University of Innsbruck

Institute of Computer Science

Intelligent and Interactive Systems

Robot Simulation and Motion Planning

Martin Griesser

B.Sc. Thesis

Supervisor: Emre Ugur, PhD
Univ.-Prof. Dr. Justus Piater, PhD
11th September 2014

Abstract

This document explains how to write IIS theses using the LaTeX 2_ε 'iisthesis' class[1], which is itself based on the LaTeX 2_ε 'book' class and is intended for compiling with pdflatex. This document does not tell you how to structure your thesis.

[1] version 00.06 , released on 2013/06/13.

Acknowledgements

Thank you to the friendly members of the IIS Team.

Contents

List of Figures

List of Tables

Declaration

By my own signature I declare that I produced this work as the sole author, working independently, and that I did not use any sources and aids other than those referenced in the text. All passages borrowed from external sources, verbatim or by content, are explicitly identified as such.

Signed: ..... Date: .....

Chapter 1

Introduction

Operating a robot and developing high level control code for it is a cumbersome task. Very often various groups of developers are working simultaneously on different parts of the software. But in most cases there is only one device that can be used for testing and so only one can run his/her code at a time. The others have to wait until they get access to the robot. Moreover, tests on real robots always come with a high level of risk. Incorrect algorithms can lead very fast to damages on robot components or their environment and entail costly repairs. In the worst case even people can get hurt by uncontrolled robot motions.

The solution to those problems is the usage of a simulator that mimics the robot components and their behavior as accurate as possible. It has to provide the same control interface so that each part of the software can get tested on the simulator before it gets utilized on the real robot. Those considerations motivated the first part of this thesis – the realistic replication of the IIS-Lab robot setup on a suitable simulation platform. This involves the creation of an exact model of the robot setup, containing the various robot components and their environment. The necessary steps are explained in Chapter 2.

The second part of the thesis is about motion planning. Moving the robot hand cannot follow any arbitrary trajectory towards a target pose. During that motion it might collide with itself or any other obstacle within it's environment. That means those trajectories have to be planned carefully to avoid accidental collisions and to generate smooth and well controlled robot motions. This also involves to create and maintain an internal representation of the robot and it's environment, a step that is common to the simulation part of the thesis. Chapter 3 shows the configuration and integration of the motion planning framework MoveIt into the IIS-Lab robot setup along with some usage examples.

Chapter 4 focuses on MoveIt's grasping functionality. It shows, how a reference 'Pick and Place' task can be planned with the planning tools and executed on the simulator and the real robot as well.

In Chapter 5, some test results will be presented, analyzing the quality of the various planning algorithms and approaches.

1.1 IIS-Lab Robot Setup

The robot setting, that was considered in this thesis can be seen in Figure 1.1 The main part of the robot setup in the IIS-Lab consists of an aluminium torso with two mounted KUKA light weight robot arms. Those arms have 7 degrees of freedom (DOF) each one can carry up to 7kg of payload. Additionally a Schunk SDH gripper can be mounted on each arm.

- **KUKA LWR 4+ arm**

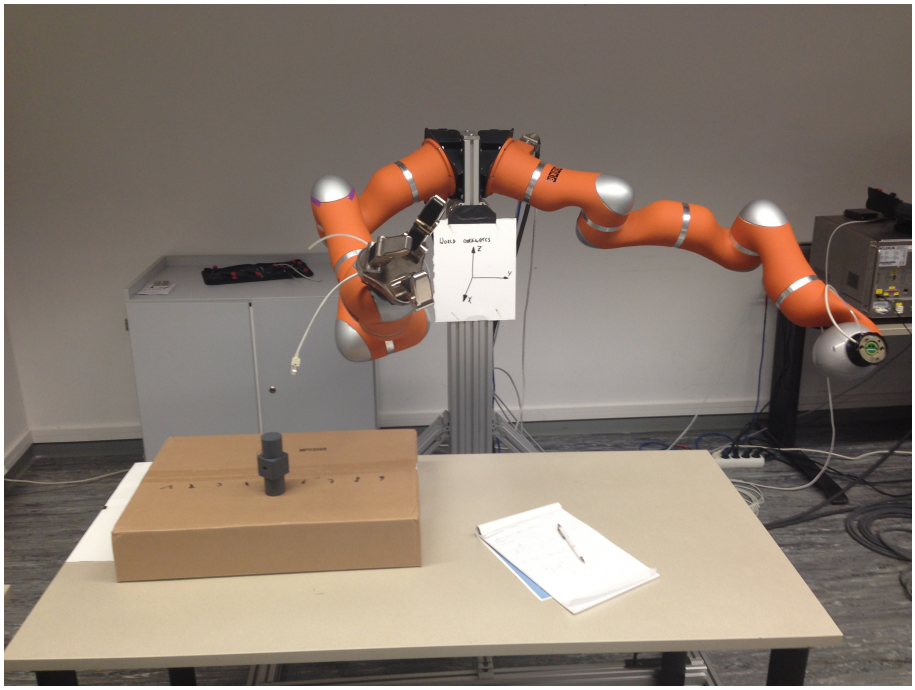

Figure 1.1: Current setup in the IIS-Lab TODO: provide more recent image

Description of the arm(7 DOF, max. 7kg payload, 16kg weight, very flexible,...

- **Schunk SDH gripper**
  Description of the gripper(3-finger-gripper, different grasp types, as powerful as human hand, very sensitive)

- **Kinect camera**
  Description of Kinect(RGB camera, Depth sensor, 3d data under any light conditions)

1.2 The Robot Operating System(ROS)

As the control of the robot components is based on ROS, a brief introduction shall be given here. As stated in Quigley et al. (2009), ROS is not an operating system in the classical sense. It can be seen as a communication layer, providing various mechanisms for inter process communication. A ROS system consists of a number of nodes. Each node is an independent computation unit and runs in it's own process, adding some clearly defined functionality to the overall system. For example one node can be responsible for planning, another one for perception or controlling the hardware. Nodes communicate with each other by passing messages, using the ROS communication infrastructure. Messages are strictly typed data structures that can be defined in a special message composition language. They can be composed of primitive types like Float, Integer or Strings and also of other message types. That makes it possible to define custom messages for each use case. Messages can be published to topics. A topic can be seen as some kind of address, consisting of a string. Topics can be organized in namespaces. If two instances provide the same interface, each one can be operated in it's own namespace. This is important, as for example the simulator should provide exactly the same interface as the real robot. Without namespaces it would not be possible to operate the simulator and the real robot in parallel. So Each node can publish and also subscribe to a number of topics. It is also possible

that more than one node publishes to the same topic. The organization of the nodes of a system can be visualized as a graph.

A node can also advertise a service. This is a special type of topic that allows a synchronized message exchange. In contrast to topics, a service with a given name can only be offered by one node. The service message is composed of a request and a response part. When a node sends a service request to another node it will block, until the advertising node has handled the request and delivers a response. The nodes that make up a ROS system can be located on different machines. One machine has to be the dedicated master that runs the roscore. Other machines can connect to the master via network. ROS also provides a centralized parameter server that can be used to store configuration data for the various nodes. A system usually consists of a large number of nodes that have to be configured and started. This can be done in launch files. Those are simple textfiles, holding startup information and configuration details for one or more nodes in an XML syntax. Using the roslaunch command, a set of nodes can be configured and launched at once. It is important to understand this terminology because especially the terms node, topic and message are heavily used throughout this thesis. A more detailed documentation can be found on the ROS website.

1.3 Project Targets

Both objectives

- **Simulation**
  Realistic replication of the robot setup on a suitable simulation platform Generation of realistic sensor data Visualization of collisions Implementation of a ROS interface, corresponding to that one of the real robot

- **Motion Planning**
  Configuration and integration of a motion planning framework Implementation of a benchmark pick and place task, executable on simulator and real robot Provide an easy to use interface to the planner Analyse the quality of the various planning algorithms

Chapter 2

Robot simulation

This chapter focuses on the simulation part of the thesis. The objective is the creation of a simulation model, particularly designed for the IIS lab robot setup. The model has to reflect the properties and behaviour of the contained robot components as good as possible. Certainly the simulated components have to provide the same control interface as their real counterparts, allowing to test and optimize control code on the simulator before utilizing it on the real robot. Preferably, the control code sees no difference about on which instance it is executed. The recommendations on such a solution can be summarized as follows:

- The simulator needs to be able to generate realistic sensor and feedback data that can be used by the control software. This includes forces and torques that are measured within force sensors, the current state of the various robot joints (position, velocity, effort), but also RGB and depth images, usually produced by Kinect cameras and vision sensors.
- The simulation solution has to provide exactly the same ROS control interface as the real robot. This interface essentially consists of a number of ROS topics, that can be used to send control commands to the various different robot components or to read actual joint states and sensor data.
- The utilized simulation platform has to provide a graphical user interface that allows to visualize the motions of the robot and it's interaction with the environment. Possibly accidental collisions of robot parts have to be registered and should also be visualized.
- In order to be used by as many people as possible, it is very important that the solution is really easy to use and does not require a long lead time. Therefore it has to be put particular focus on usability.

The following sections explain in detail, how this goal was reached. At the beginning stands the process of finding a suitable simulation platform that meets the requirements and the considered criteria. After that, the chosen simulation platform V-Rep is introduced and an overview about how to design dynamic simulations is given, explaining some of the necessary terminology. Subsequent sections focus on the necessary steps to achieve the final solution, namely finding and modelling required robot components, assembling and configuring the final simulation scene and the implementation of the ROS control interface.

2.1 Choosing a suitable simulation platform

The tasks executed on the robot are in most cases variations of so called *pick and place* tasks. An object gets picked up, lifted and placed somewhere else within the robot's workspace. Therefore

joint target positions are sent to the control interfaces of the various robot components and they execute the commanded motions if possible. The question, if the execution is possible at all and in that case in which velocity, is influenced by a number of dynamic parameters. The maximum effort of the motors in the joints is limited. If the force that acts upon a joint is higher than the maximum effort of the motor it will not be able to maintain its current position or to reach the desired target position. This can happen if the picked object is to heavy or if the robot collides with an immovable object in it's environment. The forces that act upon each single joint are influenced by a number of parameters like the position within the kinematic chain of the robot, the summed own weight of the robot components and also the weight of a possibly additional payload.

The required solution should be able to provide a realistic simulation of those dynamic interactions. Therefore the utilized simulation platform has to take use of a powerful physics engine. A physics engine is a software component, that is capable of computing parameters of physical processes and the dynamic properties of the involved objects. Examples for such engines are the Open Dynamics Engine[1] (ODE) and Bullet physics[2]. Some of the evaluated simulation platforms even provide a number of different physics engines to choose.

The candidates that have been taken into account were Gazebo[3], V-Rep[4], MORSE[5] and Openrave[6]. After some investigation only two of them (Gazebo and V-Rep) had been evaluated in greater detail. Criteria for the selection had been:

- Which physics engine is used respectively is it possible to choose among various engines?
- Usability and stability
- Expandability
- Availability of required model components (arm model, gripper,...)
- Quality of the documentation
- Licence issues

Taking into account those criteria it went clear that V-Rep will be the simulation platform of choice. In some initial tests V-Rep seemed to be much more stable than Gazebo and the user interface is very intuitive. Another important point is that V-Rep ships with a fully functional model of the KUKA LWR4+ robot arm.

2.2 The Virtual Robot Experimentation Platform(V-Rep)

V-Rep is a powerful robot simulation platform, developed and maintained by Coppelia Robotics. The current version (V3.1.2) provides the ability to choose from three configurable physics engines (ODE, Bullet, Vortex – only trial version) for simulating dynamic processes. It also contains a very comfortable editor for modelling robot components and simulation scenes. In the *shape edit mode* it is possible to edit and simplify meshes. This is very important because for simulating dynamic processes only simple shapes with a low amount of vertices, edges and

[1] http://www.ode.org
[2] http://bulletphysics.org
[3] http://gazebosim.org/
[4] http://coppeliarobotics.com
[5] http://www.openrobots.org/wiki/morse/
[6] http://openrave.org

faces should be used to reduce complexity. The contained model browser provides a rich set of different robot models, static objects and various sensors, ready to use. A powerful feature are V-Rep's so called *calculation modules*. They can be configured to provide additional calculation functionalities on groups of scene objects. The *collision detection module* is capable of detecting and visualizing all kinds of collisions within the simulation scene. The *inverse kinematics calculation module* allows to solve inverse kinematics problems for robot components. The behaviour of the simulator is highly customizable via a rich programming API for C++ as well as the scripting language LUA[7]. V-Rep is no open source software but it provides a free licence for educational units and can therefore be used for research purposes. A more detailed introduction to V-Rep can be found in Freese et al..

2.3 Dynamic simulations in V-REP

For a better understanding of the modelling process it is necessary to explain a few fundamental concepts about designing dynamic simulations in V-Rep. This section just covers those aspects that are important for the underlying project. A more detailed explanation can be found in the official V-Rep documentation[8].

Each simulation scene in V-Rep is composed from a number of models that are arranged within the environment. A model consists of various scene objects, combined in a tree like structure to mimic the kinematic chain of a robot component. Each model has a dedicated model base and constitutes a sub-tree of the scene hierarchy. The model base is the root element of the model tree. There exist various types of scene objects within V-Rep, but only those, which are important for the implementation will be explained here.

- **Shape**
  A shape is a 3 dimensional body. Shapes represent the visual parts as well as the dynamically enabled parts of the scene. It is necessary to distinguish between primitive shapes (Cylinder, Cuboid, Sphere, Plane and Disk) and complex shapes (triangle meshes). Primitive shapes are much easier to handle for the physics engine as there can happen a lot of optimization during dynamics calculations. Complex shapes usually look better and therefore they are used mainly as the visual part of the model but also by the collision detection module. Various shapes can be combined to groups and therefore treated as one single object. Shapes can be defined as *static* or *non-static* objects. The position of a static object is fixed relative to it's parent node within the scene hierarchy and will not change during simulation. Non-static objects underlie gravity and will fall down if they are not constrained by a dynamically enabled joint or a force sensor connection. It is also necessary to distinguish between *respondable* and *non-respondable* shapes. Respondable shapes have a clearly defined mass and moment of inertia and therefore they create collision reactions when colliding with other respondable objects during simulation. Only respondable shapes are considered during dynamics calculation. Usually a model in V-Rep is composed of a visual part, consisting of complex shapes and a hidden part, consisting of groups of primitive shapes that are configured to be used for the dynamics calculations. This issue will be covered again when explaining the creation of the hand model. Each shape also has some additional flags, defining special attributes used by the *calculation modules*. The *collidable* attribute states that a shape has to be considered during collision

[7] http://www.lua.org
[8] http://www.coppeliarobotics/helpFiles

detection. The *renderable* flag marks a shape to be recognized by a *vision sensor*. There are more flags available but only those two were used within this project.

- **Joint**
  A joint is a flexible connection between two rigid parts of a robot. It has to be distinguished between *revolute* or *prismatic* joints with one degree of freedom and *spherical* joints with three degrees of freedom. In the arm and hand model only revolute joints are used. Joints can be passive or actuated by a motor. The motor settings include maximum force or torque and velocity limits. If the control loop is enabled then a desired target position for the joint can be set. The controller will then try to reach this position, based on it's PID settings while respecting the motor limits. It would also be possible to program a custom control loop for each joint but for the current project this is not necessary. Joints can be operated in various different modes. A joint in *torque/force mode* is simulated, using the physics engine. This is the most realistic control mode. A joint in *inverse kinematics mode* is controlled by the inverse kinematics calculation module and if additionally the option *hybrid operation* is selected then the dynamic parameters of the joint are also taken into account. Operating a joint in *dependent mode* means that it's position depends on the position of another joint within the scene. How this dependency looks like can be configured by setting a *dependency equation*.

- **Vision sensor**
  States a simulated image capturing device. A vision sensor can capture images of the simulation scene, depending on it's configuration. It can deliver RGB image sequences as well as depth images. Vision sensors are used in the Kinect camera model.

- **Force sensor**
  A force sensor in V-Rep is a rigid connection between two dynamically simulated objects. The calculated forces and torques can be measured and visualized. It is also possible to define a maximum force the connection is able to bear. When this maximum value is exceeded, the connection will break.

- **Dummy**
  Dummies are the simplest scene objects at all but they provide some important functionality, especially for the various calculation modules of V-Rep. They can be understood as the origin of a named reference frame with a configurable position and orientation within the simulation scene. Two dummies can be linked as *tip-target pairs* to be used by the IK calculation module. The importance becomes more clear when the construction of the simulation scene will be explained, particularly when configuring the inverse kinematics calculation module.

Groups of scene objects can be organized in *collections* and treated as one single entity. Collections play an important role in the collision detection module.

2.4 Designing the simulation scene

This section explains in detail the structure of the simulation scene and the contained model components. The setup contains three types of robot component - the Kuka LWR4+ robot arm, the Schunk SDH-2 gripper and the Kinect camera. For each one of them, a simulation model was required in order to be able to build up the scene. V-Rep ships with realistic and fully functional model for the Kuka arm and also for the Kinect camera. But as there is currently no

model for the Schunk gripper available, it had to be created from scratch. The necessary steps of the modelling process are explained in the next section.

2.4.1 Modelling the Schunk SDH-2 gripper

A model within a V-Rep simulation scene basically consists of three major parts. First there are the shapes that form the visual part of the model. Those shapes are triangle meshes for each single link of the robot component. Preferably they are not to complex, i.e. the amount of faces is not to high. The second part is formed by the joints that connect those shapes and allow to actuate the flexible parts of the model. The third part consists of the dynamically enabled parts of the model. Theese shapes form the body of the model, how it is seen by the physics engine. Those respondable parts are usually an approximation of the original meshes, composed from groups of primitive shapes. Those parts are very important as they make the model realistic and allow to interact with other respondable objects within the scene. Without them, the model would just move through other bodies as only respondable shapes are able to produce collision reactions. The steps, described within this section enclose the modelling of that three parts for the gripper model.

