluated. Each skill is decomposed in its skill primitives and the primitives are evaluated separately. In order to see the robustness of the different subtasks the focus of the evaluation is on the single primitives. In addition three parts of the system are picked out and evaluated in detail. The box detection is compared to an ICP algorithm and to the FPFH with SAC-IA alignment, both implemented in the PCL. The next detailed evaluation concerns planning and execution of grasps for different positions of the item inside the transport box. The last part shows grasps, planned by the Eigengrasp planner, for the Baxter gripper and the Schunk WSG 50 gripper in the GraspIt! framework. Besides the presentation of the evaluation results particular aspects of the evaluation will be discussed in detail.

The skill primitive evaluation has the same structure for all primitives. First the environmental setup - including the robot state - is described the primitive starts with. Then the expected outcome is described. A skill primitive's execution is called successful if the expected outcome is achieved. Otherwise the execution is called a failure. The failure cases are distinguished into two groups: failures which are detected by the system and failures which are not detected. The detected failure rate is a measure for the robustness of the system.

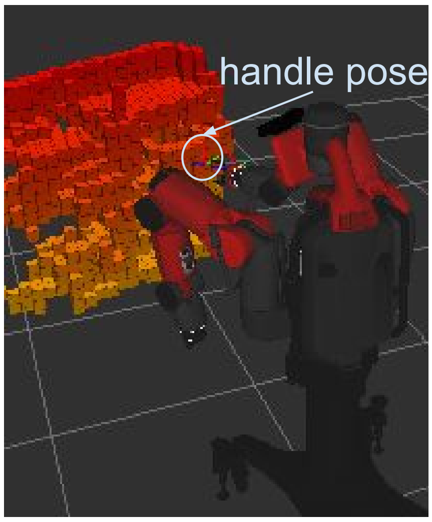
7.1 Skill MoveBoxFromLevelToTray

As described in Section 5.3 this skill is composed of skill primitives. These primitives are tested separately. First the primitive *detectHandle* is evaluated followed by the *graspHandle* primitive. Then the *moveToSupportPose* primitive is tested, afterwards the *pullBox* and *moveBox* are evaluated. Finally the *deliverOnTray* primitive is evaluated. For all primitives a start state is defined and the expected outcome. The success of a primitive execution is verified by the person who runs the tests. For perception primitives the detected poses and objects were visualized in RViz and verified as success or failure by the user. The same holds for the manipulative primitives. Within this test setup there is no opportunity to externally verify the robot's ground truth arm poses, or the poses of the manipulated objects. So there is a person in the testing loop, which verifies the successful outcome of the primitive.

7.1.1 detectHandle



(a)Handle detection in the camera image.



(b)Marked handle in the collision scene.

Figure 7.1: This figure shows the handle in the camera image as well as in collision scene. In the left figure the detected handle pose is marked with a green dot by the perception module, as well as the red diamonds for the finger poses. The text and the arrows were added to highlight the perception.

Setup The environment is already scanned and the collision scene loaded. Baxter is standing in front of the shelf. Both arms are in a neutral pose. Their endeffectors are pointing towards the floor. The box is on the level which is delivered via parameter.

Expected Outcome The grasping arm is moved in front of the level. The primitive returns the *handle pose* with respect to the global coordinate system H_G, the *left finger pose* L_G, the *right finger pose* R_G as well as the *pre-grasp pose* P_G. The poses are then visualized in the sensed collision scene. The user verifies the outcome with success, if the arm is moved in front of the level and the poses are visualized at corresponding poses in the sensed collision scene. First the handle detection in the image is verified (visualized as green dot, see Figure 7.1a) and the handle pose is visualized in the scene (Figure 7.1b).

Results This primitive executes 79.3% times successfully of 58 trials. Of the 20.7% failure cases, 30% are correctly detected as error. This happens when the trajectory for an arm movement is planned, but not correctly executed by Baxter. In 70% of the failure cases the whole planning node (movegroup node) crashed and this failure is not detected. So the primitive needs to be aborted and the whole system restarted.

7.1.2 graspHandle

In Figure 7.2 Baxter is shown while it tries to grasp the handle.

Setup The environment is already scanned and the collision scene loaded. Baxter is standing in front of the shelf. The grasping hand is already in front of the level, where the box stands on.

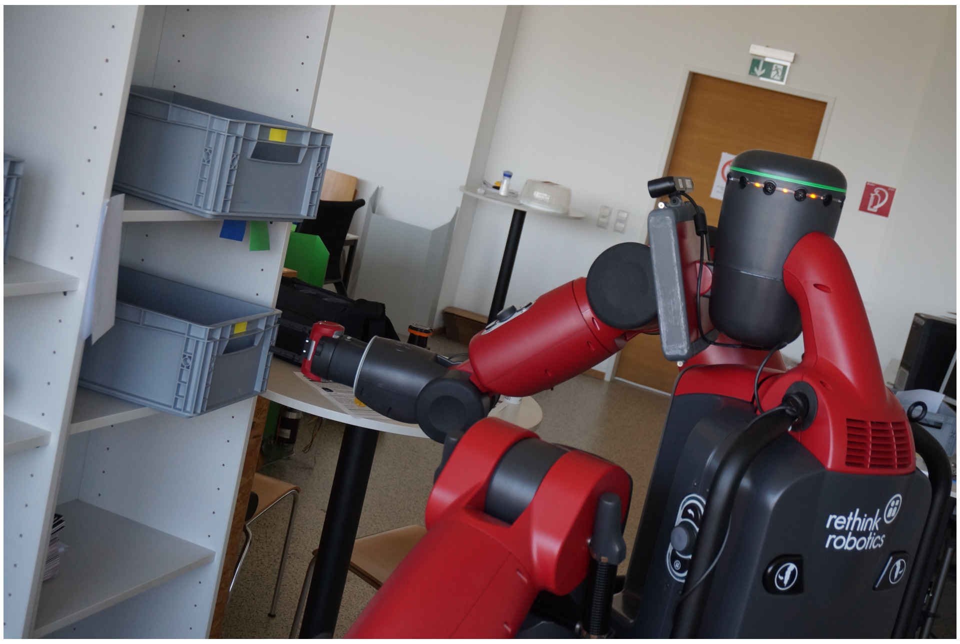


Figure 7.2: This figure shows Baxter while the grasping handle skill primitive.

Expected Outcome Baxter grasps the handle securely and reports about the success. The user verifies the secure grasps in pulling and pushing a little bit. If the box does not slip out of the gripper the user verifies the successful outcome.

Results This primitive executes 70% successfully out of 49 trials. 50% of the fails were correctly detected as failures and reported. Here the biggest problem was the inaccurate path execution. Due to this inaccuracies sometimes not the handle but a different part of the box was grasped. This grasps were not stable but reported as success by the primitive, but were failure cases. The box slipped out of the gripper when the user touched the box.