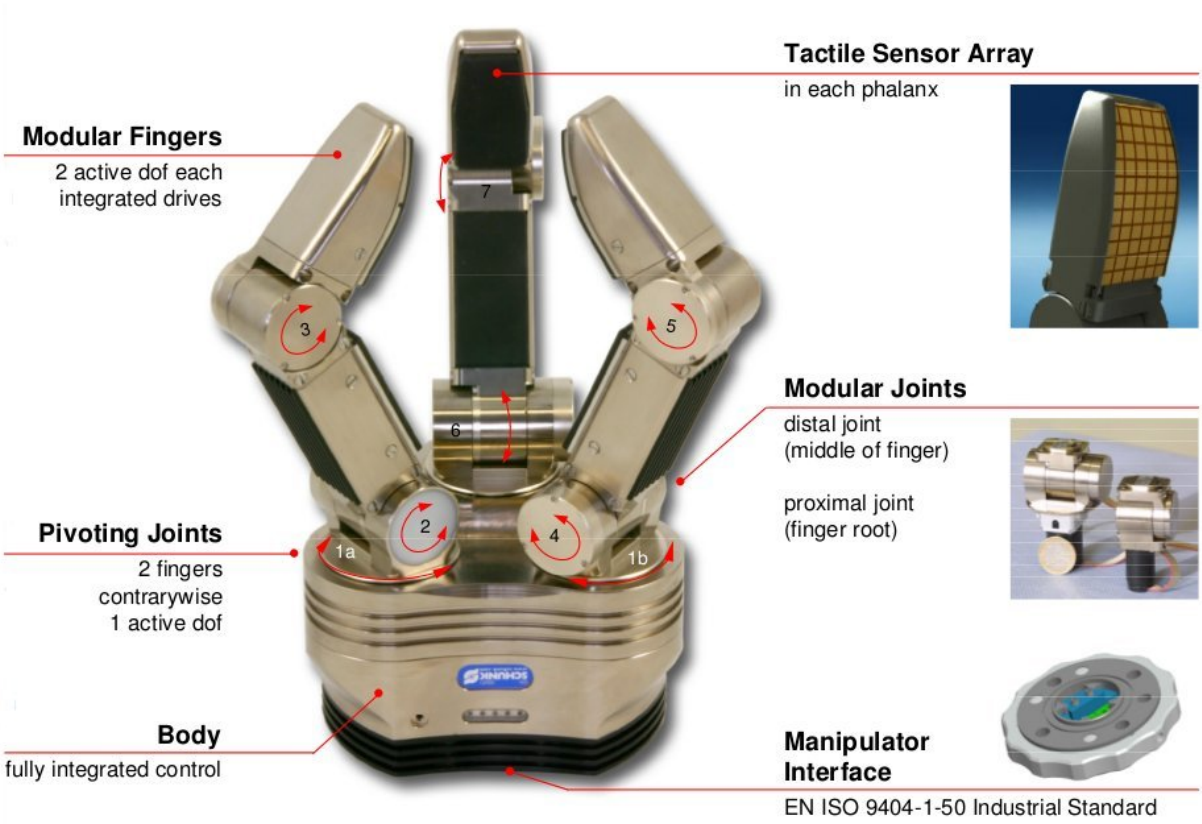

Figure 2.1: Schunk SDH-2 gripper

Image source: Schunk (2010)

Figure2.1 shows the Schunk SDH-2 hand. The gripper has 3 fingers, each one containing two modular joints. The joints located closer to the wrist are called the *proximal* finger joints whereas the joints, actuating the finger tips are called the *distal* finger joints. Two of the fingers can be rotated along their vertical axis but they are connected contrariwise. That means if one

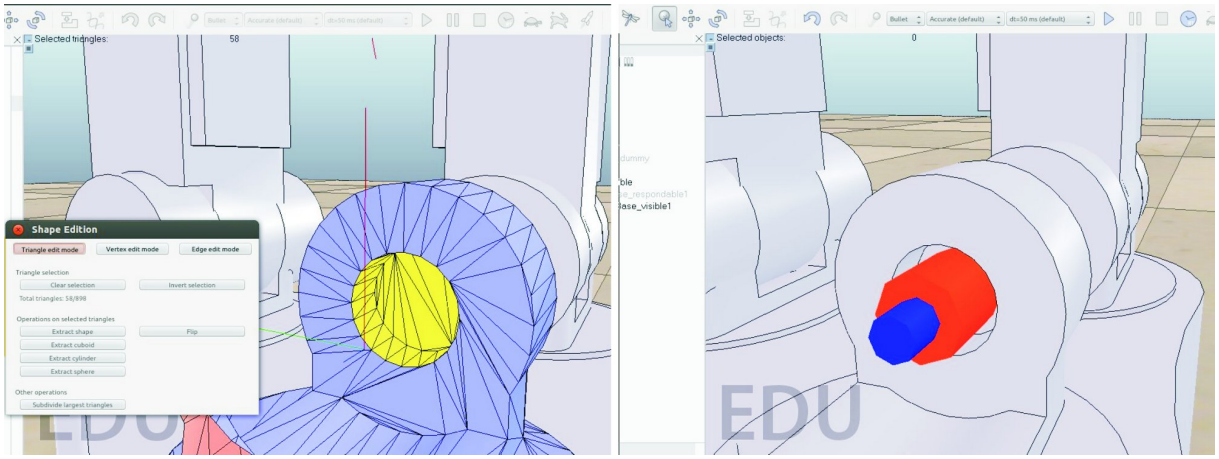

Figure 2.2: Placing a joint within the model

finger rotates to the left, the other one is rotated to the right for the same angle, actually adding one additional degree of freedom. Those two joints are called the *pivoting* joints. The visual part of the hand model basically consists of 4 different shapes - the wrist, finger knuckles, finger links and finger tips. Suitable meshes were taken from the schunk_description[9] ROS package and imported into the V-Rep robot editor. They have then been arranged according to the technical description. The next step was to insert and arrange the gripper joints on their appropriate locations.

Therefore it was important to determine the correct position and orientation for each single joint within the model to allow the correct movement of the fingers. This was achieved by using the *shape edit mode* and select the cylinder shaped area within the mesh, where the joint has to fit. From that selection a cylinder was extracted and the joint was then centred within this newly created cylinder. Those steps had to be repeated for all 8 joints. The placement process can be seen in Figure2.2.

The left image shows the extraction on the target area. On the right image, the joint is already placed on it's appropriate location.

After placing all the joints and links within the scene, the model tree was adjusted to form

[9]http://wiki.ros.org/schunk_description

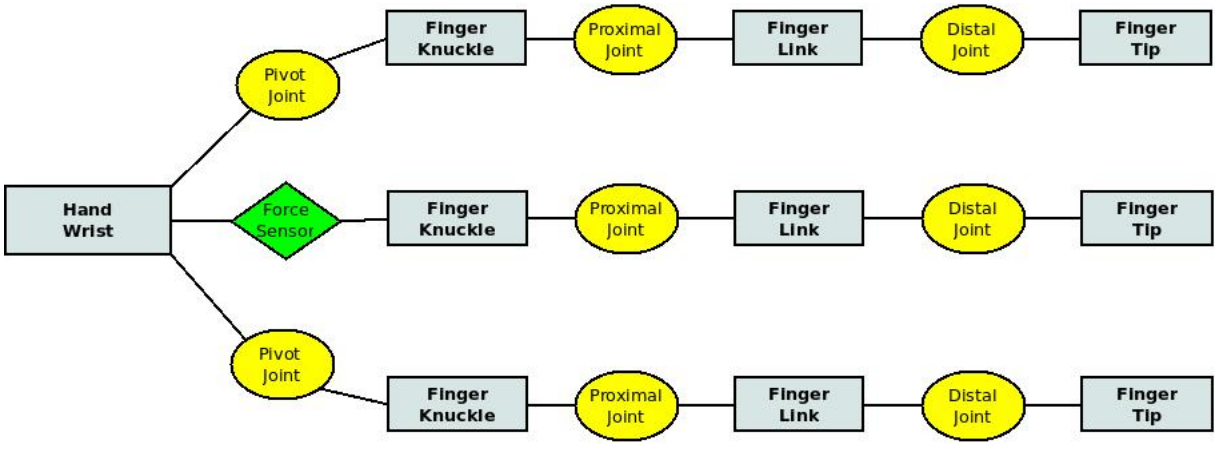

Figure 2.3: Kinematic chain of the Schunk gripper

the kinematic chain of the hand as can be seen in Figure2.3. The dynamic parameters of the joints were set according to the technical description. Those parameters include the joint limits, maximum velocity and maximum effort. As the positions of the pivot joints are connected to each other, the configuration of the second finger's root joint looks sightly different. To achieve this mirroring behaviour the joint is operated in the *dependent mode*, which means it's position *depends* on the position of a connected joint and that dependency is expressed as *dependency equation*. The configured equation just copies the actual position but the joint is operated in opposite rotational direction.

The meshes only form the visual part of the model, but they are to complex to be used for dynamics calculations. So the shape of each link had to be approximated by groups of primitive shapes. This was achieved by executing the following steps for each single part of the model:

- Within shape edit mode locate parts of the mesh that could be approximated by a primitive shape (cuboid, cylinder), by selecting suitable groups of vertices
- Extract the primitive shape by using the corresponding editor functionality
- Repeat those steps until the most important parts of the link are approximated that way
- Group those primitive shapes to treat them as one single object
- Adjust the dynamic parameters (mass, material settings, inertial matrix)
- Adjust the local respondable mask
- Give the group the same name as the corresponding mesh, but with the *_res* suffix
- Remove the extracted shapes from the current visibility layer because they are just used for dynamics calculations

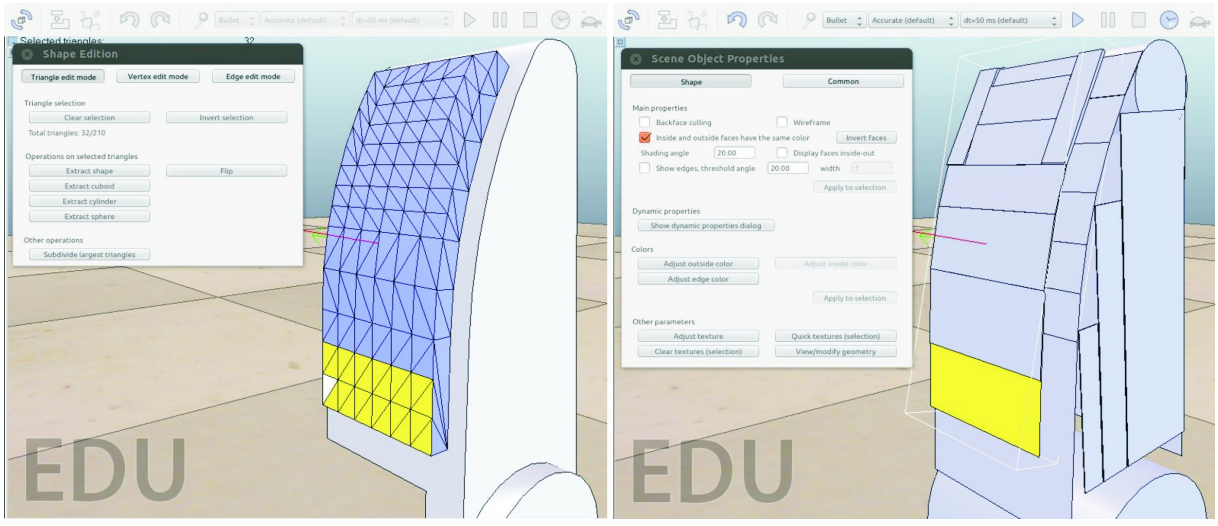

Figure 2.4: Process of approximating the original mesh with pure shapes

The extraction process is visualized in Figure2.4. Dynamic parameters like mass and inertial matrix have been provided by Alex Rietzler. The predefined *highFrictionMaterial* setting was used for each single part of the finger because that showed better results when picking up objects

later on.

The last step of the modelling process was the adjustment of the model hierarchy. Root element of the hand model is the respondable part of the wrist which is also the dedicated model base. It is very important to follow the V-Rep guidelines for designing dynamic simulations because if the hierarchy is wrong, the model will simply fall apart when starting the simulation (detailed information can be found in the corresponding chapter[10] of the V-Rep dodumentation). Each non-static and respondable shape has to be connected to it's parent by a joint or a force sensor. The visual part of the link is always a child object of it's corresponding respondable. That way, the kinematic chain of the gripper is formed.

2.4.2 Assembling the scene

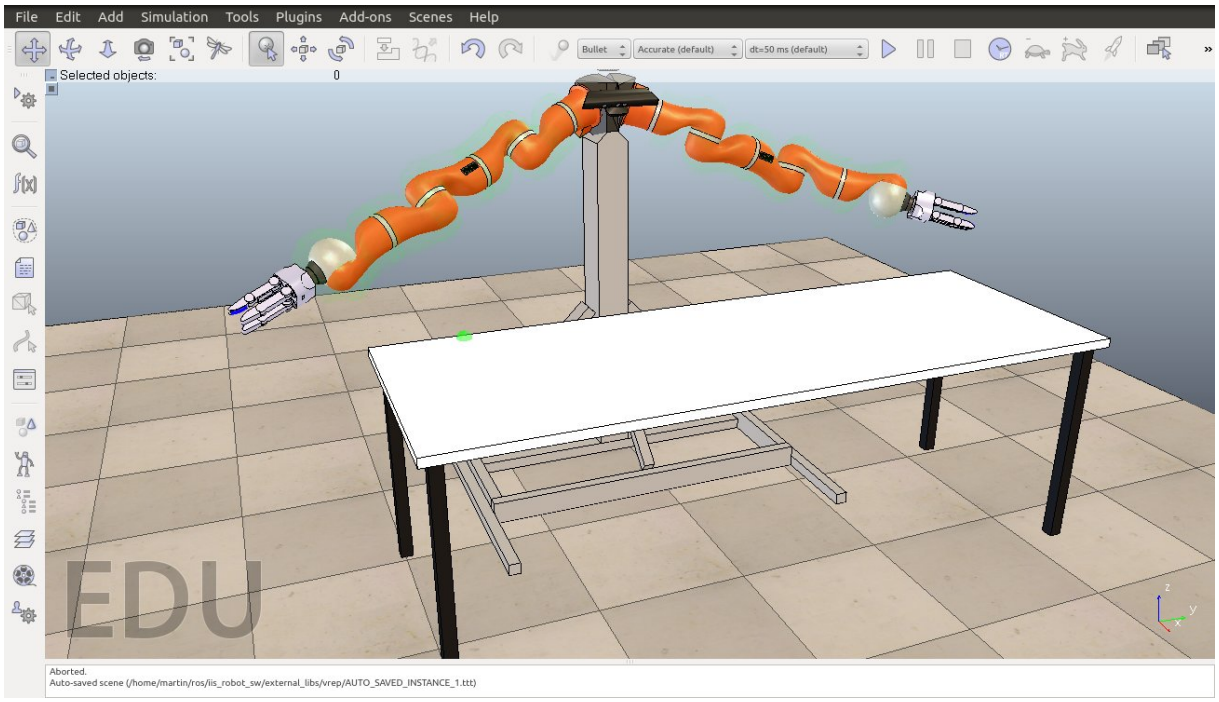

Figure 2.5: Structure of the V-Rep simulation scene

After finishing the gripper modelling process, each necessary component was available to build up the simulation scene as can be seen in Figure2.5. The final scene consists of a model of the robot's torso, two KUKA LWR4+ arm models with attached grippers and a model of the table in front of the robot. The origin of the world reference frame is located on the upper left side of the table, indicated by a slightly green shimmering sphere. A dummy object called *ref_frame_origin* was placed at that location. Each position calculation later on will happen relative to that dummy element. If it is necessary to move the origin to another location within the workspace, this can simply be achieved by just moving that dummy to the required location. The position and orientation of the torso, the table and the two arms are set relative to the world reference frame. (TODO: transformation was provided - how was it achieved?). The grippers were placed on the tip of each arm. Within the scene hierarchy they are child elements of the last node in the corresponding arm tree. The correct rotation and offset was measured on the

[10] http://www.coppeliarobotics.com/helpFiles/en/designingDynamicSimulations.htm

real counterpart and then adjusted accordingly.

The plate and the legs of the table were modelled as group of primitive cuboids. The table is defined as respondable to ensure that it will produce a collision reaction if a robot component collides with it. But as it is a static object it's position will not be influenced by such a collision because it is fixed within the scene. The material setting is set to *highFrictionMaterial*.

The Kinect camera model was also taken from the model browser. As it's position and orientation in the real world is not fixed and might change from time to time, it's position within the simulation scene is just an approximation to reflect the real world setting as good as possible.

2.4.3 Configuring the collision detection module

One of the requirements to the final solution is the ability to detect and visualize possible accidental collisions of the simulated robot with itself or it's environment. Moreover it would be a convenient feature to have some kind of warning if some of the robot's parts come dangerously close to an obstacle during movement. This could be achieved by creating a *collision shield*, which means an enlarged version of the robot and do additional collision checking. These considerations lead to two different types of possible collisions. *Soft collisions* are collisions, detected on the collision shield of the model. They just indicate a warning that the robot comes very close to an object it is not allowed to touch. *Hard collisions* mean that a robot component directly hits another collidable object. This would be also a collision in the real world. The solution should be able to distinguish cleanly between those two types.

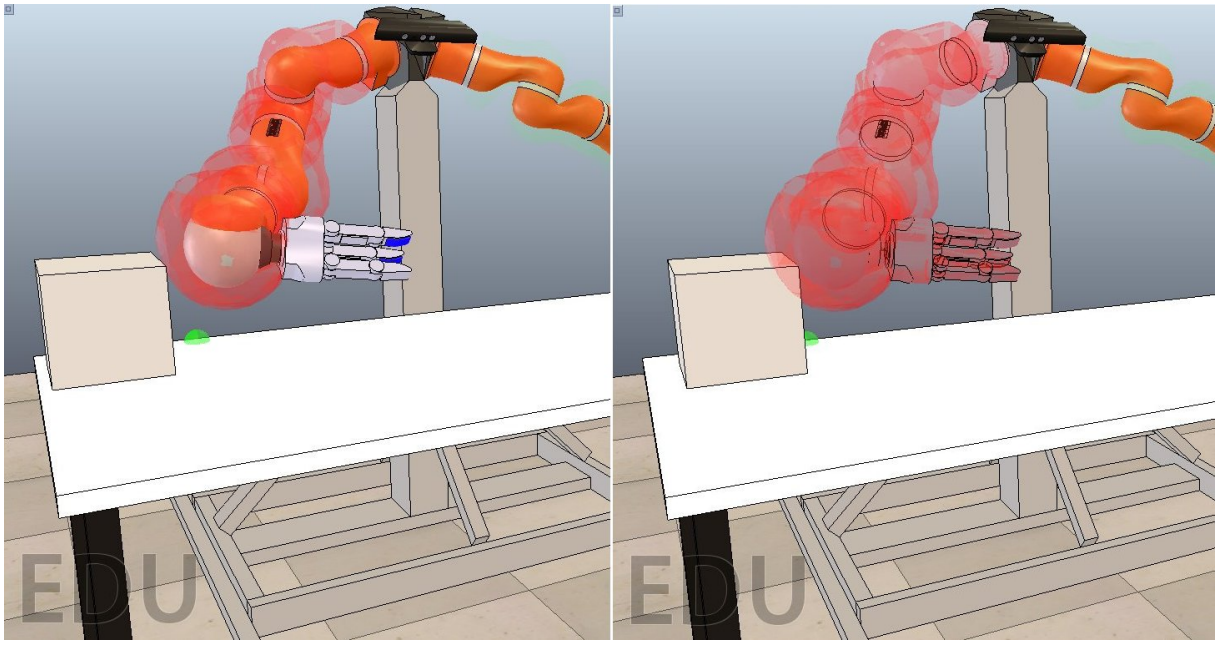

Figure 2.6: Collision detection and visualizing

That goal was achieved by using V-Rep's *collision detection module*[11]. This calculation module is capable of detecting and visualizing collisions within the simulation scene by check-

[11] http://www.coppeliarobotics.com/helpFiles/en/collisionDetection.htm

ing for interferences between *collidable* shapes. It is important to understand that the collision detection module only *detects* collisions. Producing *collision reactions* is the responsibility of the *dynamics module* and the chosen physics engine. Usually the shapes that form the visual part of the model are used for collision checking. The configuration is done by registering one or more *collision objects*. Each collision object consists of a *collider* and a *collidee*. Both of them can be single shapes or collections of shapes. The collidee settings also offer the option *all other collidable objects in the scene*. In that case, the collider is checked against all elements in $S = \{ s \mid s \in scene \text{ and } s \notin collider \}$. The big advantage in using collections is that they allow to exactly describe which shapes should be checked against which other ones. Detected collisions are visualized by applying different colouring either to the collider or to the collidee as can be seen inf Figure2.6. The left image shows a hit on the collision shield, the right image indicates a direct hit.

The first step during realization was to model the collision shield. Therefore it was necessary to create an additional shape for each link that is slightly larger than the original one. This modelling process started at the second link of each robot arm and for all subsequent links the following steps were performed:

- Create a copy of the shape that forms the visible part of the robot link
- Morph the copied object into a group of convex shapes to reduce complexity. This can be done by using the corresponding shape editor functionality.
- Ungroup the resulting group of shapes and merge them into one single shape
- Grow the resulting shape, but only in x and y direction of it's own reference frame. The resulting mesh should be approximately 10cm larger but keep the same height.
- Adjust the outside color to make it green and nearly transparent. The collision shield should be visible but not occlude the original model.
- Apply a meaningful name to the newly created shape to be be able to easily identify it within the model hierarchy. Here the name of the original shape with the '_col' suffix was used.
- Make the new shape a sibling of the original one within the model hierarchy.
- Adjust the shape object properties. Define the new shape to be *static* and *non-respondable*.
- Disable the *collidable* flag on the shape. This behaviour will be overridden in the collection settings later on when configuring the collision detection module.

This modelling process is visualized in Fig???.

The next step was to adjust the configuration of the *collision detection module*. Each arm requires two *collision objects*, defined as listed in Table2.1. The first two collision objects are designed to detect direct hits on the corresponding arm. Therefore the colliders (collections *leftArm* and *rightArm*) are checked for collisions against *all other collidable objects in the scene*. This setting is only possible because the *collidable* flag is disabled within the scene object settings of the collision shield elements, which means that they are excluded from collision checking by default. The collision objects, named with the *Shield* suffix are responsible to check for hits solely on the collision shield elements. The colliders (collections *leftArmShield* and *rightArmShield*)

Collision object	Collider	Collidee
left_arm	leftArm	all other entities
right_arm	rightArm	all other entities
left_armShield	leftArmShield	exLeftArmShield
right_armShield	rightArmShield	exRightArmShield

Table 2.1: Configured collision objects

Collection	Definition
leftArm	$\{ s \mid s \in$ *subtree of left arm* $\}$
leftArmShield	$\{ t \mid t \in$ *element of left collision shield* $\}$
exLeftArmShield	$\{ u \mid u \in$ *scene and* $u \notin$ *subtree of left arm* $\}$
rightArm	$\{ v \mid v \in$ *subtree of left arm* $\}$
rightArmShield	$\{ w \mid w \in$ *element of left collision shield* $\}$
exRightArmShield	$\{ x \mid x \in$ *scene and* $x \notin$ *subtree of right arm* $\}$

Table 2.2: Collection definitions

consist only of the collision shield elements. As those elements were defined without the *collidable* flag, the option *Collection overrides collidable properties* is selected in the corresponding settings to explicitly enforce collision checking when using those collections. The collections, defining the collidees (*exLeftArmShield* and *exRightArmShield*) include all other objects except those, contained in the left/right arm's subtree. This exclusion is necessary because otherwise those collision objects would detect collisions between the shield elements and the other arm links. The collection definitions are listed in Table2.2. All collision objects are defined not to be handled explicitly. This means that V-Rep does not check them automatically on each simulation pass. It has to be done manually and will be explained later on in the section about control interface implementation.

2.4.4 Configuring the IK calculation module

The ROS control interface needs the ability to set arm target positions in *joint space* or in *Cartesian space*, depending on the selected control mode. Joint space targets are relatively easy to handle as that only means to set the target position of each single joint. Setting targets in Cartesian space requires to solve the inverse kinematics problem for the corresponding robot component. Therefore V-Rep offers the *inverse kinematics calculation module* [12]. This calculation module allows to define and register various *IK groups*. Each IK group has to contain at least one *IK element*, defining the kinematic chain, constraints and desired precision settings (linear and angular). IK elements can be configured to enforce position constraints and/or orientation constraints for each single axis. The kinematic chain is specified by selecting the dedicated base link and the tip. The tip is a dummy object, indicating the end effector reference frame. The IK target is defined by another dummy object. Tip and target have to be linked, forming a *IK, tip-target* connection by selecting the appropriate link type within the dummy object settings. Figure2.7 shows a schematic description of that concept. The target pose is set by placing the target dummy at the desired location. The IK calculation model will then adjust the joint positions until the tip pose matches the target pose, respecting joint limits and configured constraints and tolerance values. The joints within the specified kinematic chain need to be operated in *inverse kinematics mode*, otherwise the module is not able to control them.

[12] http://www.coppeliarobotics.com/helpFiles/en/inverseKinematicsModule.htm

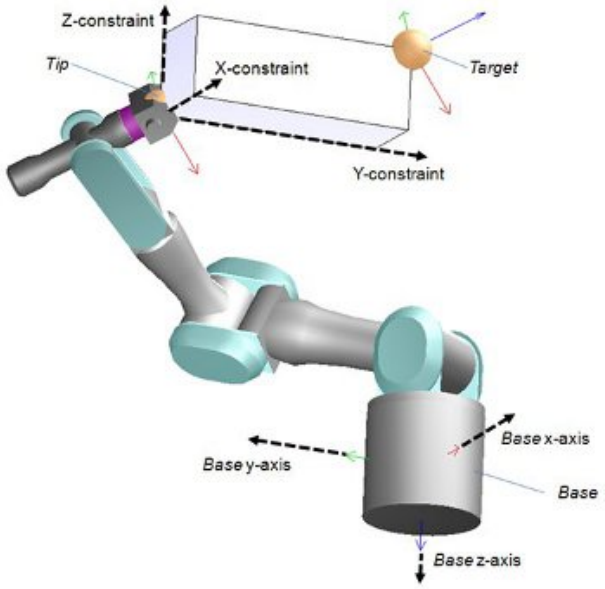

Figure 2.7: IK calculation module concept
Image source: http://www.coppeliarobotics.com/helpFiles/en/solvingIkAndFk.htm

The IK calculation module allows to define various IK groups for each robot component with differing configurations. Along the collection of IK elements, the IK group settings include the desired calculation method and the maximum amount of calculation iterations to use. Available IK calculation methods are *pseudo inverse* (PI) and *damped least squares* (DLS). As stated by Buss (2004), the PI method is usually faster than DLS. The tradeoff is that PI tends to be unstable in configurations where the target position is unreachable. That fact leads to a jittery behaviour of the manipulator. The DLS method provides higher stability in such situations but requires more calculation time.

IK group	**Method**	**Iterations**	**Prec. lin/ang**
left_arm	PI	9	0.001 / 0.1
left_arm1	PI	3	0.002 / 0.2
left_arm2	DLS	3	0.002 / 0.1
right_arm	PI	9	0.001 / 0.1
right_arm1	PI	3	0.002 / 0.2
right_arm2	DLS	3	0.002 / 0.1

Table 2.3: IK group definitions

Three IK groups have been created for each arm. The configuration settings are listed in Figure2.3. They were chosen, following the guidelines from the V-Rep documentation. The first two groups use the faster PI calculation method. The first one allows a higher amount of maximum iterations while demanding stricter precision settings than the second one. The third one is designed to increase stability especially for positions close to singularities. Therefore the DLS method was chosen with an increased position tolerance value. That configuration is less performant because of the DSL calculation method and therefore it is only used if the other groups failed to find a solution. The IK groups are called sequentially until one of them is able to solve the problem. This process is explained in the ROS control interface section later on.

2.5 Implementing the ROS control interface

In the real world, each type of robot component has its own, clearly defined ROS control interface. Those interfaces are composed from sets of inbound and outbound ROS topics that allow sending commands and to retrieve state data. The simulated components have to provide exactly the same ROS interface as their real counterparts, using similar topic names and message types. The structure of this control flow can be seen in Figure2.8.

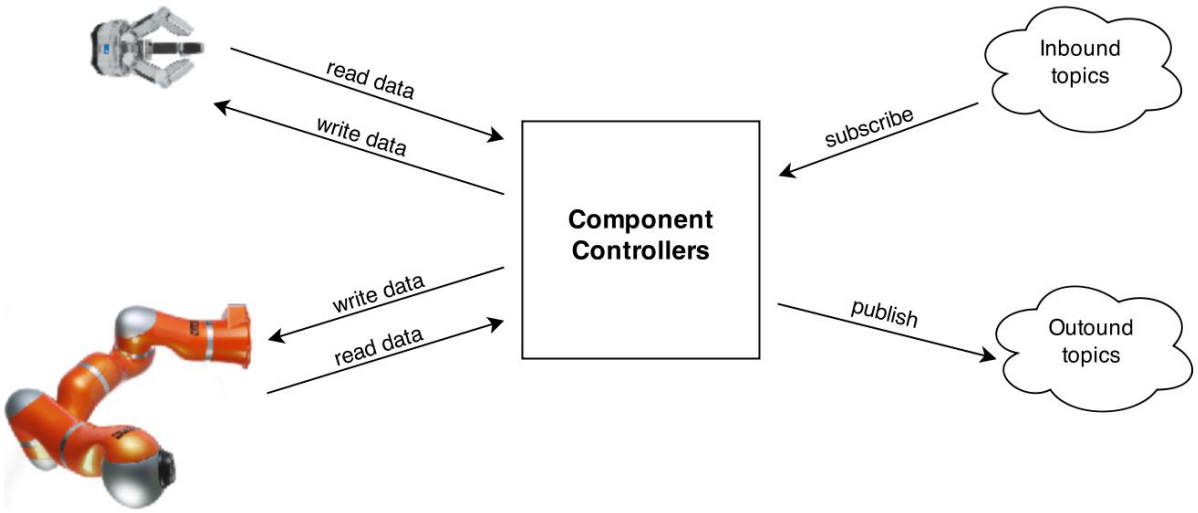

Figure 2.8: Control flow structure

The existing ROS interface that is part of the V-Rep distribution is not suitable as it only provides a very general approach for controlling joint values and reading state data. Therefore V-Rep provides various extension points, allowing to add custom functionality. The final solution was implemented as a *simulator plugin*, written in C++ and using V-Rep's *regular API*[13]. This approach states the most flexible solution as this API provides more than 400 functions that can be used to extend the simulator functionality. A plugin is a compiled library file, written in C++ that has to follow some V-Rep specific naming conventions and must reside in the V-Rep working directory. The library file gets automatically loaded on V-Rep startup and runs in the main simulation thread. The source code is located in the *iis_simulation* package that is part of the *iis_robot_sw* repository. The following section describes the design of the plugin architecture and the most important components within the package.

2.5.1 Plugin architecture

A plugin is a compiled library file, written in C++. This library needs to be placed in the V-Rep working directory and follow the V-Rep specific naming conventions. On startup, V-Rep looks for library files, prefixed with `libv\_repExt`. Matching files are automatically loaded. A plugin runs in the main simulation thread - that means it has to be programmed really carefully to avoid performance leaks during simulation. The plugin has to provide a clearly defined interface, consisting of 3 function definitions:

- `unsigned char v_repStart(void* reserved,int reservedInt)`
  This function is called on V-Rep startup and is used to perform necessary initialization steps. A return value of zero indicates that the initialization process failed and the plugin

[13] http://www.coppeliarobotics.com/helpFiles/en/apiOverview.htm

gets unloaded immediately. The current solution depends on a running roscore during startup. If that is not the case it is not able to work.

- `void v_repEnd()`
  Called before shutdown and is used to do some general cleanup and free allocated memory.
- `void* v_repMessage(int msg, int* auxData, void* custData, int* resData)`
  This function is called very often during the whole V-Rep lifecycle and is therefore a very performance critical method. Via this function V-Rep notifies plugins about events like start/end of simulation, simulation step, scene content change, scene switch and more. The plugin code can react to those events accordingly. In the current implementation, those messages are just passed to the software components that are responsible to handle the specific action.

The plugin code is organized as can be seen in the UML diagram in Figure2.9. The major parts of the system are described in the subsequent paragraphs.

SimulationComponent A simulation component is a single, reusable model of a specific robot component that can be utilized in different environments. Each component provides it's own clearly defined control interface. A simulation scene can contain multiple components from various types. The *SimulationComponent* class is the abstract base class for all simulation components. Currently there are existing two concrete implementations – the *LWRArmComponent* and the *SchunkHandComponent*. If the scenario should be extended and new components have to be introduced, it is necessary to create a new subclass of *SimulationComponent* and provide implementations for the abstract methods.

ComponentContainer This class represents the set of all identified simulation components within the current scene. On V-Rep startup an instance of *ComponentContainer* is created. Each time, the content of the current simulation scene changes, the method *actualizeForSceneContent* is triggered. This method then performs the following steps:

- It validates all currently registered *SimulationComponent* instances if they are still valid and present in the scene.
- It traverses the whole scene hierarchy to identify newly created components
- If a new component is identified, a corresponding concrete instance of *SimulationComponent* is created and added to the container.

The process of traversing the scene hierarchy and identifying components is explained in section2.5.2. During a running simulation, the *ComponentContainer* gets notified about each single simulation step. It simply forwards that message to all registered components. Those can then perform all necessary steps like triggering collision checking or handling IK groups.

ROSServer The *ROSServer* is a static class that encapsulates all ROS related functionality. It tries to initialize ROS on plugin startup, forcing a shutdown, if the connection to the master cannot be established. Otherwise it creates and maintains a ROS *NodeHandle* for the 'simulation' namespace. Each *SimulationComponent* can register *ComponentController* instances at the *ROSServer*. On simulation start it initializes all registered controllers with the maintained *NodeHandle*. The *ROSServer* gets also notified about each simulation step and forces the controllers to handle the received commands and publish all the necessary data. On simulation end it triggers the shutdown of all registered controllers.