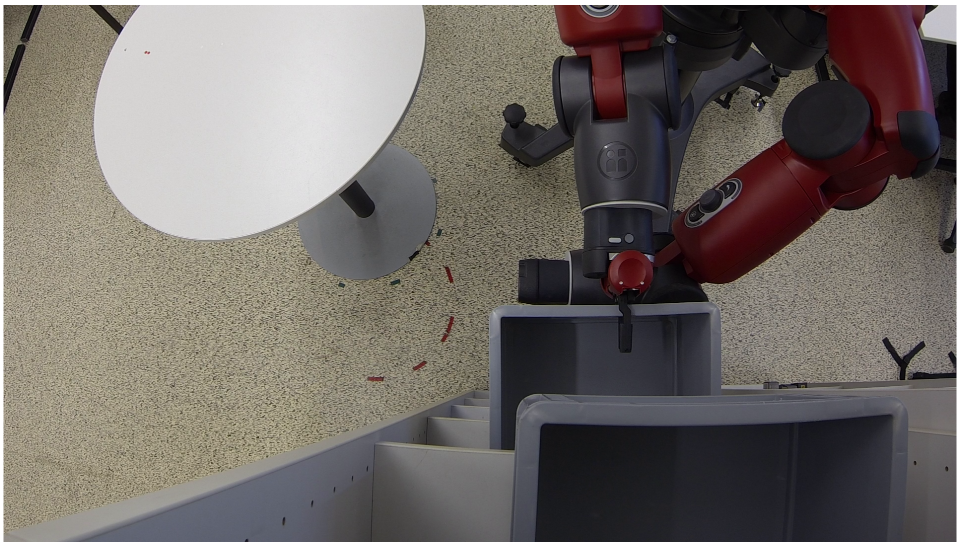


Figure 7.3: This figure shows the top view of the support pose. Within the next step Baxter pulls the box on the support arm.

7.1.3 moveToSupportPose

Setup The environment is already scanned and the collision scene loaded. Baxter is standing in front of the shelf and securely grasps the box.

Expected Outcome Baxter moves the support arm in support position, so that the box could be pulled out. After the movement the user tries to pull out the box of the level. If the box is supported by the support arm and not lost, the execution is called successful.

Results It successfully executes in 85% of 34 trials. During the failure cases it detected errors with 80%. These errors were mainly either that no trajectory could be planned, or that the arm got stuck in the rack due to a inaccurate execution of the trajectory.

7.1.4 pullBox

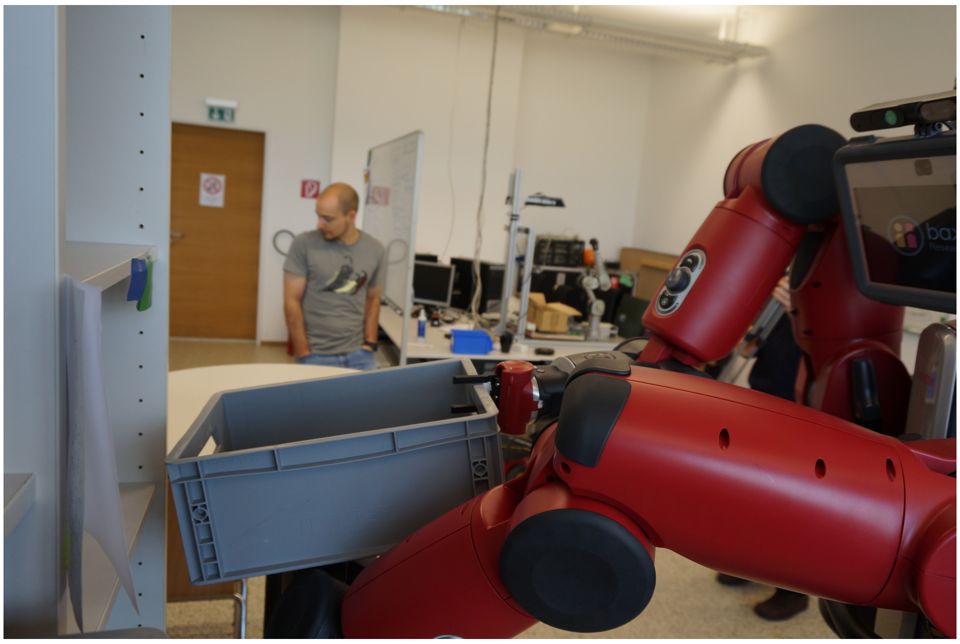


Figure 7.4: This figure shows Baxter pulling the box

Setup Baxter grasped the box securely and the second arm is in support pose.

Expected Outcome Box is pulled out of the level on the support arm. Baxter is now balancing the box on both hands in front of the level. The user again pushes and pulls the box a little bit. If the box is not lost, the execution of this primitive is called successful. If the box is already lost during this small movements, it is not grasped securely any more. This is considered being an error, as well as loosing the box during the arm movement. In Figure 7.4 Baxter is shown after the pull Box primitive was performed. It still securely grasps the box while supporting it with the second arm.

Results This primitive executed 82% of 28 cases without failure. During all the failure cases it detected the error. When an error occurred Baxter always lost the box and the box fell to the ground.

7.1.5 moveBox

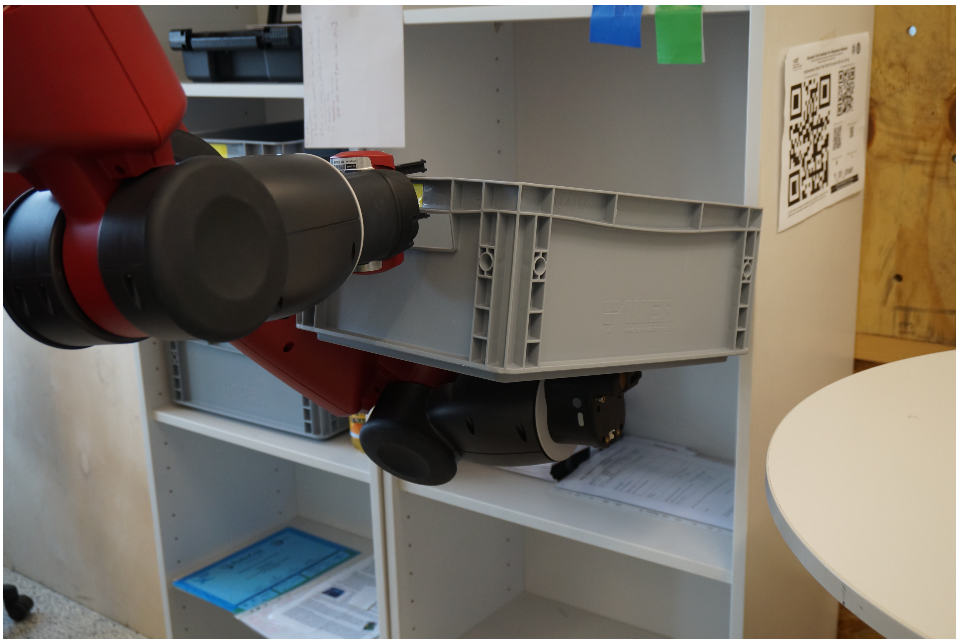


Figure 7.5: This figure shows Baxter while moving the box.

Setup Baxter is balancing the box in front of the level.