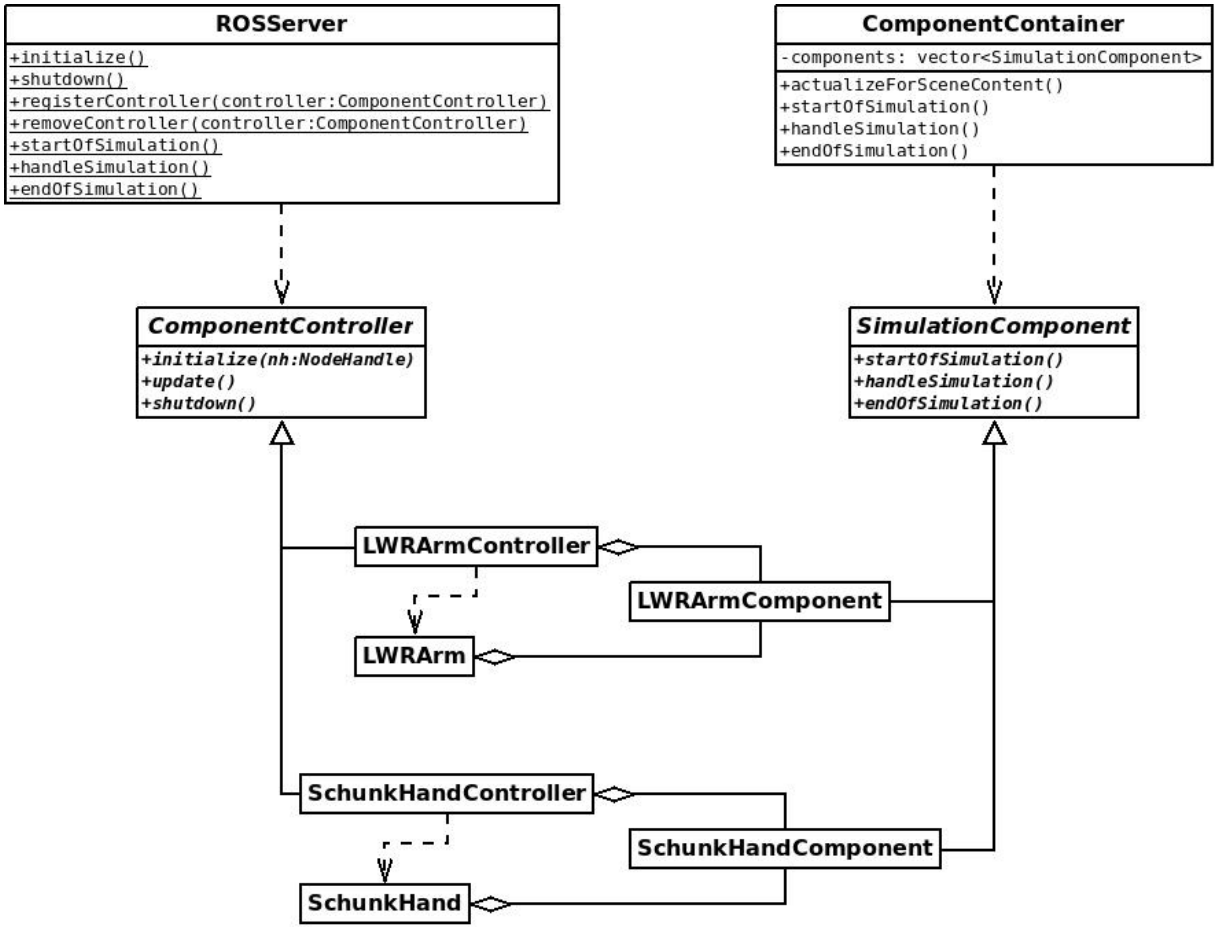

Figure 2.9: Simulator plugin architecture

ComponentController This is the abstract base class for all controllers. A controller actually represents the ROS interface of a specific simulation component, maintaining all the inbound and outbound topics used to control the simulated hardware. It is responsible for delegating commanded values to the underlying component as well as reading and publishing state data. Concrete implementations are the *LWRArmController* and the *SchunkHandController*. A *ComponentController* needs to be registered at the *ROSServer* and gets initialized on simulation start. Concrete implementations can use the provided NodeHandle to create all the necessary publishers and subscribers. The *update* method is called by the *ROSServer* on each simulation step and forces the controller to publish all the required data. The *shutdown* method is called by the *ROSServer* on simulation end, forcing the controller to shutdown all publishers and subscribers.

2.5.2 Identifying simulation components

The plugin functionality should not be tied to a specific simulation scene but to specific models of robot components. Each time a known component is added to the scene it should be recognized by the plugin and the corresponding instance of *SimulationComponent* has to be instantiated and added to the *ComponentContainer*. As visualized in Figure2.10, each simulation scene is a tree structure, consisting of various different types of scene objects. It is necessary to identify subtrees within this hierarchy that belong to known simulation components and should therefore

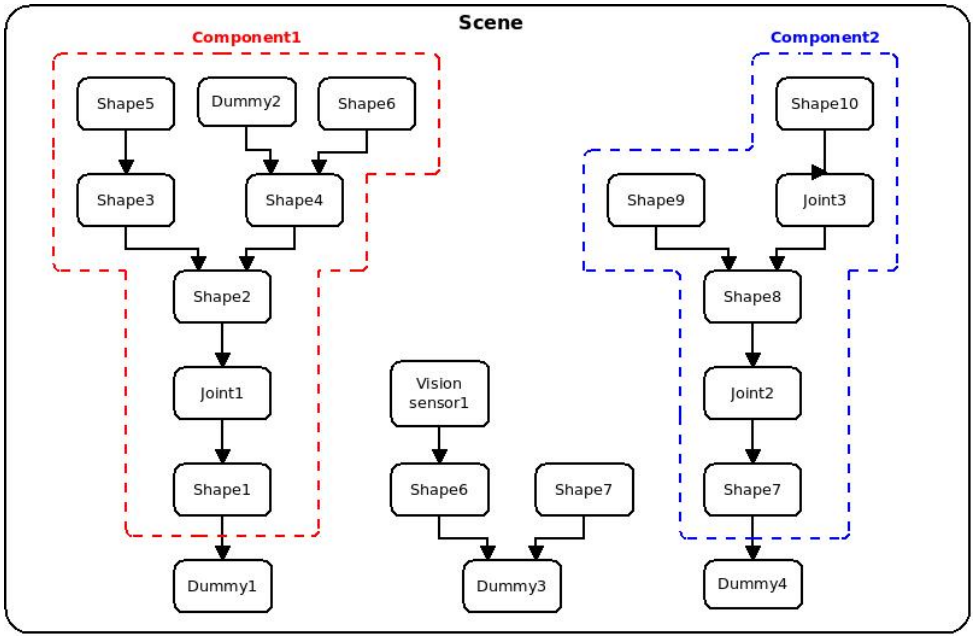

Figure 2.10: Sample scene hierarchy

be handeled by the plugin. If a component is identified, the plugin has to discover each single part of the model. Depending on the model this can be joints, force sensors, reference frame dummies, IK groups or collision objects.

One possible way would be to give each part a clearly defined, unique name and then search for those names within the scene. But this approach would require to hardcode each single object identifier and this is not a preferable solution for this problem. If a user accidentally changes a name inside the model tree then the solution is broken because the plugin looses connection to the underlying object and cannot control it any more. Here V-Rep's custom developer data functionality comes into play. It is possible to put auxiliary data segments to each single object in the scene. Those data segments are serialized together with the object and can be read programatically. Each data segment starts with a header number which is used to uniquely identify the data from a specific developer. A visualization of this concept can be seen in Figure2.11.

As the format of the data can freely be chosen, it was decided to use string representations of key/value pairs, separated by a colon (:). The key is an integer number, used to determine the type of the tagged object. The value segment can be used to provide additional information, e.g. the name of a joint. The left arm's model base for example is tagged with the data segment

```
2497,1:left_arm
```

The number 2497 is the header that identifies the data segment to belong to this plugin. The data segment identifies that element as the model base of a *LWRArmComponent* ($Key = 1$) with the name `left_arm`. The available keys are explained in the sections that correspond to the specific simulation components. When actualizing for scene content change, the *ComponentContainer* traverses the scene hierarchy and looks for objects, that are tagged as known components. On success, it creates the specific instance providing the object ID of the underlying scene object to the constructor. During initialization, the concrete *SimulationComponent* implementation then traverses the rest of the model subtree to extract all the remaining parts that belong to that specific model (joints, dummies, force sensors...), also by looking for tagged objects. The implementations for arm and hand model provide feedback output on the console window about

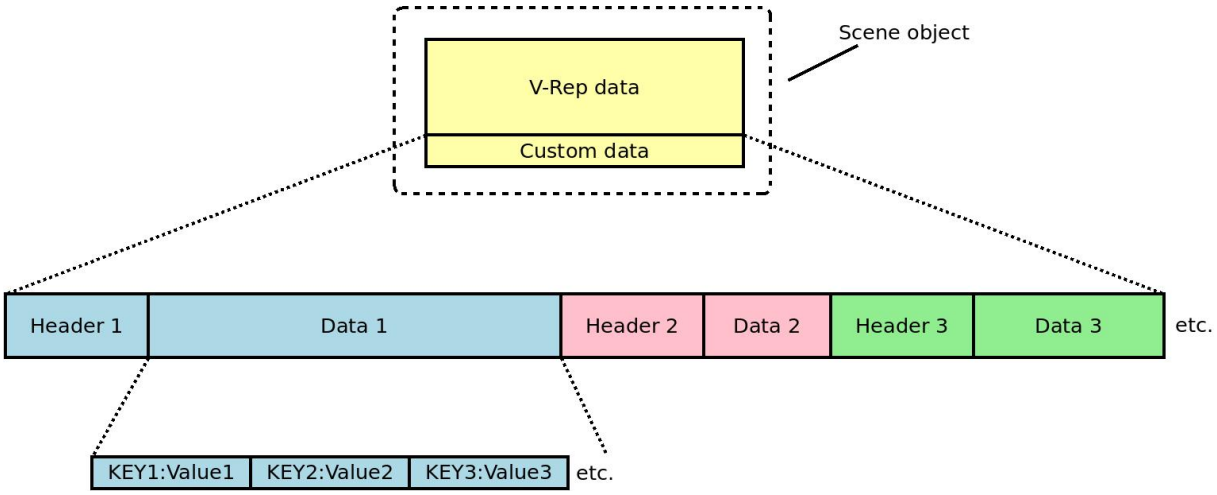

Figure 2.11: Custom developer data segments on scene object
Image source: http://www.coppeliarobotics.com/helpFiles/en/pluginTutorial.htm

the status of this initialization process and provide meaningful error messages in case that not all necessary parts of a model could be successfully located.

2.5.3 The LWRArmComponent

The *LWRArmComponent* is a concrete subclass of *SimulationComponent* and states the abstraction of a simulated KUKA LWR4+ arm model. It is composed of two parts - an instance of the *LWRArm* class and a corresponding *LWRArmController*, as can be seen in the UML diagram in Figure2.9. Their functionalities are explained in the following paragraphs.

LWRArm This class states the connection to the (simulated) hardware itself, providing methods to set commanded values and access all available state data like joint positions, Cartesian position of the end effector, current collision state and more. On creation, it is passed the base of the model tree as constructor argument. The base is identified by the `LWR_ARM_COMPONENT` tag. The value segment of this tag states the name of the arm which has to be unique within the scene. During initialization it traverses the model tree and extracts all required parts by searching for tagged scene objects as described in the previous section. The parts to identify are the 7 arm joints, the force sensor on the last link of the arm, IK tip and target dummies. The corresponding tags are listed in Table2.4. Additionally the *LWRArm* also requires a con-

Constant	Key	Value	Description
LWR_ARM_COMPONENT	1	Arm name	Identifies KUKA LWR arm model
LWR_ARM_JOINT	12	Joint name	Joint in KUKA LWR arm
LWR_ARM_CONNECTOR	13	-	Force sensor on arm tip
LWR_ARM_TIP	14	-	IK tip dummy
LWR_ARM_TARGET	15	-	IK target dummy

Table 2.4: Tag data items for LWRArmComponent

nection to the configured IK groups and collision objects. As there is no possibility to place custom developer tags on IK groups and collision objects, the extraction is done by using a special naming strategy. The first IK group needs to have the same name as the arm itself and it's existence is mandatory, leading to a configuration error message if no such group can be

extracted during initialization. Subsequent groups are optional and must have the same name with consecutive numbering (`[ARM_NAME]1`, `[ARM_NAME]2...`). That allows to reconfigure the IK calculation module and introduce additional IK groups without touching the plugin code. The two required collision objects are also searched, based on the name of the arm (`[ARM_NAME]` and `[ARM_NAME]Shield`). If one or both of them cannot be detected, an error message is stated on the console and the collision detection functionality will not work as expected.

To fullfill the requirements for the control interface, the arm has to be able to operate in *joint control mode* (FK mode) and in *inverse kinematics mode* (IK mode). Initially the arm starts in FK mode, which means the joints are operated in *torque/force* mode and accept target positions to be set. Switching to IK mode is done by changing the joint control mode to *inverse kinematics* mode, which means that they are conntrolled by the IK calculation module further on. Setting a target pose in Cartesian space is done by moving the IK target dummy to the required location and orientation. The known IK groups are then handled sequentially, until one of them is able to solve the problem and set the proper joint target positions. When switching from FK to IK mode, the IK target dummy has to be aligned with the tip dummy to prevent the arm from doing uncontrolled motions.

The current *collision status* is determined by using the configured collision objects. On each simulation step the collision object that is responsible for detecting direct collisions is handled first. If that one detects a collision, a direct hit is reported and it is not necessary to handle the second object at all, because a direct hit always implies a hit with the shield as well. Only if no direct hit was detected, the second collision object is handled. The outcome can be queried as the current collision state of the arm. The collision state is evaluated on each simulation step.

LWRArmController The *LWRArmController* is a subclass of *ComponentController* that encapsulates the whole ROS interface for the *LWRArmComponnt*. On creation it is passed a reference to the underlying *LWRArm*. The arm controller offers 4 basic control modes:

- **Joint control mode**
  The controller accepts target positions in joint space via the `joint_control/move` topic. The message basically consists of a vector, containing a target angles for each single joint, measured in *radiant*. Incoming target positions are validated not to exceed a specified velocity limit, enforcing the constraint

  $$|c_i - t_i| < \delta, \forall\ i\ \in\ [0, 6] \tag{2.1}$$

  where c_i are current and t_i are target positions for the joints 0 to 6 and δ is the current velocity limit. If the velocity limit is violated, the message is dropped and a corresponding error message is published to the `sensoring/error` topic and written to the console output. The default limit is set to a value of 0.1 and can be adjusted via the `joint_control/set_velocity_limit` topic. Valid target positions are simply passed to the underlying *LWRArm* instance, which is operated in FK mode.

- **Cartesian control mode**
  The controller accepts target positions in Cartesian space via the `cartesian_control/move` topic. Therefore, the *LWRArm* needs to be operated in IK mode. The commanded target pose is also validated before accepting it. The constraint is defined as

  $$|c_x - t_x| < \sigma\ and\ |c_y - t_y| < \sigma\ and\ |c_z - t_z| < \sigma \tag{2.2}$$

where c is the current and t is the target position in Cartesian space and σ is the current Cartesian velocity limit. The velocity limit can be adjusted, using the `cartesian_control/set_velocity_limit` topic. Invalid messages are dropped and result in an error message as well, published in `sensoring/error` topic. Valid target poses are passed to the underlying *LWRArm*.

- **Follow mode**
  This mode is only on the simulator available and designed to mirror the behaviour of the real robot. The controller reads the joint states of it's real counterpart from the corresponding topic and uses it's current position as target. Therefore, the *LWRArm* is operated in FK mode. This results in an exact copy of the real robot's motions. This mode can be used to test the accuracy of the simulated model, for example by carefully moving the robot close to positions where it collides with the table and check when the collision is reported.

- **Stop mode**
  The arm does not accept any movement commands at all. Ongoing motions will be stopped immediately when switching the arm into this control mode.

The controller has to be registered at the *ROSServer* in order to be able to work. This section only covered the most important control modes and topics. The complete interface description can be found in the documentation, located in Appendix A.

2.5.4 The SchunkHandComponent

The *SchunkHandComponent* is designed, using the same approach as for the *LWRArmComponent*. Therefore only the most important facts will be stated here. The two composing parts are the *SchunkHand* and the *SchunkHandController*.

Constant	Key	Value	Description
SCHUNK_HAND_COMPONENT	2	Hand name	Identifies Schunk hand model
SCHUNK_HAND_JOINT	22	Joint name	Joint in Schunk hand model

Table 2.5: Tag data items for SchunkHandComponent

SchunkHand States the connection to a Schunk hand model. The only additional parts that have to be discovered during initialization are the 7 gripper joints and their corresponding names. Available tags are listed in Table2.5. The *SchunkHand* class provides methods to set joint positions and to retrieve current joint states. Additionally it allows to modify the motor strength for each single joint, based on a percentage of maximum force. This reflects the possibility of adjusting the motor currents in the real hand, which also leads to the effect that the maximum motor strengths of the finger joints can be increased or decreased.

SchunkHandController The *SchunkHandController* provides an implementation of the ROS interface for the Schunk gripper model. The basic topics are quite similar to those of the *LWRArmController*, as there are `joint_control/move` for setting joint target positions or `joint_control/get_state` for retrieving joint states. Additionally the hand is able to perform grasps, based on specific *grasp types* as visualized in Figure2.12. This functionality can be accessed via the `joint_control/gripHand` topic. A grasp is defined by the parameters *grasp type* and *close ratio*. The controller then calculates the corresponding joint positions,

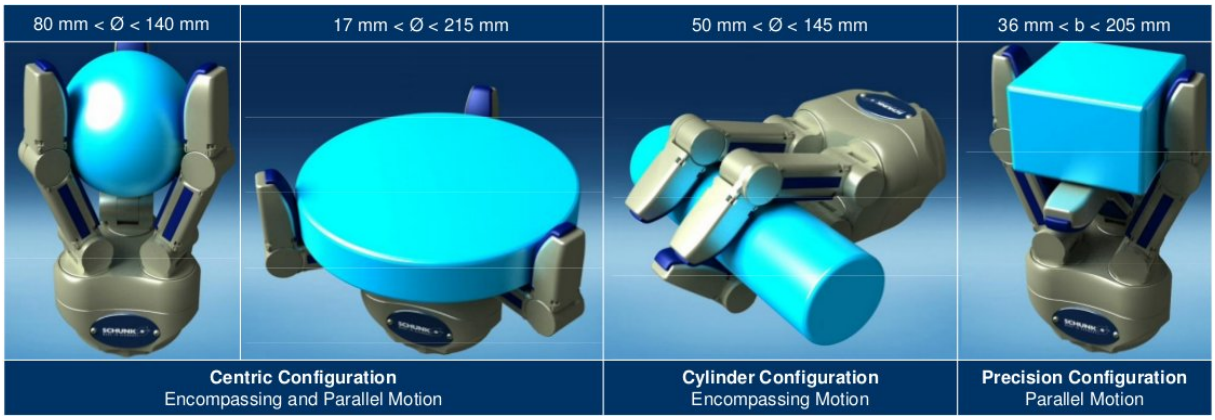

Figure 2.12: Grasp types `SPHERICAL`, `CENTRICAL`, `CYLINDRICAL` and `PARALLEL`
Image source: Schunk (2010)

based on those parameters and sets the desired target positions. Each grasp can be expressed, using three functions – one for the position of the pivoting joints and the other two for the positions of the distal and proximal finger joints. The utilized functions were taken from the original controller code of the `schunk_sdh` ROS package, located in the `iis_robot_sw` code repository. The definitions are listed in Table2.6. The grasp strength can be adjusted, using the `settings/set_motor_current` topic. The message basically consists of a vector, holding a value for each joint, defining a percentage of the maximum motor strength. The real gripper also provides a topic that allows to read the current motor temperatures. For the sake of consistency, the *SchunkHandController* also provides this topic, but as V-Rep is not able to simulate motor heatings, only constant fake values are published. A complete interface description can be found in Appendix A.

Name	**pivoting**	**proximal**	**distal**
CYLINDRICAL	0	$(-30+30x)\frac{\pi}{180}$	$(30+35x)\frac{\pi}{180}$
PARALLEL	0	$(-75+82x)\frac{\pi}{180}$	$(75-82x)\frac{\pi}{180}$
CENTRICAL	$\frac{\pi}{3}$	$(-75+82x)\frac{\pi}{180}$	$(75-82x)\frac{\pi}{180}$
SPHERICAL	$\frac{\pi}{3}$	$(-40+25x)\frac{\pi}{180}$	$(40+15x)\frac{\pi}{180}$

Table 2.6: Joint position functions based on close ratio x

2.5.5 Publishing Kinect camera data

The Kinect camera model does not contain any flexible parts that have to be controlled by the plugin. But it is necessary to make the images, captured by the simulated vision sensor available via ROS topics. This was achieved by using the corresponding functionality of the V-Rep default ROS interface that allows publishing vision sensor data, using the correct message types. Therefore a LUA script was attached to the base element of the camera model within the scene. This script simply extracts the object ID of the vision sensor and enables publishers for the captured RGB and depth images. The corresponding topics are `kinect1/sensoring/rgb_image` and `kinect1/sensoring/depth_image`.

Chapter 3

Motion planning

Chapter overview

3.1 Introduction

(Choset, 2005, p. 1–11) describes motion planning as to be the task of finding a collision free path from one robot *configuration* to another one. The classic path planning problem is the so called *piano mover's problem*, originally mentioned by Schwartz and Sharir (1983). It is assumed to have a piano, which states a three dimensional rigid body and a set of known obstacles. The problem is to find a continuous motion that moves the piano from it's current position to a given target position without touching any of the obstacles. Thereby the piano can freely be moved and rotated in Cartesian space.

A generalized version of the *piano mover's problem* is to find paths for a robot, composed from a set of rigid bodies, linked by joints while enforcing *constraints* during that motions. A *constraint* could be to avoid obstacles or to keep the robot's end effector in an upright position. Therefore it is important to have a representation of a robot's state that allows to determine the location of all robot parts. This representation is called the *configuration* of a robot and the *configuration space* is the set of all possible configurations, the robot is able to acquire. The dimension of the configuration space is the amount of *degrees of freedom* (DOF), which is the number or independent variables that are necessary to describe a configuration. An imaginary free flying piano has six degrees of freedom as it's configuration consists of the position and orientation $(x, y, z, roll, pitch, yaw)$ in Cartesian space. A robot arm with 7 joints has 7 degrees of freedom and it's configuration are the joint positions. The motion planning problem is to find a curve in the configuration space that connects start and goal configuration without violating constraints. This is a very complex problem and there exist various different approaches to find solutions. Examples are among others the *bug algorithms*(Choset, 2005, chapter 2), *potential functions*(Choset, 2005, chapter 4) or *sampling-based methods*(Choset, 2005, chapter 7). As the solution within this project only uses sampling based algorithms, a short overview about this class of methods will be given in the following section.

3.2 Sampling-based motion planning

Based on (omp, chapter 2), sampling-based motion planning can be seen as a powerful concept, capable of handling planning problems efficiently, especially for systems with many degrees of freedom. The general idea is to generate a uniform set of random sample points in the configuration space and then connect start and goal state by connecting the samples via collision

free paths, with respect to possible motion constraints. Those methods are usually faster than traditional approaches because it is not necessary to reason about the whole configuration space but only about a finite number of sample configurations. The majority of sampling-based approaches are known to be *probabilistic complete*, which means that the probability of finding an existing solution tends to 1 as the number of sample points increases to infinity. But they are not able to decide if a valid solution exists at all. The following definitions are used throughout this section to describe the concepts of sampling-based motion planning.

- **State space**
  The *state space* $\mathcal{S}$ is equal to the configuration space and consists of all possible robot configurations (states).

- **Free state space**
  The *free state space* $\mathcal{S}_{free}$ is a subset of $\mathcal{S}$, containing only collision free states.

- **Path**
  A *path* is a sequence of states. If each state within the path is contained in $\mathcal{S}_{free}$, it is called a *collision free* path.

The sampling-based motion planners can be categorized into two major types - *probabilistic roadmaps* (PRM) and *tree-based planners*. Common to both methods is that they create uniformly distributed samples within the free state space. As the shape of $\mathcal{S}_{free}$ is not explicitly known, the created sample states are checked for collisions before using them. The following paragraphs give a short overview about both approaches.

Probabilistic roadmaps That approach uses the sampled states to create a "roadmap" of the free state space. Therefore each sample point is connected to an amount of k nearby sample points via collision free paths. This is done by a local planner that simply interpolates between two points in the desired resolution while watching out for collisions. If no collision is detected, a new edge is introduced into the graph that is formed by the roadmap. After completing the graph, a planning query can be reduced to finding the shortest path within that graph that connects the start state and the goal state. (Images will follow...)

Tree-based planners A lot of different sampling-based planning algorithm are using the tree based approach, as there are for example (RRT, EST, SBL or KPIECE). The difference to PRM is that this method uses a tree data structure of the free state space, which means that the resulting graph contains no cycles. The root of the tree is the start state and the tree is then expanded towards the goal state by creating collision free connections between the sample points. If the goal is reached, the solution is found. The different approaches differ in the strategy that is used to expand the tree towards the goal state. (Images will follow)

Generally can be said that the tree based approaches are more adequate for *single query planning* because the tree usually does not has cover the whole free state space. A roadmap could be reused for subsequent queries. As the most methods also require the search of a nearest neighbour, the utilized distance metric is also a crucial part within sampling-based motion planning as it is not always easy to identify the optimal method for finding nearby states in systems with large degrees of freedom.

3.3 The MoveIt! motion planning framework

MoveIt[1] is an open source framework for motion planning. It is the successor of the previous arm navigation stack and therefore fully integrated into ROS. MoveIt was originally developed by Willowgarage[2] but since April 2012 it is maintained by the Open Source Robotics Foundation (OSRF). Figure3.1 gives an overview about the system architecture. Central part of the framework is the `move_group` node which provides a rich set of ROS topics and services that can be used to solve planning problems and execute calculated motion plans on connected hardware. Configuration of that node happens via the parameter server. It requires a complete kinematic and semantic description of the robot setup and a lot of other configuration parameters which will be described in subsequent sections. The most important parts of MoveIt are implemented as plugins which means that it is highly customizable. By default it uses the Kinematics and Dynamics Library[3] (KDL) as IK solver and the Open Motion Planning Library (OMPL) as planning plugin but it is possible to replace them with custom solutions if necessary.

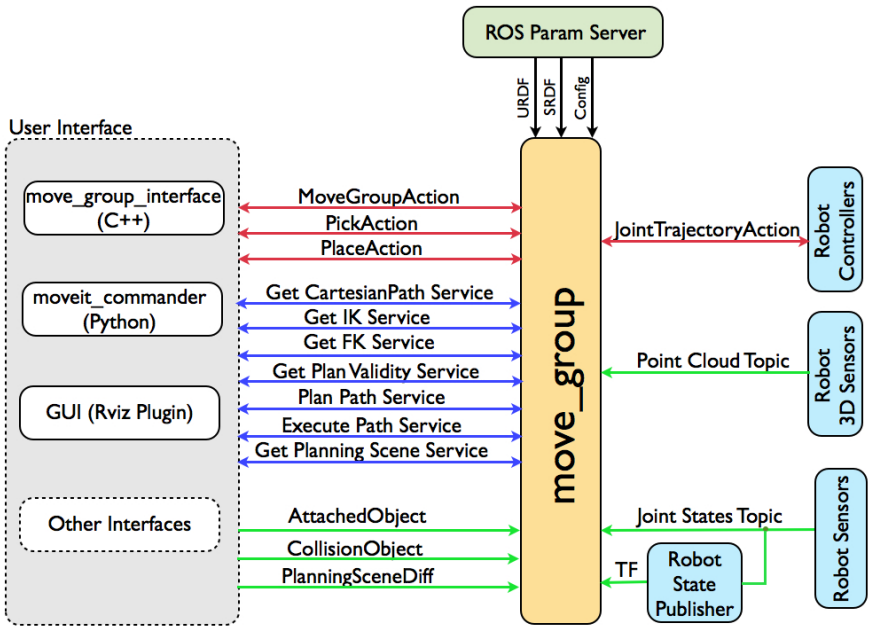

Figure 3.1: MoveIt architecture

Image source: `http://moveit.ros.org/documentation/concepts`

MoveIt maintains a planning scene which is an internal representation of the world, including the robot and it's environment. The base is a description of the robot and it's various planning groups. The kinematic description needs to be available in form of a URDF model. Necessary steps to create that model are described in Section3.4. Based on that URDF model, a semantic description (SRDF) is created that contains additional information about the setup, as explained in Section3.5. On top of those static descriptions, MoveIt allows to modify the planning scene on runtime via appropriate ROS topics.

[1] http://moveit.ros.org
[2] http://www.willowgarage.com
[3] http://www.orocos.org/kdl

To keep track of the current state of the robot MoveIt needs to be informed continuously about actual joint states. This is done via publishing the joint states to a specific topic. If there are additional objects within the robot's workspace they also have to be added to the planning scene. This can be done either by explicitly adding them via the corresponding topic or by integrating sensor information like Kinect camera data. MoveIt can then take those objects into account during motion planning and avoid collisions. But some collisions are intended. For example if an object has to be picked up, the gripper has to get in contact to this object. That means, collision checking for specific objects has to be (temporary) disabled to allow those controlled collisions. Therefore MoveIt maintains an *AllowedCollisionMatrix* to which objects can be added or removed. The connection to the hardware happens via the *FollowJointTrajectory* action interface. Each robot component has to provide this interface if it is intended to be control via MoveIt. This interface consists of a set of ROS topics that provide trajectory execution functionality and also allow monitoring the current execution status. The major contribution within this chapter to the whole project was to provide an implementation of this interface and connect it to the one that is currently available.

3.4 Creating the URDF model of the robot setup

The *Unified Robot Description Format* (URDF) is a markup language, designed to describe robots. The description happens in text files, in a special XML format. The most important elements in the XML specification[4] are:

- **<link>**
  Describes the all necessary properties of a specific robot link. Each link must have a unique name. The visual, inertial and collision details are configured in the corresponding subtags of the link element. The visual part as well as the collision model can either be composed from primitive shapes or from mesh files. If mesh files are used it is important that they are not to complex. Especially for the collision model it is recommended to use a simplified model to avoid a performance loss.

- **<joint>**
  Describes the properties of a joint. A joint is a connection between two links, having exactly one parent and one child link. Each joint states a new reference frame for it's child link and it is positioned relative to it's parent frame. A *fixed* joint is a rigid connection between parent and child link. There are different types of joints available but for the current project only *revolute* joints[5] are of interest. Details like limits, axis orientation and dynamic properties can be configured in the corresponding subtags of the joint element.

Those elements are used to form the URDF graph that exactly describes the kinematic chain of the robot components and their placement relative to each other as visualized in Figure3.2. This description can get very large as a lot of different components are involved. So it is possible to organize it into a set of text files, each one describing one part of the whole. For example one file describes the arm itself. Other files can then use that description and insert multiple instances of that arm. The `xacro` ROS package provides the necessary functionality to combine all those text files into one XML string. *XACRO* stands for *XML Macro* and is designed to parse xacro files and combine them into one single XML document, containing the resulting URDF description.

[4] http://wiki.ros.org/urdf/XML
[5] Rotational joint with one degree of freedom

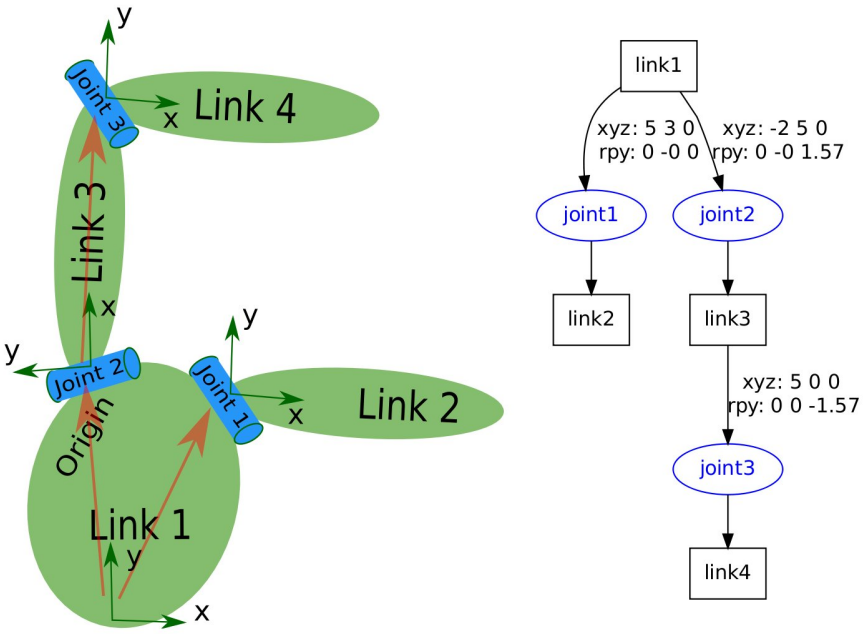

Figure 3.2: URDF graph

Image source: http://wiki.ros.org/urdf/Tutorials/Create your own urdf file

The URDF description of the IIS robot setup is spread across multiple packages, located in the `iis_hardware` stack. Arm and gripper descriptions are located in seperate packages (`lwr_description` and `schunk_description`), the `iis_robot` package brings all the components together. The modelling process started with the search for pre-existing URDF descriptions of the required robot components. A suitable description of the Schunk SDH gripper was taken from the `schunk_description` ROS package. A model of the KUKA LWR arm was found in the Github repository[6] of the *Robot Control and Pattern Recognition Group*[7]. The other parts of the model, namely the robot torso and the table had to be created. The descriptions are located in a seperate files within the `uibk_robot` package (`torso.xacro` and `table.xacro`).

The mesh files, used in the description of the robot torso have been exported from the V-Rep simulation scene. For the visual part, the mesh was taken as it is. For the collision model, the width of the original mesh was increased to provide some safety padding. Moreover a cylinder with a diameter of 40cm was added in the head area to ensure that the planner avoids accidental hits in this sensitive region. Figure3.3 shows the visual and the collidable part of the torso model. The table was modelled from primitive shapes. It was taken care that the size of the table can easily be adjusted, as can bee seen in Listing3.1. The collision model of the table is also slightly larger than the visual part to provide some safety margins.