Expected Outcome Baxter moves the box towards the tray. It balances the box in front of the tray but does not deliver it on the tray. The execution is called successful if the box is balancing in front of the tray (about 5 centimeters verified by the user). If the box is lost during the movement this is called an error.

Results During the evaluation this primitive never failed (22 trails). This primitive performs that well, because auch single movement of the arms is relatively small and slow. So the trajectory execution seems to be more precisely in small and slow motions. So Baxter is very unlikely to loose the box. Furthermore in this setup the distance between the shelf and the tray is also quite short. In Figure 7.5 Baxter is moving the box towards the tray.

7.1.6 deliverOnTray

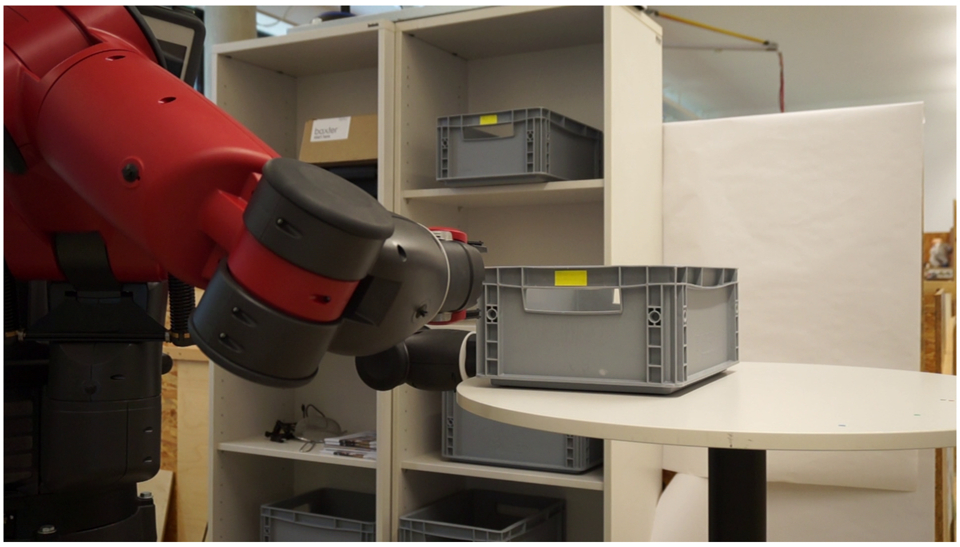


Figure 7.6: This figure shows Baxter after it delivered the box on the tray.

Setup Baxter is balancing the box in front of the tray.

Expected Outcome Baxter delivers the box on the tray. It is considered being a success if the box stands on the tray (not falling to the ground) and if the robot could grasp the item inside the box. Further it reports the *box release pose* BR_G were it releases the handle. The user verifies if the robot could grasp the item. Therefore it moves the grasping arm in the box and grasps the object using the compliant arm mode. Figure 7.6 shows the box's successful delivery on the tray.

Results This primitive executes successfully in 82% of 22 cases. In the failure cases, it either looses the box, or no possible grasp afterwards was be possible (verified by the user).

7.2 Skill graspItem

This skill is composited of three primitives: *detectItem*, *graspItem* and *deliverItem*. Again the primitives are tested separately. The user still has to verify the success of one primitive. Because parts of this skill are evaluated later on in detail, some parts are not discussed here. This includes the box detection, as well as a further analysis of grasping items.

7.2.1 detectItem

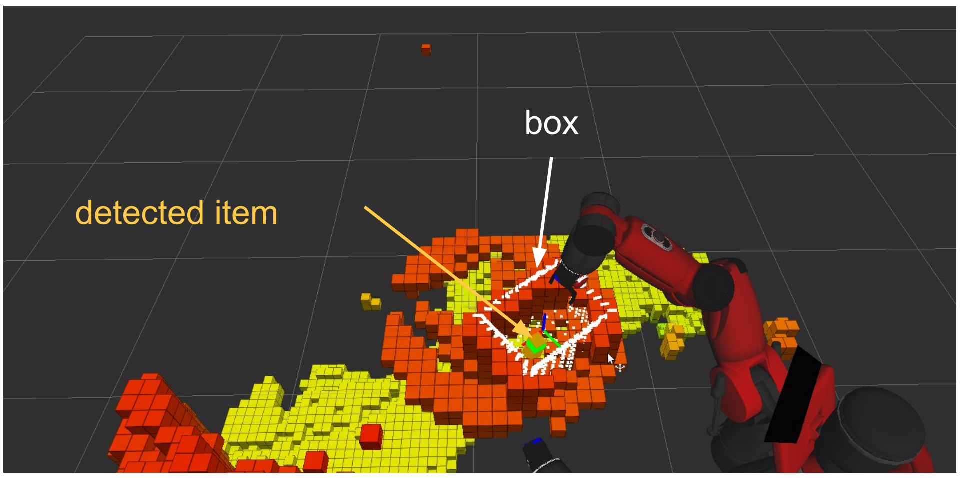


Figure 7.7: This figure shows the detected item and the box.

Please note that the box detection is tested separately and the results are stated in Section 7.4.

Setup The box is standing on the table. In this case the box pose is known in advance.

Expected Outcome Baxter moves with its grasping arm over the box and detects the item using the camera mounted on the endeffector. It reports box pose and item pose. If the skill primitive succeeds the box and the item pose is visualized in RViz (see Figure 7.7). If it is verified by the user, the primitive is counted as success.

Results During the evaluation Baxter was always able to move over the box and detect the item. Only in one case out of 22, where one finger disturbed the camera image, the item position was correct but not its orientation.

7.2.2 graspItem

One part of the grasp item analysis was tested during the test of this skill. More tests on grasping the item in different poses inside the box are shown in Section 7.5.

Setup The box and the item pose are given.

Expected Outcome Good grasps are calculated utilizing the Eigengrasp planner. Trajectories are planned to perform this grasps and Baxter executes one of those. If the primitive returns about the success, the user needs to verify if the item is really grasped.

Results Baxter was only able to grasp it securely two times out of 22. One big problem was, that no trajectory could be planned, because the collision scene was corrupted. It had some strange obstacles directly in front of Baxter. This obstacles especially appeared, when the arm moved in front of the 3D camera. Because this problem caused many error cases this evaluation is more or less worthless. Experiments on grasping object were repeated, where this changes in front of Baxter are ignored and the trajectories are simply executed. Nevertheless the second problem here was, even when the collision scene was correct detected and the trajectory could be planned, that the trajectory execution was too inaccurate to grasp the small item. The actual grasp pose was only a few centimeter (one to two cm) away from the planned one. So quite often the item was nearly grasped but slipped out of the gripper.

7.3 Skill moveBoxFromTrayToLevel

For this skill only the primitives *pullBox* and *deliverOnLevel* are evaluated. Because during testing the other primitives worked as well as in the *MoveBoxFromLevelToTray* skill. *pullBox* is evaluated again, because pulling the box on the tray behaves a little bit different than pulling on the level, due to different surfaces. The user verifies the successful outcome of the primitives.

7.3.1 pullBox

Setup The box is on the tray. Baxter securely grasps the handle. The support arm is in support position.