[6] https://github.com/RCPRG-ros-pkg/lwr_robot/tree/hydro-devel/lwr_defs
[7] University of Warsaw (http://robotyka.ia.pw.edu.pl/twiki/bin/view/Main)

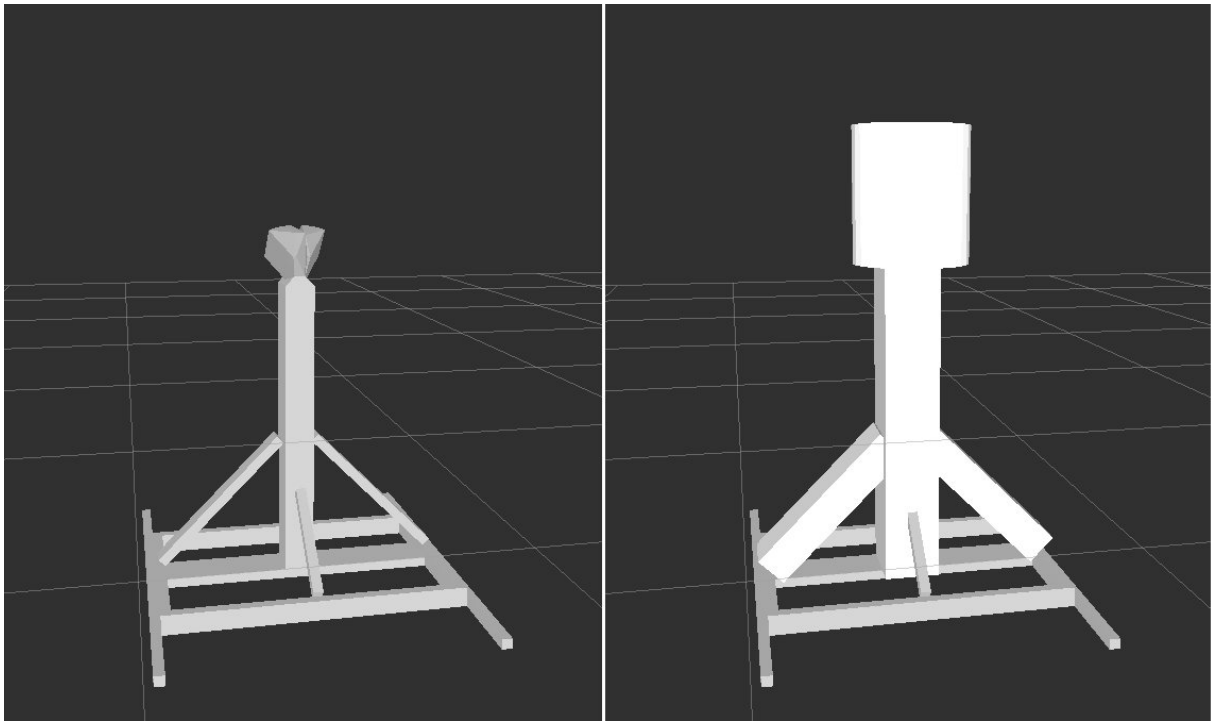

Figure 3.3: Left image shows visual part, right image the collidable part of the torso

Listing 3.1: XML snippet, inserting the table model into the URDF

```
<!-- draw the table relative to the origin -->
<xacro:model_table name="table"
        parent="world"
        length="2.22"
        width="0.8">
  <!-- Place the table relative to the world reference frame -->
  <origin xyz="-0.029 -0.3 0" />

</xacro:model_table>
```

When attaching the two grippers it showed that the offset between gripper wrist and last arm link was not correct. To correct that issue the file `sdh_with_connector.xacro` was created which simply places an additional ring between the last arm link and the gripper.

The file `iis_robot_table.xacro` draws all the pieces together. It describes the whole setup, consisting of the torso, two arms, two grippers and the table. The root element of the model hierarchy is a link called `world_link`. Torso, table and both arms are positioned relative to that root link. Changing the world reference frame could easily be achieved by shifting the root link to a new position. Figure3.4 shows a visualization of the URDF description.

3.5 Configuring the planning tools

The MoveIt configuration package is created, using the *MoveIt Setup Assistant*. Precondition is an existing URDF description of the robot setup which was created in the previous step. The setup process comprised of the following steps:

- **Computation of the self collision matrix**

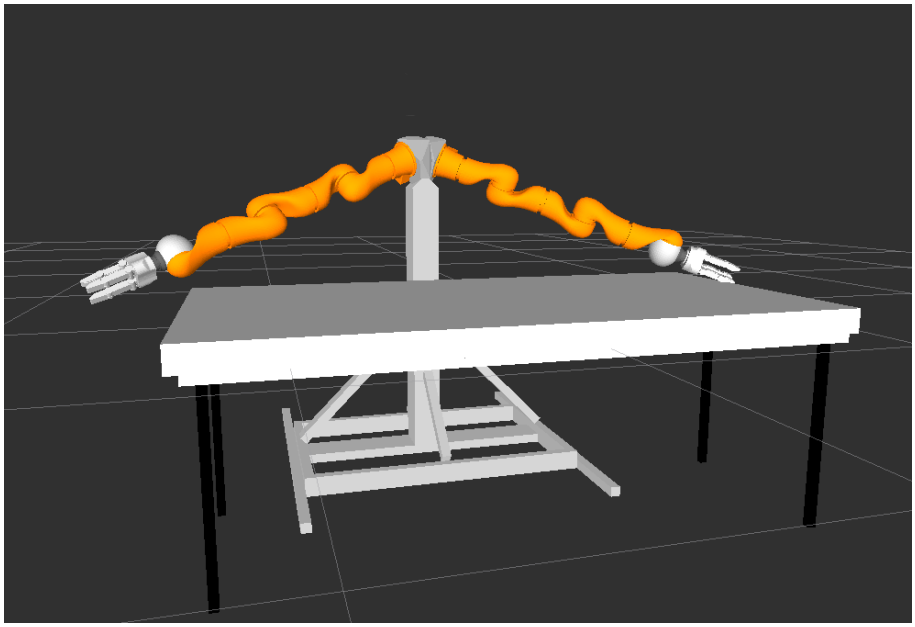

Figure 3.4: URDF description in RViz

The self collision matrix consists of pairs of robot links that can safely be excluded from collision checking. Neighbouring links for example are in permanent collision. Collisions between other links can never happen because they are simply too far apart. The Setup Assistant calculates a large number of different robot configurations and tracks for link pairs that are mostly always in collision and pairs that are never in collision. The self collision matrix can be adjusted manually if necessary. Excluding a large number of link pairs raises performance during motion planning because collision checking is an expensive process.

- **Defining the planning groups**

  Each planning request in MoveIt is done against one of the defined *planning groups*. A planning group is a group of links and joints within the model that can be seen as one logical component. Planning groups are defined for both arms and grippers. The configuration for the arms also requires the definition of the utilized IK solvers but currently only the default KDL solver is available. An additional planning group, called `both_arms` allows planning requests for both arms simultaneously.

- **Defining the end effectors**

  End effectors are the left and the right gripper. Each end effector has a name and consists of one of the predefined planning groups, a parent group and the parent link which is the last link in the kinematic chain of the parent group.

- **Generate the configuration files**

  After completing all the configuration steps the configuration package can be generated. Therefore a package name has to be specified which was set to `uibk_robot_moveit_config`. This package is a prototype, containing a large number of predefined configuration files. Some of those files have to be extended manually, adding additional information about available robot controllers and sensors.

Completing the last step results in a ROS package, containing necessary configuration and launch files for the given robot setup. The semantic robot description can be found in the `iis_robot.srdf` file. It contains previously defined parameters like the self-collision matrix, planning groups and end effectors. The configuration package can be tested, running the following command on the command line:

```
roslaunch uibk_robot_moveit_config demo.launch
```

This commands launches a `move_group` node, using the previously created configuration files and starts an instance of RViz with the motion planning plugin. There it is possible to switch between the planning groups, set start and target configurations, do planning requests and visualize the outcome. The setup can be modified by launching the Setup Assistant again, using the `setup_assistant.launch` file from this package.

3.6 Connecting MoveIt to the existing robot control interface

Now MoveIt is configured and ready to handle planning requests for the robot setup. But execution of the resulting trajectories is still impossible because of the missing connection between MoveIt and the involved robot hardware. Each component that is intended to be controlled by MoveIt needs to provide the *FollowJointTrajectory* action interface, as defined in the `control_msgs` package. This is a special kind of ROS interface that allows to send trajectories to robot components and monitor the execution status. As the existing control interface of the IIS robot components does not fulfil these requirements, an additional node is necessary that is capable of executing trajectories, using the existing infrastructure.

The planning outcome of MoveIt is a time parametrized trajectory, expressed by the *JointTrajectory* message type. This message consists of a set of waypoints. A waypoint is a joint configuration, described by the tuple (p, v, a, t) where $p \in \mathbb{R}^n$ are the positions, $v \in \mathbb{R}^n$ the velocities and $a \in \mathbb{R}^n$ the accelerations at time t and n is the number of involved joints. Those waypoints mark the important points along the path, the manipulator has to move. The controller needs to be able to translate this trajectory into a sequence of suitable motor commands. Just sending the joint positions within the waypoints to the robot would not suffice as a trajectory is also constrained in terms of *velocities* and *accelerations* over *time*. Therefore the controller needs to be able to interpolate between the subsequent waypoints and calculate intermediate joint positions based on the loop rate of the control cycle. This interpolation is a difficult task and the implementation would be far beyond the scope of this project. Luckily the required functionality is already available within the `ros_control` stack[8]. Those packages are created to integrate and control robot hardware in a generalized way, facilitating the usage of existing controllers on various different robots.

3.6.1 ROS control stack overview

Figure3.5 shows an architectural overview of the ROS control stack. The required infrastructure for using generic controllers is mainly provided by two components:

- **RobotHW**

  This is the abstract base class for robot hardware abstraction. It's main purpose is to send commands to the hardware and retrieve state data. The *RobotHW* also maintains a set of

[8] http://wiki.ros.org/ros_control

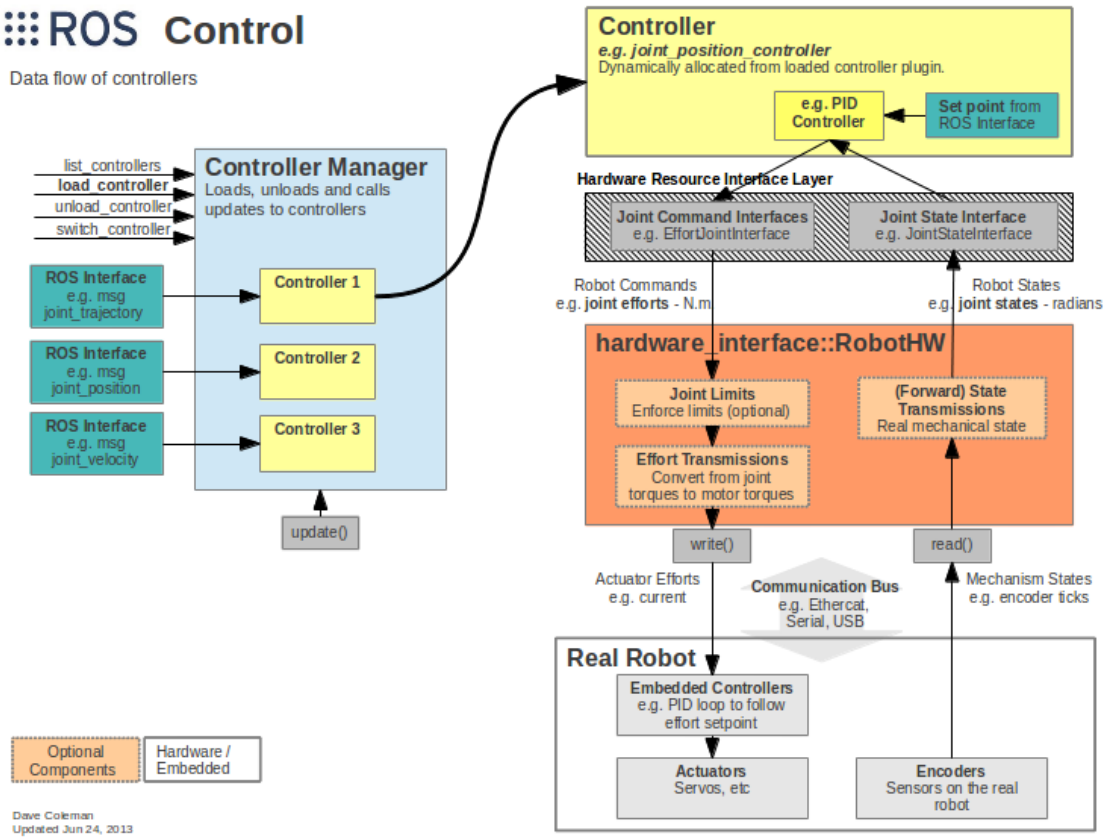

Figure 3.5: ROS control architecture

Image source: http://wiki.ros.org/ros_control

hardware ressource interfaces. Each generic controller needs a special kind of interface for controlling joints or read their current state. Concrete implementations have to register the required interfaces. For retrieving state data, a *JointStateInterface* is required. Controllers using that interface can query it for a specific joint and ask for the current state. Sending commands to the joints is done via subtypes of the of the abstract *JointCommandInterface*. The *JointPositionInterface* for example allows to send target positions to registered joints. Other implementations are *JointEffortInterface* and *JointVelocityInterface*. Which interfaces are provided depends on the way how the actual hardware is controlled.

- **ControllerManager**

The *ControllerManager* is responsible to maintain a set of controllers. It provides a ROS interface that allows to load, start and unload controllers and for switching between them. During the control cycle, the *RobotHW* is forced to read current state from the hardware. Then the *ControllerManager* is triggered to update the active controllers based on the current time step. The commanded values are then sent back to the concrete hardware. This control cycle is intended to run in real time when interacting directly with the hardware. Controllers are usually configured on the parameter server and loaded on demand, using the ROS interface of the *ControllerManager*. Custom controllers can be created by inheriting from the abstract *ControllerBase* class.

3.6.2 Designing the hardware adapter

The *hardware adapter* is an independent ROS node that acts as connection between MoveIt and the simulated or real hardware. It provides the necessary infrastructure for using generic controllers from the `ros_control` packages. Figure3.6 gives an overview about the hardware adapter architecture. The *UibkRobotHW* is a subclass of *RobotHW* and represents the connection to the robot. It maintains the complete state of all joints in the connected components. Joints are represented by the *Joint* datatype, consisting of a unique joint name, state parameters and a commanded target position. For controllers, the *UibkRobotHW* class provides a *JointStatesInterface* and a *JointPositionInterface*. Active controllers use those interfaces to access the current state and for commanding target positions.

The *UibkRobotHW* utilizes a set of *JointStateAdapters*, each one representing a connection to one specific robot component. During the control cycle the *JointStateAdapters* read current joint states from the appropriate topics and send commanded values back to the hardware. The *JointStateAdapters* are created during initialization, based on the configuration settings.

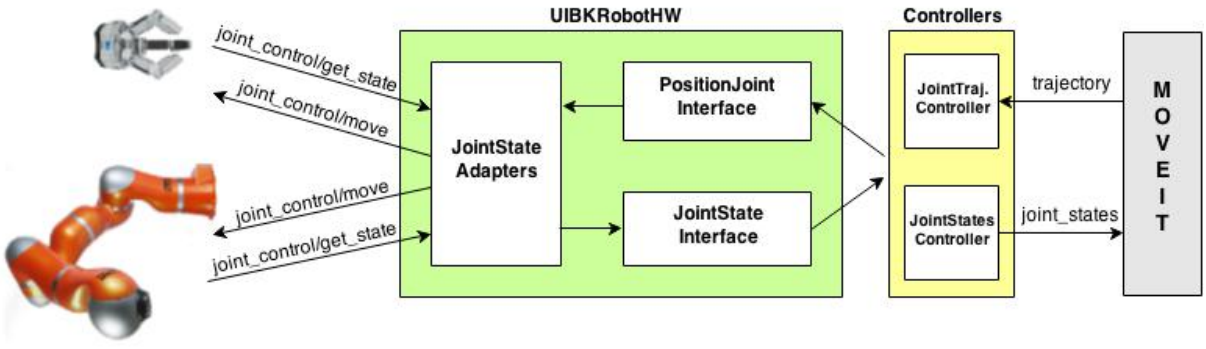

Figure 3.6: Hardware adapter architecture

After creation, each *JointStateAdapter* waits a certain amount of time for an initial joint states message. If it does not receive such a message before timeout, it will automatically shut down for safety reasons and report an error. This is very important as the *JointStateAdapter* imediately begins to send joint positions after initialization. The position to be sent is initially the currently known position, as long as no other values have been commanded by the controllers. Therefore the initial position always has to be known, otherwise dangerous and rapid robot motions could occur on startup. The adapter configuration happens via the parameter server. Available configuration parameters are:

- **`adapter_list`**

  Contains a list of all *JointStatesAdapters* that have to be created. For each adapter name mentioned in this list a detailed configuration is required. The subsequent parameters have to be configured for each single adapter.

- **`joint_state_topic`**

  The name of the topic where a specific adapter listens for joint states. The expected message type `sensor_msgs/JointStates`. This parameter is mandatory.

- **`readonly`**

  This parameter is optional. If true then the adapter will only listen for joint states but not publish commanded values.

- **`joint_command_topic`**

  The name of the topic the adapter should use to publish the commanded values to. The expected message type is `std_msgs/Float64MultiArray`. This parameter is mandatory if the adapter is not configured to be read only.

- **`joints`**

  A list of joint names that should be controlled by the adapter. The order in this list also determines the order of the values in the published message.

- **`joint_name_prefix`**

  This parameter can be used to be able to uniquely identify joints. For example both arms are using the same joint names (`arm_0_joint`, `arm_1_joint`,...). But in a URDF model, joint names have to be globally unique. Therefore the prefix can be used to prepend the original name to achieve uniqueness. A value of `right_` for example will lead to the joint names `right_arm_0_joint`, `right_arm_1_joint`,...).

An example configuration can be found in Listing3.2. As the topic names, used by the *JointStateAdapters* can be configured freely, the hardware adapter is able to interact with the simulator and the real robot as well, as both of them provide exactly the same ROS interface.

Listing 3.2: Adapter configuration for left arm and gripper

```
adapter_list:
  - left_arm
  - left_sdh

left_arm:
    joint_state_topic: "left_arm/joint_control/get_state"
    joint_command_topic: "left_arm/joint_control/move"
    joint_name_prefix: "left_"
    joints:
      - arm_0_joint
      - arm_1_joint
      - arm_2_joint
      - arm_3_joint
      - arm_4_joint
      - arm_5_joint
      - arm_6_joint

left_sdh:
    joint_state_topic: "left_sdh/joint_control/get_state"
    joint_command_topic: "left_sdh/joint_control/move"
    joint_name_prefix: "left_sdh_"
    joints:
        - knuckle_joint
        - finger_12_joint
        - finger_13_joint
        - thumb_2_joint
        - thumb_3_joint
        - finger_22_joint
        - finger_23_joint
```

During hardware adapter startup an instance of the *UibkRobotHW* and *ControllerManager* is created from configuration. As the control loop must not be interrupted by ROS callback functions, it is launched in a separate thread after completing initialization, using a loop rate of 100hz. On each iteration the exact time since the last step is calculated. The *ControllerManager* then updates all registered controllers based on current state and time since last iteration. After handling the controllers, the *JointStatesAdapters* are forced to send the commanded values to the hardware. The ROS callback functions are handled in the original thread.