Expected Outcome Baxter pulls the box on the support hand. It balances the box in front of the tray. If the primitive returns success, the user verifies if the box is securely hold on the support arm in pushing and pulling the box a little bit. If the box is lost, then has not been grasped securely any more. This is counted as failure of the primitive.

Results This skill primitive worked out also 82% correctly. During the failure cases it was not able to pull it completely on the support arm. It would have been lost during the first next movements, so this cases were marked as failure.

7.3.2 deliverOnLevel

Setup Baxter is balancing the box in front of the level.

Expected Outcome The box is on the level and Baxter released the handle. This primitive is succeeds if the box is placed in the level and not falling to the ground. This is verified by the user.

Results Only 60% tests were successfully executed. The main problem was, after releasing the box handle and moving back as described in Section 5.4.10, the finger got caught on a small part of the handle. So Baxter pulled out the box from the level again and the box fell to the ground. The trajectories were planned correctly but the execution was to inaccurate.

7.4 Box Detection

Three different algorithms for detecting the correct box pose are evaluated: *ICP*[1], *FPFH* with *SAC-IA* alignment [1] and the coarse to fine approach presented in Section 5.4.7.1. The box was detected three times per pose on 15 different poses on the tray. Some of these poses are visualized in Figure 7.8. The results are plot in Figure 7.9 for the testing poses *Pose 1* to *Pose 7* and in Figure 7.10 for the testing poses *Pose 8* to *Pose 15*. Our coarse to fine approach detects the box robustly. But whereas the *ICP's* average computation time is about 10 seconds as well as the *FPFH* with *SAC-IA* alignment, our algorithm needs around 1.2 minutes, which is the drawback of this approach. For continuing primitives a stable box detection is a necessary

[1]Algorithm as it is provided in the PCL

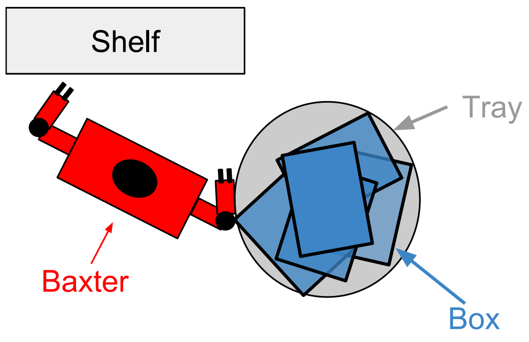


Figure 7.8: In this figure the schematic test setup for detecting the box is shown. Some of the different box poses are visualized.

prerequisite for a successful execution.
There are multiple reasons why the coarse to fine approach works better in this environment than the pure *FPFH* with *SAC-IA* alignment or the *ICP*. First due to occlusions and no 360°view around the box, only some parts of the box are captured by the Asus 3D camera and represented as point cloud. Further for the *FPFH* approach the surface of the box is simply too smooth and features are mainly detected on the edges. If the point cloud is preprocessed this approach gives a good guess of the pose but no exact pose. If the point cloud is not preprocessed and the *FPFH* with *SAC-IA* alignment is run on the raw cloud, there are too many similar shapes in the environment and this algorithm cannot find a good guess. The same holds for the *ICP*, it runs in a local minima if is performed on the raw cloud or on the preprocessed one. Some times the *ICP* is able to detect the pose (Pose 8 and Pose 15), maybe because the camera was able to capture a slightly larger point cloud of the object. But if *ICP* receives a good initial guess it aligns the box really well. Of course only three detections per pose shown in Figure 7.9 and Figure 7.10 are far too few to make any conclusions on a statistic, but they show at least that our approach is able to detect the box.

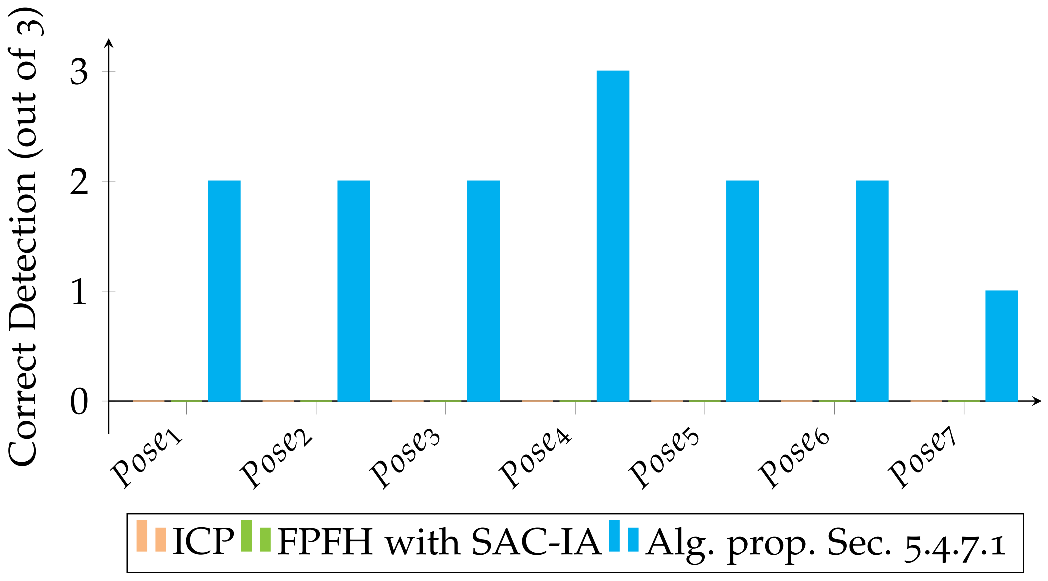


Figure 7.9: In this figure the evaluation for the different algorithms (*ICP*, *FPFH* with *SAC-IA* alignment, algorithm proposed in Section 5.4.7.1) is shown for testing poses *Pose 1* to *Pose 7*

7.5 Grasp Item

For this evaluation the box pose remained the same. The pose of the item inside the box was changed. For each pose the following is tested for five times:

1. Detect item
2. Plan Grasp
3. Execute Grasp

Within this experiment the item positions on the border of the box were tested. These positions are marked with a blue rectangle in Figure 7.11. The lower middle pose was not tested, because it was outside of Baxter's arm workspace. So it could not reach this positions. We tested if Baxter could at least move to the positions using the compliant mode.