After starting the hardware adapter, the required controllers have to be loaded and started. This is usually done, by using the corresponding ROS services, provided by a running *ControllerManager* instance, but the `controller_manager` package contains a tool named `spawner` that simplifies the starting process. The configuration of each controller has to reside on the parameter server. Utilized controllers are the *JointTrajectoryController* and the *JointStateController*. The *JointTrajectoryController* provides the required *FollowJointTrajectory* action interface for a group of joints. For each robot component a separate *JointTrajectoryController* is used. The *JointStateController* publishes the collected states of all robot joints at once to the `joint_states` topic, which is used by MoveIt to monitor the robot configuration.

3.6.3 Launching the hardware adapter

The file `hardware_adapter.launch`, located in the `uibk_moveit_adapter` package was created to handle the necessary configuration parameter upload and launch the hardware adapter node for the simulator and the real robot as well. As can be seen in Listing3.2, the configured

topic names for the *JointStateAdapters* are defined, using *relative* graph resource names (i.e. without trailing slash). The required instance is than accessed by simply shifting the node into `simulation` or `real` namespace. This is realized, specifying the `config_name` parameter of the launch file. The command line statement

```
roslaunch uibk_moveit_adapter hardware_adapter.launch config_name:=simulation
```

launches the hardware adapter in `simulation` namespace.

After launching the hardware adapter node, the required controllers are loaded and started. The node provides feedback information about the state and errors during startup process on the console output. It is crucial that the joint state topics of simulator or real robot are available before launching the hardware adapter node, otherwise it will not be able to work because of missing initial joint states. After successful startup, the additional controller topics can be found in the corresponding namespace.

3.7 Adjusting the MoveIt configuration

The prerequisites are now made for connecting MoveIt to the (simulated or real) robot. The last remaining step is to adjust the MoveIt configuration for establishing a connection to the *FollowJointTrajectory* controllers. Therefore the configuration file `controllers.yaml` was created in the `uibk_robot_moveit_config` package. It contains a list, describing the available controllers. Each controller description contains the name of the controller, it's action namespace, the controller type and a list, containing the names of the controlled joints.

For starting MoveIt, the launch file `moveit_planning_execution.launch` was created. This file is intended to launch a `move_group` node either for simulated or real robot, together with the corresponding hardware adapter. The namespace can be selected using a boolean parameter named `simulation`. The statement

```
roslaunch uibk_robot_moveit_config moveit_planning_execution.launch simulation:=false
```

launches a MoveIt configuration, connecting the `move_group` instance to the real robot. The launch file also starts RViz configured with the motion planning plugin. This is used for visualizing the planned trajectories and can also be used to test the connection to the robot by making planning requests and execute the resulting trajectories.

Chapter 4

Pick and place

The implementation of a benchmark pick and place task, executable on simulator and real robot as well, states one of the objective targets of this project. The implementation heavily uses major parts of MoveIt's planning functionality. The overview at the beginning of this chapter gives some general information about pick and place tasks, explaining the process step by step and describing the single stages that have to be performed. The second part focuses on the pick and place functionality of MoveIt and discusses the involved action servers, topics and messages. The third section explains the implementation of the benchmark pick and place task in greater detail and demonstrates how that functionality of MoveIt was used. The last section describes some observations that have been made during the implementation process.

4.1 Overview

A pick and place task is the process of grasping an object, lifting it and dropping it somewhere else. Humans do that permanently, without even think about it. But when teaching a robot to perform a pick and place action, it shows how difficult and complex this task is and how much planning is involved to achieve the desired result. The planner requires exact knowledge about the robot and it's environment, including the objects to grasp and possible obstacles. Accidental collisions have to be avoided but other collisions are mandatory when the robot has to get in contact with the world. The gripper definitely collides with the object to pick, but only during grasping and holding. Therefore there has to be a mechanism to explicitly tell the planner that specific collisions are allowed during particular stages of the operation. Moreover, after grasping an object it has to be considered as an additional part of the robot during subsequent planning requests. That means that it must be attached to the manipulator temporarily and removed, after releasing the object. As long as the object is part of the robot, it possibly increases the size of the end effector.

Stationary objects usually stand or lie on a surface, called the *support surface*. During the interaction with an object, possible collisions with the support surface have to be taken into account. It is also possible that the whole process underlies additional constraints, so called *path constraints*. This type of constraint has to be enforced along the whole path, the grasped object takes during the operation. For example when carrying a glass, filled with liquid it has to stay in an upright position, otherwise the liquid is lost. That means, the glass has to be held in a specific orientation during the whole task. This can be described as *orientation constraint* which is a special type of path constraint. The planning solution has to provide mechanisms to define end enforce such types of *path constraints*.

Pick and place tasks can be split into two independent phases, each one composed from a number of trajectory stages:

- **Pickup phase**

  This phase starts at an arbitrary robot configuration. In the first stage, the manipulator has to be brought into a position closed to the object to grasp, but in a distance that allows the gripper to open safely without touching the object. This position is called the *pre-grasp pose*. The next stage is to set the gripper into *pre-grasp posture*. That means, bring it's fingers into an open configuration that allows to completely enclose the object (or at least that part that is used to clutch it) after approaching towards the final *pose*. How this configuration looks like depends on the shape of the object to grasp and the structure of the gripper. Further on, the gripper has to be moved towards the *grasp pose* using the correct approach direction. This is the place where the robot gets in contact with the object. The gripper moves it's fingers into the *grasp posture* - a configuration that encloses the object and applies as much force as necessary to be able to take and hold it. The resulting collisions between the gripper links and the object have to be ignored. At that point the object has to be attached to the gripper and further on treated like an additional link of the robot although still being in collision with the support surface. The pickup phase completes after lifting the object along the retreat direction. The object is now part of the robot and no collisions should be tolerated any more. Figure 4.1 shows the stages of the pickup phase.

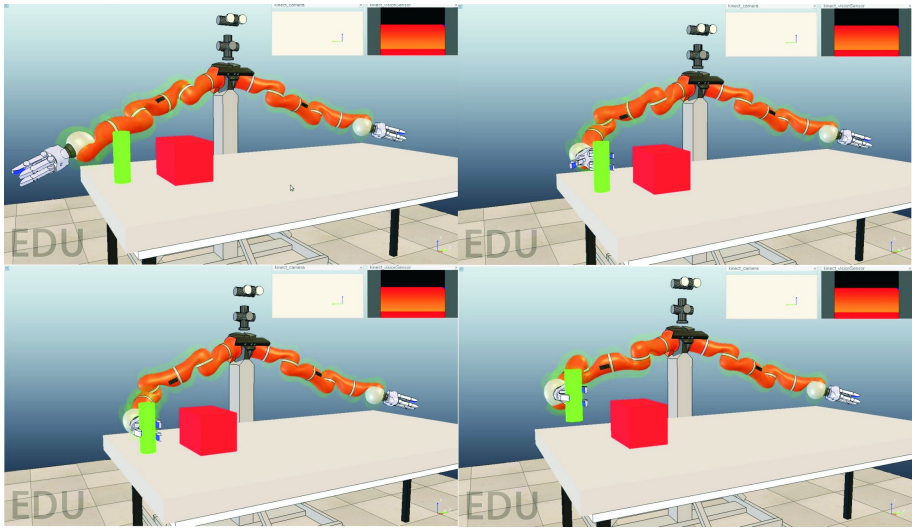

Figure 4.1: Stages of the pickup phase in the simulator

- **Placement phase**

  The placement phase starts after a successful pickup. The grasped object is enclosed by the gripper and considered to be part of the robot. Now the manipulator moves towards the *pre-place location* - the place where the final approach towards the goal starts. The easy way is to just drop the object at that point. In that case, the placement phase completes after bringing the gripper fingers into the *post-place posture* (opening it) and detaching the object from the robot.

If the object should to be placed carefully, the gripper has to approach from the *pre-place location* towards the final *place location* along the specified approach direction. Here has to be considered that the object will get into contact with the support surface again when it's final position is reached. At that stage the gripper can open and the object has to be detached from the robot. From that point, the object needs to be treated as obstacle again which means the planner is forced to avoid collisions with it. The placement phase completes after the manipulator has moved away from the object along the specified retreat direction. The stages of the placement phase can be seen in Figure 4.2.

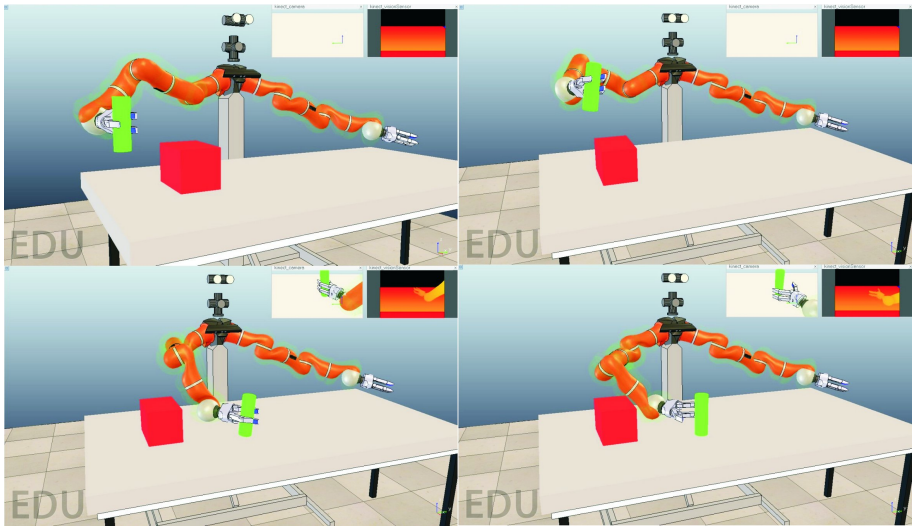

Figure 4.2: Stages of the placement phase in the simulator

Both phases of the task are only considered to be complete if each single stage has successfully been executed. The necessary planning parameters like pre-grasp and grasp pose, gripper postures and approach and retreat directions are usually provided by a *grasp planner*. This is an additional node within the planning pipeline that identifies objects in the environment of the robot, usually based on 3D sensor data and calculates the necessary grasp parameters. Explaining the functionality of grasp planners is far beyond the scope of this project though the section about implementing the reference task discusses the used parameters in greater detail and shows what would be usually delivered by the grasp planner.

4.2 Pick and place tasks in MoveIt

During the various stages of the pickup and placement phases a lot of motion planning is required. Each single stage requires to plan a trajectory that is free of accidental collisions while respecting the limits of the robot and enforcing possible additional constraints. Each phase can only be considered executable if a valid motion plan for each single stage exists. Planning all the subsequent stages one after the other would be a cumbersome task. Therefore MoveIt provides a set of messages and action servers that greatly simplify those planning tasks.

The *PickupAction* server handles the planning and optionally execution of all required stages during the pickup phase at once. Pickup requests are done, using a *PickupAction* client to send *PickupGoal* messages to the server. The request message is composed of all the parameters that are necessary to completely describe the planning problem. Picking up an object can always be done in several ways. Depending on the shape of the object there hardly always exist a lot

of possible poses where the gripper can safely approach and grasp. Therefore MoveIt allows to provide a set of possible grasp definitions for a pickup request. There is also a quality parameter that can be used to tell the planner, how 'good' a specific grasp is compared to other ones. MoveIt can then favour grasps with higher quality if several valid solutions are found by the planner. Providing a number of different grasps increases the probability for the request to be successful. The grasp definition is composed of the pre- and post grasp postures for the gripper, the final grasp pose and the approach and retreat vectors. As MoveIt needs to know which object should be picked within the planning scene, it is also necessary to include the ID of the object to grasp.

The *PlaceAction* server is responsible for planning the whole placement phase. Therefore same concept is used as above - a set of possible place locations can be provided to increase the probability of a successful planning attempt. The parameters that are used to describe a place location include the post-place posture of the gripper along with it's target pose, and vectors, describing the pre-place approach and the post-place retreat.

Pickup and placement requests as well require a number of additional parameters that are explained in greater detail within the section about benchmark task implementation.

Objects contained in the task environment have to be brought to MoveIt's attention. This can be done either by manually adding them to the planning scene, using the corresponding topics or by configuring MoveIt to be aware of sensor data. As the integration of depth information from the Kinect camera did not work stable during the evaluation phase, that possibility is not covered within this thesis and all involved objects were added manually.

Pickup- and place action servers provide the ability to choose whether to execute motion plans immediately or to do just the planning and return the outcome. In that case the execution has to be handled manually later on. The first method is much more comfortable as MoveIt executes the trajectories and also handles additional requirements like attaching and detaching the grasped object in time. The drawback is that also possibly weird trajectories get executed immediately as they might be valid solutions for the given planning problem though they are obviously unsuitable. So an additional safety mechanism needs to be introduced that allows to interrupt the execution when facing any problems. The second method allows to visualize the resulting trajectories and then decide whether to execute them or not. But then each single trajectory stage has to be executed manually which also includes attaching or detaching objects to the manipulator. The advantage of this method is the clean separation between planning and execution, allowing maximum control over the execution flow. Therefore this method was favoured during benchmark task implementation.

4.3 Implementation of the benchmark task

This section describes the implementation of the benchmark pick and place task in greater detail. The source code can be found within the 'uibk_moveit_tests' package. The workspace is the table in front of the robot covered with a 9cm thick foam mat. This mat will be declared as the support surface later on. The object to grasp is a cylinder with 4cm radius and a height of 25cm. This corresponds to the size of a usual SIGG bottle which can be used to run the benchmark task in the real world. The cylinder is located on a fixed, known position. There is also a box shaped object located within the workspace that acts as an additional obstacle. The goal of the task is to pick the cylinder up, using the right arm of the robot and place it at the goal location without colliding with the obstacle or other parts within the robot's environment. The implementation makes use of MoveIt's pick and place functionality. The necessary steps

are explained in the following subsections.

4.3.1 Creating the environment

As MoveIt is currently not configured to use sensor data, it has to be notified about the task environment. The table and the surface mat are already part of the URDF description but the cylinder and the obstacle have to be added to the task environment. This is done utilizing the `planning_interface::PlanningSceneInterface` class that provides functionality to manipulate the current planning scene. The same effects can be achieved by manually publishing to the '/collision_object' topic but using the convenience class saves a lot of boilerplate code. The *CollisionObject* type is used to describe those objects. Both of them are primitive shapes. Necessary parameters are the shape type, dimensions and the target pose. Additionally each *CollisionObject* needs a unique ID which is used to identify the shape within the planning scene. After adding the *CollisionObjects* to the collision world, they are visualized in RViz. The image in Figure4.3 shows the task environment after adding the *CollisionObjects*.

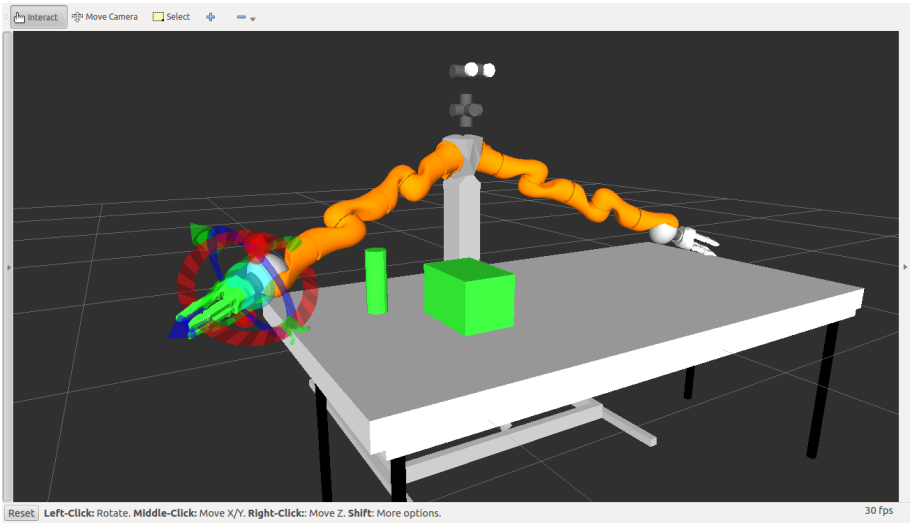

Figure 4.3: Planning scene after inserting the collision objects

4.3.2 Generating possible grasps

Before calling the pickup action server it is necessary to generate a set of possible grasps for the object to pick. This information is usually provided by a grasp planner. The sample task encapsulates the grasp planner functionality within the *generateGrasps* method. This method takes the current pose of the cylinder as input parameter and calculates 10 possible grasp poses along a semi circle around the cylinder location. Pre-grasp and grasp postures are joint trajectories for the gripper, containing just one trajectory point stating the target configuration for the gripper in the opened respectively the closed state. The pre-grasp approach and the post-grasp retreat are defined as *GripperTranslations*. This is a special message type that describes the direct gripper movement from one position towards a target. The direction is defined as a three dimensional vector. The length of the translation can be set in a flexible way by specifying a desired distance and a minimum distance. Experiments showed that the success rate is much better if the grasp parameters allow some flexibility to the planners. The approach vector depends on the grasp pose and points along the z-axis of the end effector frame towards the object. The desired distance between pre-grasp and grasp pose is set to 20cm while the minimum

distance is 10cm. The gripper retreat vector points up, along the z-axis of the world. Desired and maximum distances are also set to 20cm respectively 10cm. As no one of the generated grasps should be favoured among the others, the grasp_quality parameter of each grasp is set to 1. This means that they are equal in quality. The last necessary parameter is a unique identifier for each grasp. As usually a lot of different grasps are provided for each pickup request, this ID can be used to identify, which one was finally used to solve the planning problem.