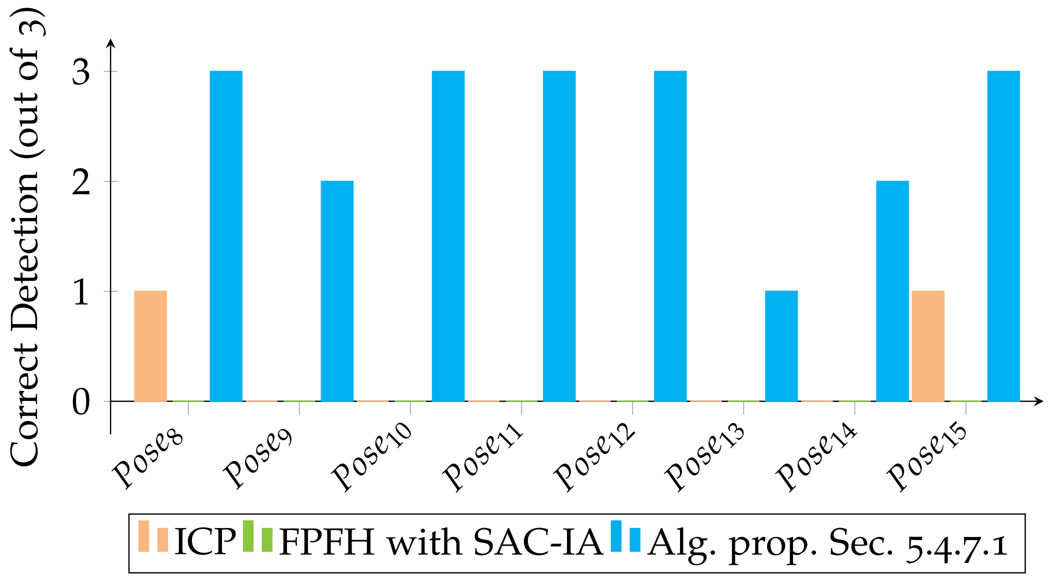


Figure 7.10: In this figure the evaluation for the different algorithms (*ICP*, *FPFH* with *SAC-IA* alignment, algorithm proposed in Section 5.4.7.1) is shown for testing poses *Pose 8* to *Pose 15*

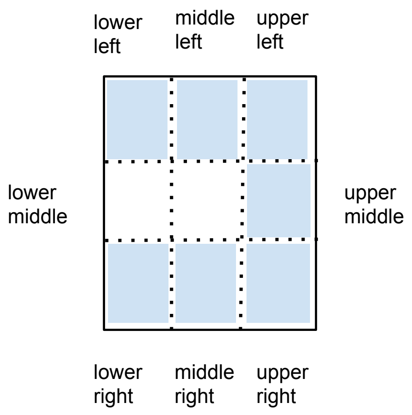


Figure 7.11: This figure shows different item positions which were tested.

7.5.1 Left Middle Item Pose

The schematic setup is shown in Figure 7.12. The results of the outcomes of the evaluated steps are shown in Table 7.1. In Experiment 2 the item was not detected, in all other experiments no valid trajectory could be planned. From this experiments it looks like that this was an unreachable pose for Baxter.

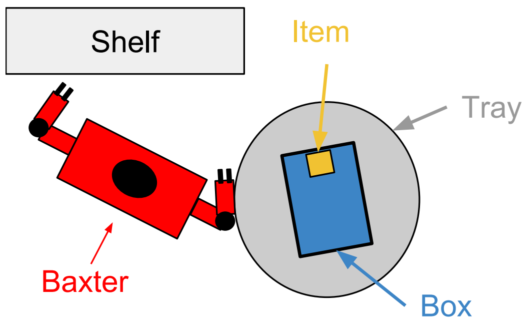


Figure 7.12: In this figure the schematic test setup for grasping the item at the left middle pose is shown. Baxter is marked as red rectangle. The shelf is represented as gray rectangle, and so the tray. The box is shown as blue rectangle. The item is marked as yellow square.

Table 7.1: Evaluation of grasping an item at the left middle pose inside the box.

Experiment Nr.	detect item	plan grasps	executes grasps
1	✓	✓	Failed
2	Failed	-	-
3	✓	✓	Failed
4	✓	✓	Failed
5	✓	✓	Failed

7.5.2 Right Middle Item Pose

The schematic setup is shown in Figure 7.13. The results of the outcomes of the evaluated steps are shown in Table 7.2. Whereas in Experiment 1 and Experiment 4 a motion plan was found and a grasp was executed unsuccessfully, in Experiment 2, Experiment 3 and Experiment 5 no trajectory could be planned.

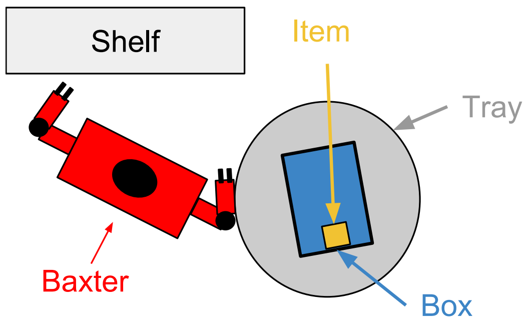


Figure 7.13: In this figure the schematic test setup for grasping the item at the right middle pose is shown. Baxter is marked as red rectangle. The shelf is represented as gray rectangle, and so the tray. The box is shown as blue rectangle. The item is marked as yellow square.

Table 7.2: Evaluation of grasping an item at the right middle pose inside the box.

Experiment Nr.	detect item	plan grasps	executes grasps
1	✓	✓	Failed
2	✓	✓	Failed
3	✓	✓	Failed
4	✓	✓	Failed
5	✓	✓	Failed

7.5.3 Upper Left Item Pose

The schematic setup is shown in Figure 7.14. The results of the outcomes of the evaluated steps are shown in Table 7.3. In Experiment 2 and Experiment 5 the item was successful grasped. In Experiment 3 no feasible trajectory could be planned. In Experiment 4 a motion plan was executed but the item slipped out of the gripper. In Experiment 1 no grasps could be planned. The item was detected but at a slightly wrong position. This lead to unfeasible grasps only.

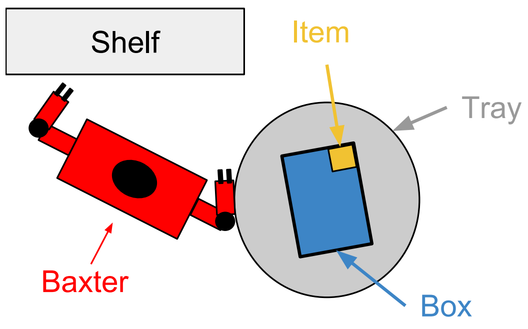


Figure 7.14: In this figure the schematic test setup for grasping the item at the upper left pose is shown. Baxter is marked as red rectangle. The shelf is represented as gray rectangle, and so the tray. The box is shown as blue rectangle. The item is marked as yellow square.

Table 7.3: Evaluation of grasping an item at the upper left pose inside the box.

Experiment Nr.	detect item	plan grasps	executes grasps
1	✓	Failed	-
2	✓	✓	✓
3	✓	✓	Failed
4	✓	✓	Failed
5	✓	✓	✓

7.5.4 Lower Left Item Pose

The schematic setup is shown in Figure 7.15. The results of the outcomes of the evaluated steps are shown in Table 7.4. In Experiment 1 and Experiment 5 no trajectory could be planned so that Baxter would have grasped the object. In Experiment 2 it grasped the item but it slipped out of the gripper. In Experiment 3 the item was not detected and in Experiment 4 the detection was slightly wrong, but that much that no feasible grasps could be calculated.

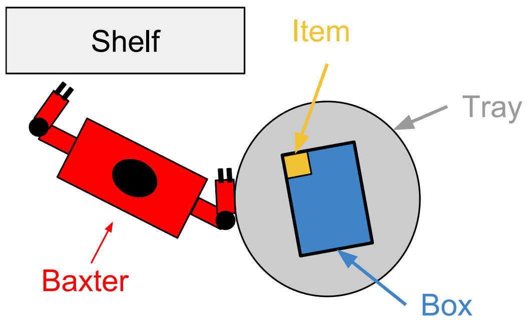


Figure 7.15: In this figure the schematic test setup for grasping the item at the lower left pose is shown. Baxter is marked as red rectangle. The shelf is represented as gray rectangle, and so the tray. The box is shown as blue rectangle. The item is marked as yellow square.

Table 7.4: Evaluation of grasping an item at the lower left pose inside the box.

Experiment Nr.	detect item	plan grasps	executes grasps
1	✓	✓	Failed
2	✓	✓	Failed
3	Failed	-	-
4	✓	Failed	-
5	✓	✓	Failed

7.5.5 Upper Right Item Pose

The schematic setup is shown in Figure 7.16. The results of the outcomes of the evaluated steps are shown in Table 7.5. Within this experimental setup the item was never detected, because it was either occluded by one finger or it was not visible in the camera image. So no grasping evaluation could be performed.

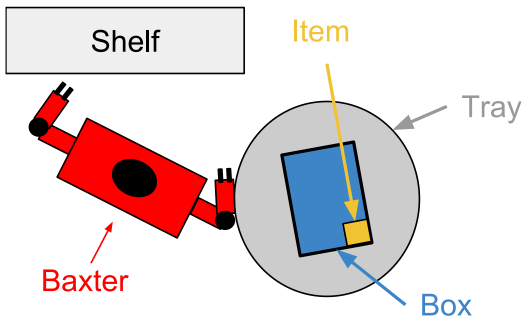


Figure 7.16: In this figure the schematic test setup for grasping the item at the upper right pose is shown. Baxter is marked as red rectangle. The shelf is represented as gray rectangle, and so the tray. The box is shown as blue rectangle. The item is marked as yellow square.

Table 7.5: Evaluation of grasping an item at the upper right pose inside the box.

Experiment Nr.	detect item	plan grasps	executes grasps
1	Failed	-	-
2	Failed	-	-
3	Failed	-	-
4	Failed	-	-
5	Failed	-	-

7.5.6 Lower Right Item Pose

The schematic setup is shown in Figure 7.17. The results of the outcomes of the evaluated steps are shown in Table 7.6. Within all these experiments the item was detected correctly and valid grasps were calculated. But in no experiment a valid motion plan could be found. This indicates that in this pose of the item simply cannot be reached by Baxter.

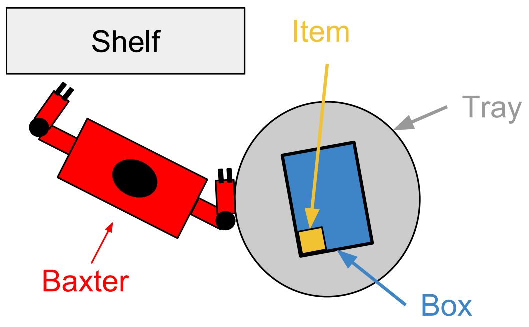


Figure 7.17: In this figure the schematic test setup for grasping the item at the lower right pose is shown. Baxter is marked as red rectangle. The shelf is represented as gray rectangle, and so the tray. The box is shown as blue rectangle. The item is marked as yellow square.

Table 7.6: Evaluation of grasping an item at the lower right pose inside the box.

Experiment Nr.	detect item	plan grasps	executes grasps
1	✓	✓	Failed
2	✓	✓	Failed
3	✓	✓	Failed
4	✓	✓	Failed
5	✓	✓	Failed

7.5.7 Upper Middle Item Pose

The schematic setup is shown in Figure 7.18. The results of the outcomes of the evaluated steps are shown in Table 7.7. Within all these experiments the item detected and grasps were planned correctly. Also motion plans were calculated. During execution the item always slipped out of the gripper.

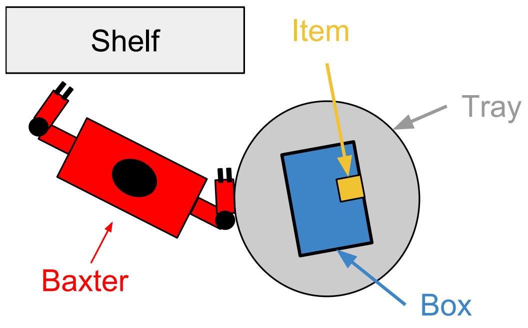


Figure 7.18: In this figure the schematic test setup for grasping the item at the upper middle pose is shown. Baxter is marked as red rectangle. The shelf is represented as gray rectangle, and so the tray. The box is shown as blue rectangle. The item is marked as yellow square.

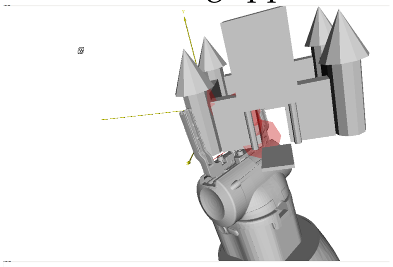
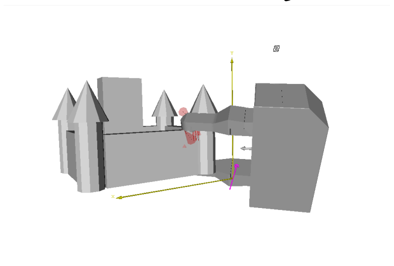
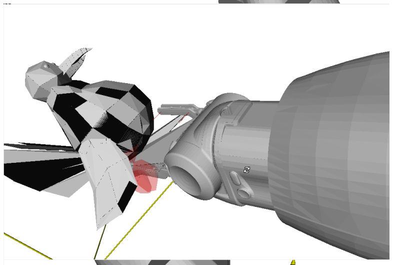
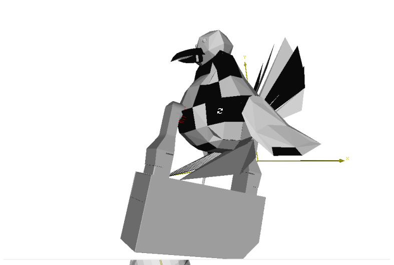
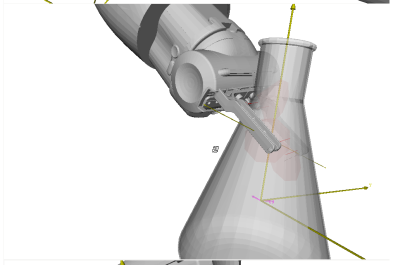
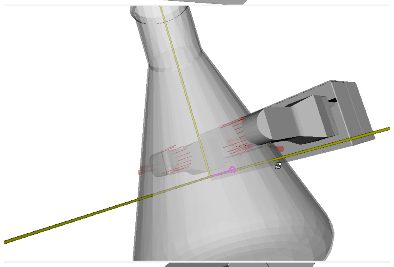
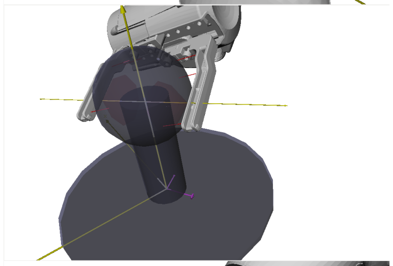
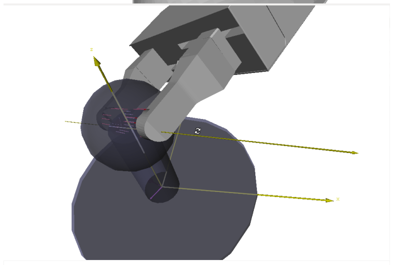
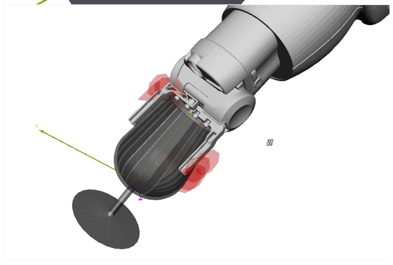
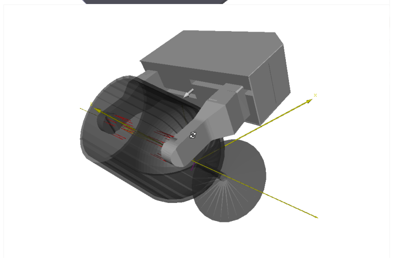
Table 7.7: Evaluation of grasping an item at the upper middle pose inside the box.