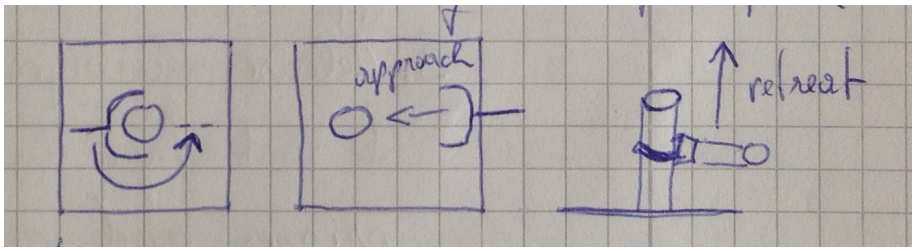

Figure 4.4: Grasp poses, gripper approach and retreat

4.3.3 Planning and executing the pickup

After generating the set of grasps, the planning request can be sent to the pickup action server. As there is a lot of boilerplate code necessary, a reusable helper class was created that can be used to simplify all kinds of planning requests. The utility class is called *PlanningHelper* and can be found within the 'uibk_planning_node' package. A call to the pickup action server is done by using the 'plan_pick' method of the helper class. This method takes a set of possible grasps and the ID of the object to pick as parameters. The outcome is a pointer to an instance of *PlanningResult*, a structure that contains all the necessary information about a planning attempt. Success or failure is indicated by the 'status' parameter. On success, the parameter 'trajectory_stages' holds a vector, containing the resulting trajectory stages. The actual call to the pickup action server is done by defining the *PickupGoal*, using the predefined grasps. Additional parameters are the ID of the object to pick, the name of the chosen planning group and the name of the link within the robot model that acts as the support surface. Optional parameters are among others the ID of the planner to use and the maximum allowed planning time. The parameter 'plan_only' within the planning options is set to true to avoid the immediate execution of the planned trajectories. This allows a visual verification of the planning outcome before execution. The resulting robot path is shown in RViz. If the solution is satisfying it can be executed, passing the PlanningResult to the corresponding method of the PlanningHelper class. This method uses the '/execute_kinematic_path' service provided by the 'move_group' node. The service sends a given trajectory to the responsible controller and provides feedback information about the execution status. The 'PlanningHelper' also takes care to attach the picked object to the gripper after the grasp stage. The pickup phase completes after successful execution of all trajectory stages.

4.3.4 Planning and executing the placement

The placement phase is planned and executed in a somehow similar manor. The cylinder has to be placed on a specific location within the workspace in an upright position - the rotation around the z-axis doesn't matter. Therefore a set of possible place poses is generated in 20 different orientations around the z-axis. This again gives some freedom to the planner as it can choose which one to use (TODO: provide image that shows the principle). The final approach towards the goal location is specified as *GripperTranslation*. The direction vector points down

along the z-axis of the world. Desired and minimum distances are set to 20cm respectively 10cm, allowing some flexibility during that stage. As post-place posture for the gripper, the same configuration is used as for the pre-grasp posture. The last required parameter for a place location is the *GripperTranslation* that describes the retreat after releasing the object at the goal location. The direction depends on the chosen orientation and points towards the negative z-axis of the gripper reference frame.

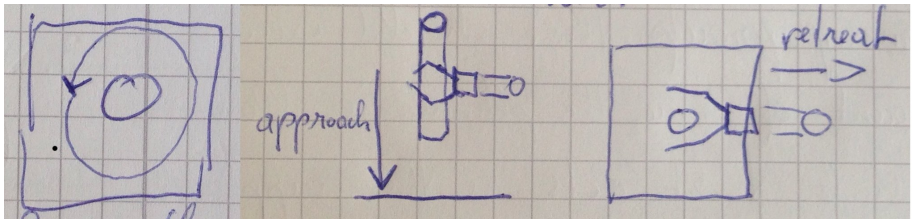

Figure 4.5: Place locations, pre-place approach and post-place retreat

The call to the place action server is again handled by the formerly mentioned planning helper class. The 'plan_place' method takes a vector of the predefined place locations and the ID of the object to place as input parameters. The actual call to the place action server is done, defining a *PlaceGoal* message. The most important parameters are the ID of the object to place, the name of the planning group and the set of possible place locations. Optionally can be specified which planner to choose and also the maximum allowed planning time. The method returns the result of the planning request. On success, the resulting trajectories are again visualized in RViz and can then be executed the same way as during pickup phase. After the execution, the task is considered to be complete and the picked object is detached from the gripper and again part of the collision world.

4.4 Executing the benchmark task

The benchmark task is designed to run on the simulator and the real robot as well. The sample program only depends on a running 'move_group' instance. The easiest way to run the sample is to start MoveIt in demo mode (demo.launch) and then execute the sample code, using.

```
rosrun uibk_moveit_tests sample_pick_place
```

This demonstrates the functionality but only executes the trajectories on the fake controllers provided by MoveIt. If the sample should be executed on the simulator or the real robot, the corresponding namespace has to be specified before running the code. This can be done by setting the 'ROS_NAMESPACE' environment variable to either 'simulation' or 'real', within the terminal. For example

```
export ROS_NAMESPACE=simulation
rosrun uibk_moveit_tests sample_pick_place
```

runs the benchmark task on the simulator. Of course it is necessary to start the simulator and launch the corresponding MoveIt instance before doing that. After creating the environment and adding the objects to the planning scene the pickup phase gets planned. The outcome can be seen in RViz and the program will ask if the resulting trajectory is ok and should be executed. A negative answer will force the program to replan the pickup and ask again, otherwise the trajectory gets executed. After successful execution, the place phase gets planned. The resulting plan also is the visualized as well and execution needs confirmation again. The program exits after successful completing the placement phase.

4.5 Observations

This section gives an overview about the most important observations that have been made during the implementation of the sample task:

- It is very important to provide some degree of freedom to the planner at various points. Major points are the amount of different grasps or place locations and the allowed range within the defined gripper translations. Very strictly defined planning requests are very likely to fail whereas requests with a higher degree of flexibility drastically raise the overall success rate.
- Working with path constraints drastically drops the success rate because the high complexity of the planning problem. Enforcing path constraints requires much more allowed planning time and a very fast IK solver because of the large number of necessary IK requests. Maybe a faster IK solution than the one available could help to solve the problem but that is not guaranteed. Therefore there are no path constraints used within the sample task.
- Planning requests often fail due some MoveIt-internal issues and not because planning is not possible at all. Therefore it is necessary to repeat failed requests because it is very likely that planning succeeds on subsequent attempts. In the sample task implementation, the planning requests are done within a loop. If a request fails, it gets repeated. A successful request breaks the loop and continues the execution flow.
- Resulting trajectories should always be visualized in RViz before confirming the execution. The calculated motion plans might be valid but sometimes they are a bit confused and therefore unsuitable. Moreover, the planner can only take into account what it knows about the robot's environment. Especially in the robot lab there is equipment mounted in the area above the robot, that is not taken into account during planning. Therefore the visual validation is necessary to avoid damages on the robot and it's environment.

Chapter 5

Conclusions

Current status, open questions, possible further work

Bibliography

Open motion planning library: A primer. `http://ompl.kavrakilab.org/OMPL_Primer.pdf`. Published by Kavraki Lab, Rice University.

Samuel R Buss. Introduction to inverse kinematics with jacobian transpose, pseudoinverse and damped least squares methods. *IEEE Journal of Robotics and Automation*, 17:1–19, 2004.

Howie M Choset. *Principles of robot motion: theory, algorithms, and implementation.* MIT press, 2005.

Marc Freese, Surya Singh, Fumio Ozaki, and Nobuto Matsuhira. Virtual robot experimentation platform v-rep: A versatile 3d robot simulator.

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, 2009.

Schunk. Sdh2 documentation overview, 2010.

Jacob T Schwartz and Micha Sharir. On the "piano movers" problem. ii. general techniques for computing topological properties of real algebraic manifolds. *Advances in applied Mathematics*, 4(3):298–351, 1983.

Appendix A

Simulator documentation

This chapter provides a documentation for the simulation solution that was developed during this project. Some of the steps, described within this documentation require basic knowledge about using the V-Rep scene editor.

A.1 Installation and startup

The simulator plugin code as well as the scene and model files is located in the `iis_simulation` package which is part of the `iis_robot_sw` repository. The plugin code was written in C++ and needs to be compiled, using the catkin build system which is part of the ROS distribution. Detailed information about how to install and build the source code can be found in the top-level README file within the repository. The compilation process generates a library file, named `libv_RepExtSimulatorPlugin.so`. To be able to work, that library file needs to be copied into the working directory of the V-Rep installation. This happens automatically when using the top-level makefile of the `iis_robot_sw` repository. The plugin library is automatically detected and loaded during V-Rep startup. But as it's functionality depends on ROS it will be unloaded immediately if no running roscore can be detected. The status of recognized plugins can be seen on the console output.

The package contains the reference scene file, called `scenes/model_assembled.ttt` which can be started, using the `robot_scene.launch` file. The command

```
roslaunch iis_simulation robot_scene.launch scene:=MY_CUSTOM_SCENE.ttt
```

starts an instance of V-Rep, loads the scene file provided with the (optional) `scene` parameter and starts the simulation. If no scene file is provided, the default scene is loaded which is the provided reference scene. Scene files have to reside in the `scenes` folder within the package, otherwise the launch file is not able to locate them.

A.2 Modifying the simulation scene

The reference scene contains a fully functional replication of the IIS lab robot setup, composed from two arms, two grippers and a Kinect camera. It is recommended to start with the setup within this file to create alternative scenes and store modified versions in different files. The base of each arm is marked by a dummy element at root level of the scene hierarchy, i.e. `dummy_frame_left_arm` and `dummy_frame_right_arm`. Modifying the mounting of the arms is done by changing the alignment of those dummy elements. Currently they are placed relative

to the origin of the world reference frame. Parts of the scene can safely be deleted without breaking the functionality. To delete a gripper, it's base element has to be located in the scene hierarchy. This is done by expanding the model tree of the corresponding arm until the gripper base can be selected and removed. Each other part contained within the scene can be removed or rearranged in the same way (table, torso, Kinect camera).

The dummy element called `ref_frame_origin` states the origin of the reference frame used by the arm controllers and is indicated by a slightly green shimmering sphere near the top left corner of the table. Cartesian positions received from and sent to the arms are interpreted relative to that reference frame. The origin can be changed easily by just modifying the position and orientation of that dummy element. If it is removed than all positions are interpreted to be relative to the absolute world reference frame (which is currently at the same location).

A.3 Modifying custom developer data tags

Important parts within the scene hierarchy have to be tagged with custom developer data tags, as explained in Section2.5.2.

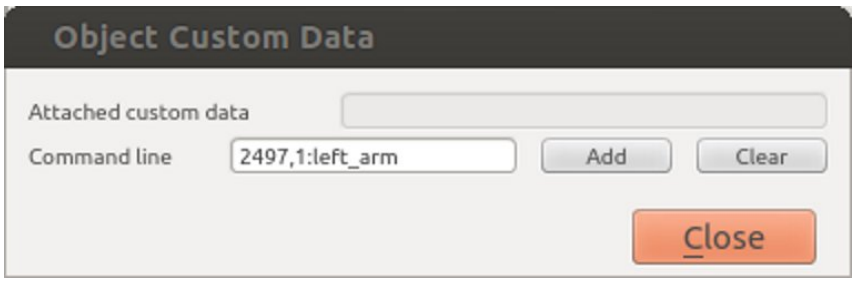

Figure A.1: Edit custom developer data dialog

To edit those data segments, the corresponding scene object needs to be selected in the scene hierarchy. After clicking the option *View/edit custom data* within the `common` section of the *Scene Object Properties* dialog, a new dialog box appears that allows to attach and remove custom data segments. Existing segment have to be removed prior to attaching a new value. This is done by entering the header ID (2497) into the text box and clicking the `clear` button. That causes all data segments, identified by the given header ID to be deleted on the selected scene object. After that, the new data segment can be attached by entering the proper values into the text box and clicking the `add` button. The values, entered in the dialog, shown in FigureA.1 identify a scene object to be the model base of a *LWRArmComponent* with name `left_arm`.

A.4 ROS interface

The ROS interface for the simulated robot components is composed from a set of inbound and outbound ROS topics, that can be used to send commands or to retrieve state data. The used topic names follow the IIS lab internal naming conventions. Each topic name has the structure

`[namespace]/[componentname]/[controltype]/[name]`

The namespace is necessary to distinguish between the topics of simulator and real robot. All simulator related topics reside in the `simulation` namespace. The component name identifies a specific instance within that namespace, e.g. `left_arm` or `right_sdh`. This is useful as

often more than one instance of the same robot component is used but all of them are utilizing the same topic names. The control type groups sets of topics into different categories. Possible values are `joint_control`, `cartesian_control`, `sensoring` and `settings`. Finally, the chosen name is used to identify the specific topic within the category. The topic `/simulation/left_arm/joint_control/move` states the topic named `move` in category `joint_control` for a simulated component, named `left_arm`.

The topics are provided by the registered *ComponentControllers* and are dynamically composed, based on the component name. The overall namespace is declared within the `initialize()` method of the *ROSServer* class. The component name has to be provided as value segment of the custom developer data tag that identifies the model base of a specific component within the scene. The control type and name segments of the topics are defined within the `initialize()` methods of the concrete controller instances. Some of the provided topics are only fake implementations as the corresponding functionality cannot be implemented on the simulator. Therefore they are just added for consistency reasons, but only fake data is published. These include all temperature and impedance related topics. When messages are sent to one of those fake topics, a corresponding warning will be displayed on the console.

A.4.1 Arm control topics

The topics, provided by the *LWRArmController* follow the specification of the *KUKIE* control interface that gets utilized on the real robot arms in the IIS lab. The only exception is the `sensoring/get_collision_state` topic that is unique to the simulator and allows to read the current collision status of the underlying arm. A complete list of the topics, available on the simulator can be found in TableA.1. Controller related errors are published to the `sensoring/error` topic but they are displayed on the console output of the simulator as well. Joint names, used by the `joint_control/get_state` topic are defined in the value segments of the custom developer data tags, attached to the arm joints.

Topic	Message type	Direction
cartesian_control/get_impedance	iis_kukie/CartesianImpedance	outbound
cartesian_control/get_pose	geometry_msgs/Pose	outbound
cartesian_control/move	geometry_msgs/Pose	inbound
cartesian_control/set_impedance	iis_kukie/CartesianImpedance	inbound
cartesian_control/set_velocity_limit	std_msgs/Float32	inbound
joint_control/get_impedance	iis_kukie/FriJointImpedance	outbound
joint_control/get_state	sensor_msgs/JointState	outbound
joint_control/move	std_msgs/Float64MultiArray	inbound
joint_control/set_impedance	iis_kukie/FriJointImpedance	inbound
joint_control/set_velocity_limit	std_msgs/Float32	inbound
sensoring/cartesian_wrench	geometry_msgs/Wrench	outbound
sensoring/error	iis_kukie/KukieError	outbound
sensoring/get_collision_state	std_msgs/Int32	outbound
sensoring/state	std_msgs/Int32MultiArray	outbound
sensoring/temperature	std_msgs/Float32MultiArray	outbound
settings/get_command_state	std_msgs/Float64MultiArray	outbound
settings/switch_mode	std_msgs/Int32	inbound

Table A.1: Available topics of *LWRArmController*

A.4.2 Hand control topics

The hand related topics are listed in TableA.2. Joint names, used by the `joint_control/get_state` topic are defined in the value segments of the custom developer data tags, attached to the hand joints. Setting motor currents influences the effort of the joint motors. The provided values are interpreted as percentages, e.g. a value of 0.5 will reduce the effort setting of the corresponding joint motor to the half of the possible maximum value. The currents can be defined for the `move` and `gripHand` topics separately.

Topic	Message type	Direction
`joint_control/get_state`	`sensor_msgs/JointState`	outbound
`joint_control/move`	`std_msgs/Float64MultiArray`	inbound
`joint_control/gripHand`	`iis_schunk_hardware/GripCmd`	inbound
`sensoring/temperature`	`std_msgs/Float64MultiArray`	outbound
`settings/get_motor_current`	`iis_schunk_hardware/MotorCurrentInfo`	outbound
`settings/set_motor_current`	`iis_schunk_hardware/MotorCurrent`	inbound

Table A.2: Available topics of *SchunkHandController*

A.4.3 Kinect camera settings

The Kinect camera model is not directly controlled via the simulator plugin as it contains no moveable parts that have to be controlled from code. The RBB and depth images are published, using the built in ROS functionality of V-Rep. The code listing in FigureA.1 shows the LUA script that is associated to the Kinect camera model within the reference scene.

Listing A.1: LUA script, attached to Kinect model

```
if (simGetScriptExecutionCount()==0) then
  depthCam=simGetObjectHandle('kinect_visionSensor')
  depthView=simFloatingViewAdd(0.9,0.9,0.2,0.2,0)
  simAdjustView(depthView,depthCam,64)

  -- publish depth image
  cmd=simros_strmcmd_get_depth_sensor_data
  topic='/simulation/kinect1/sensoring/depth_image'
  simExtROS_enablePublisher(topic,1,cmd,depthCam,-1,'')
  -- publish rgba image
  cmd=simros_strmcmd_get_vision_sensor_image
  topic='/simulation/kinect1/sensoring/rgb_image'
  simExtROS_enablePublisher(topic,1,cmd,depthCam,-1,'')
end

simHandleChildScript(sim_handle_all_except_explicit)
```

The script determines the object handle of the Kinect vision sensor and enables two ROS publishers for the captured RGB and depth images. Topic names can be changed in that script and the publishers can be dissabled by simply commenting the corresponding lines of code. The script also assigns the sensor data to an additional view within the simulation scene. This is helpful because it visualizes, the camera images during simulation. The correct transformation of the Kinect vision sensor can be obtained by expanding the model tree of the camera model and selecting the vision sensor, called `kinect_visionSensor`. The correct transformation can be read within the *position/orientation* settings of that scene object.