Experiment Nr.	detect item	plan grasps	executes grasps
1	✓	✓	Failed
2	✓	✓	Failed
3	✓	✓	Failed
4	✓	✓	Failed
5	✓	✓	Failed

7.6 Parallel Jaw Gripper in GraspIt!

In Table 7.8 different grasps are visualized which were planned by the Eigengrasp planner for Baxter parallel jaw gripper and the Schunk WSG 50 paralell jaw gripper. Then the physics simulation from GraspIt! for the parallel jaw gripper model is started on the best grasp planned. This evaluation shows the successful implementation of the gripper modell in the GraspIt! physics engine. The first two objects (castle and bird) are taken from the Columbia grasp database (see [39]). The other objects are delivered with the GraspIt! framework.
In the different figures in Table 7.8 red cones can be seen. This cones represent *friction cones*, which visualizes the possible forces that can be applied at this contact point taking the friction of the material into account.

Table 7.8: Visualization of grasps on different objects of Baxter and the Schunk WSG 50 gripper.

Object	Baxter's gripper	Schunk WSG 50
castle		
bird		
flask		
ball on stick		
glass		

8 Conclusion

Within this thesis an autonomous order picking system was presented. In order to keep the system modular and portable a 3-TIER architecture was developed. The planning layer utilized an AI planner, which uses a PDDL description of the planning problem. The planner received the description of system skills, as well as start and goal state and provided a list of planned skills. Each skill is composed of skill primitives. These skill primitives needed to address manipulation of the box, grasping items and perceptual tasks. Manipulation of the transport box included grasping its handle through a visual servoing routine. It also included a two arm manipulation of the box for moving it from the level to the tray and vice versa. Therefore iterative arm movements were planned and executed. Grasping tasks addressed the problem of picking and placing items including an online collision aware grasping pose calculation. Perception tasks were used in many skill primitives. For instance the exact box pose was detected out of a point cloud using a coarse to fine approach. The item was marked with an AR tag and so its pose could be detected. Further the box handle is measured continuously in the visual servoing routine.
This work used the middleware ROS and its components. For computing arm trajectories the MoveIt! framework was used. Grasping parts built uppon and extended the GraspIt! framework. Perceptional tasks on point clouds applied algorithms implemented in the PCL. For the proof-of-concept implementation the robot Baxter was used.

The evaluation pointed out the problems of this proof-of-concept system. In the evaluation the outcome of single skills was evaluated first. Within most skills, the biggest problem was that the execution of planned trajectories was aborted because Baxter was not able to execute them precisely enough. However these errors were detected by our system and reported to our high-level. Further we encountered the problem that the successful execution of

trajectories was related to influences like room temperature or how long the robot had been switched on.
Further the our transport box detection algorithm was compared to two other algorithms implemented in the PCL. The evaluation showed, that in this setup our algorithm performs better than the other two algorithms.

Further we included the parallel jaw gripper model in the GraspIt! framework, and added two parallel grippers to it: Baxter's gripper and the Schunk WSG 50.

9 Future Work

In this chapter improvements for the current system are discussed. The evaluation showed that the grasping skill need to be improved. There are two main reasons why the results are not overwhelming. First the detection of the transport box and the item have small uncertainties. Nevertheless these perceptions are already quite good. The second or the major reason for failing in grasping the item are Baxter's inaccurate movements. In order to cope with that, tactile sensors could be installed on the gripper. Therefore, after the endeffector reaches the *pre-grasp pose*, it can perform a reactive behavior using data from the tactile sensors. So the item should be securely grasped.
An alternative to the parallel yaw gripper would be Baxter's vacuum gripper. This gripper offers different opportunities for picking and placing items. Especially for not small objects the success rate in grasping is higher than with the parallel jaw gripper. Because the items cannot slip out of the gripper that easily. But the gripper would not be preferable if the objects are too heavy or have a very rough surface. Then the objects would properly more often lost then with using the parallel jaw gripper.

As already mentioned, Baxter's inaccurate movements should be improved. Therefore a better controlling strategy or a different controlling algorithm should be developed and implemented. This modifications should be implemented in the Baxter's Motor Controller. This controller processes the incoming joint commands and then the motors are actuated. This controller takes currently the incoming joint commands, joint limits, velocities into account and also ensures (in some modes) that the robot's arms do not collide with each other or the torso. But this controller does not take the current state of the robot like running time, temperature or environmental influences into account.

The execution time of the box detection should be improved. One suggestion concerns the coarse detection. The FPFH should not be calculated on the whole point cloud, since most parts of the box are smooth surfaces. In order to save computation time, only on the *interesting* parts of the cloud should be considered: the edges of the box. This edges could be detected using a Harris corner detector.
A further improvement of the perception modules concerns the item detection. An item detector could be implemented, that detects the item pose without being marked with an AR tag.

The suggested improvements concern the current system only, the next improvement makes the system applicable in *Smart Factories*. If the system should be employed in a real factory, the robot must be able to work at different working stations. Otherwise it would be very expensive to have one robot for one explicit working station. So if the robot should work at different working stations, the system needs to be expanded to cope also with path planning and execution for mobile platforms. This effects the manipulation too. For example if the movement's trajectory cannot be planned the robot could reposition itself in a way the movement can be planned.

Bibliography

[1] M.R. Pedersen, D.L. Herzog, and V. Kruger. Intuitive skill-level programming of industrial handling tasks on a mobile manipulator. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4523–4530, Sept 2014.

[2] Mikkel Rath Pedersen, Lazaros Nalpantidis, Aaron Bobick, and Volker Krüger. On the integration of hardware-abstracted robot skills for use in industrial scenarios. In *2nd International IROS Workshop on Cognitive Robotics Systems: Replicating Human Actions and Activities, (Tokyo, Japan)*, Nov 2013.

[3] Francesco Rovida, Casper Schou, Jens Skovand Dimitris Chrysostomou Andersen, Rasmus Skovgaardand Damgaard, Simon Bøgh, Mikkel Rathand Bjarne Grossmann Pedersen, Ole Madsen, and Volker Krüger. Skiros: A four tiered architecture for task-level programming of industrial mobile manipulators. In *International Workshop on Intelligent Robot Assistants, (Padova, Italy)*, 2014.

[4] J. Huckaby, S. Vassos, and H.I. Christensen. Planning with a task modeling framework in manufacturing robotics. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5787–5794, Nov 2013.

[5] D. Mcdermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. Pddl - the planning domain definition language. Technical Report TR-98-003, Yale Center for Computational Vision and Control,, 1998.

[6] A. Gerevini and D. Long. Plan constraints and preferences in PDDL3. In *ICAPS Workshop on Soft Constraints and Preferences in Planning*, 2006.

[7] K. Okada, Y. Kakiuchi, H. Azuma, H. Mikita, K. Murase, and M. Inaba. Task compiler : Transferring high-level task description to behavior state machine with failure recovery mechanism. In *IEEE International Conference on Robotic and Automation (ICRA)*, May 2013.

[8] Elisabeth Freeman, Eric Freeman, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. O' Reilly & Associates, Inc., 2004.

[9] Daniel S Weld. Recent advances in ai planning. *AI magazine*, 20(2):93, 1999.

[10] Avrim L Blum and Merrick L Furst. Fast planning through planning graph analysis. *Artificial intelligence*, 90(1):281–300, 1997.

[11] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated planning: theory & practice*. Elsevier, 2004.

[12] Jörg Hoffmann. Ff: The fast-forward planning system. *AI magazine*, 22(3):57, 2001.

[13] Chih-Wei Hsu, Benjamin W Wah, Ruoyun Huang, and Yixin Chen. Handling soft constraints and goals preferences in sgplan. *ICAPS*, page 54, 2006.

[14] A. Cowley, B. Cohen, W. Marshall, C.J. Taylor, and M. Likhachev. Perception and motion planning for pick-and-place of dynamic objects. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 816–823, Nov 2013.

[15] D. Berenson, R. Diankov, K. Nishiwaki, S. Kagami, and J. Kuffner. Grasp planning in complex scenes. In *7th IEEE-RAS International Conference on Humanoid Robots*, pages 42–48, Nov 2007.

[16] S. Chitta, E.G. Jones, M. Ciocarlie, and K. Hsiao. Mobile manipulation in unstructured environments: Perception, planning, and execution. *IEEE Robotics & Automation Magazine*, 19(2):58–71, June 2012.

[17] A.T. Miller and P.K. Allen. Graspit! a versatile simulator for robotic grasping. *IEEE Robotics & Automation Magazine*, 11(4):110–122, Dec 2004.

[18] Matei Ciocarlie, Corey Goldfeder, and Peter Allen. Dexterous grasping via eigengrasps: A low-dimensional approach to a high-complexity problem. In *Robotics: Science and Systems Manipulation Workshop-Sensing and Adapting to the Real World*, 2007.

[19] J. Felip, J. Laaksonen, A. Morales, and V. Kyrki. Manipulation primitives: A paradigm for abstraction and execution of grasping and manipulation tasks. *Robot. Auton. Syst.*, 61(3):283–296, March 2013.

[20] M. Prats, P. Martinet, A.P. del Pobil, and Sukhan Lee. Vision force control in task-oriented grasping and manipulation. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1320–1325, Oct 2007.

[21] N. Vahrenkamp, M. Do, T. Asfour, and R. Dillmann. Integrated grasp and motion planning. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 2883–2888, May 2010.

[22] Paul J Besl and Neil D McKay. Method for registration of 3-d shapes. In *Robotics-DL tentative*, pages 586–606. International Society for Optics and Photonics, 1992.

[23] François Pomerleau, Francis Colas, Roland Siegwart, and Stéphane Magnenat. Comparing icp variants on real-world data sets. *Autonomous Robots*, 34(3):133–148, 2013.

[24] Radu Bogdan Rusu, Nico Blodow, and Michael Beetz. Fast point feature histograms (fpfh) for 3d registration. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 3212–3217. IEEE, 2009.

[25] Radu Bogdan Rusu, Gary Bradski, Romain Thibaux, and John Hsu. Fast 3d recognition and pose using the viewpoint feature histogram. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2155–2162. IEEE, 2010.

[26] Cheng-Tiao Hsieh. An efficient development of 3d surface registration by point cloud library (pcl). In *International Symposium on Intelligent Signal Processing and Communications Systems (ISPACS)*, pages 729–734. IEEE, 2012.

[27] Matthias Nieuwenhuisen, David Droeschel, Dirk Holz, Jorg Stuckler, Alexander Berner, Jun Li, Reinhard Klein, and Sven Behnke. Mobile bin picking with an anthropomorphic service robot. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 2327–2334. IEEE, 2013.

[28] Ioan Sucan and Lydia E Kavraki. A sampling-based tree planner for systems with complex dynamics. *IEEE Transactions on Robotics*, 28(1):116–131, 2012.

[29] Matei Ciocarlie, Kaijen Hsiao, Edward Gil Jones, Sachin Chitta, Radu Bogdan Rusu, and Ioan A Şucan. Towards reliable grasping and manipulation in household environments. In *Experimental Robotics*, pages 241–252. Springer, 2014.

[30] Ioan Sucan, Maciej Moll, Lydia E Kavraki, et al. The open motion planning library. *IEEE Robotics & Automation Magazine*, 19(4):72–82, 2012.

[31] Kai M Wurm, Armin Hornung, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. Octomap: A probabilistic, flexible, and compact 3d map representation for robotic systems. In *Proc. of the ICRA 2010 workshop on best practice in 3D perception and modeling for mobile manipulation*, volume 2, 2010.

[32] 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.

[33] Sachin Chitta, Ioan Sucan, and Steve Cousins. Moveit! *IEEE Robotics & Automation Magazine*, 1(19):18–19, 2012.

[34] R.B. Rusu and S. Cousins. 3d is here: Point cloud library (pcl). In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–4, May 2011.

[35] Conor Fitzgerald. Developing baxter. In *IEEE International Conference on Technologies for Practical Robot Applications (TePRA)*, pages 1–6. IEEE, 2013.

[36] Tully Foote. tf: The transform library. In *IEEE International Conference on Technologies for Practical Robot Applications (TePRA)*, Open-Source Software workshop, pages 1–6, April 2013.

[37] Robin Murphy. *Introduction to AI robotics*. MIT press, 2000.

[38] Eric Freeman, Elisabeth Robson, Bert Bates, and Kathy Sierra. *Head first design patterns*. " O'Reilly Media, Inc.", 2004.

[39] Corey Goldfeder, Matei Ciocarlie, Hao Dang, and Peter K Allen. The columbia grasp database. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 1710–1716. IEEE, 2009.