der Bundeswehr
Universität München

Entwicklung der Steuerung eines Roboterarms mit ROS

Bachelorarbeit

zur Erlangung des akademischen Grades

Bachelor of Engineering (B. Eng.)

Christian Waldner

Tag der Abgabe: 26. Juni 2018

Universität der Bundeswehr München
Fakultät für Elektrotechnik und Technische Informatik
Institut 4

Aufgabensteller und Prüfer: Prof. Dr. Ferdinand Englberger

Neubiberg, Juni 2018

Erklärung

gemäß Beschluss des Prüfungsausschusses für die Fachhochschulstudiengänge der UniBwM vom 25.03.2010

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, insbesondere keine anderen als die angegebenen Informationen.

Neubiberg, den 26. Juni 2018

CHRISTIAN WALDNER

Erklärung

gemäß Beschluss des Prüfungsausschusses für die Fachhochschulstudiengänge der UniBwM vom 25.03.2010

Der Speicherung meiner Bachelorarbeit zum Zweck der Plagiatsprüfung stimme ich zu. Ich versichere, dass die elektronische Version mit der gedruckten Version inhaltlich übereinstimmt.

Neubiberg, den 26. Juni 2018

CHRISTIAN WALDNER

Abstract

Ziel der vorliegenden Bachelorarbeit ist es, eine Steuerung für einen Roboterarm unter der Verwendung des Robot Operating Systems zu entwickeln.

Für die Realisierung wurde das Motion Planning Framework MoveIt! genutzt. Dieses Framework ermöglicht eine vollständige Implementierung einer Steuerung und hält bereits notwendige Bausteine wie Bewegungsplanung, Kollisionsvermeidung oder Kinematikberechnungen bereit. Neben der in MoveIt! bereits implementierten Komponenten mussten zusätzlich Bausteine wie Controller und Action Services eingerichtet werden. Die in dieser Arbeit gewonnenen Erkenntnisse dienen als Grundlage für weitere Entwicklungen unter der Verwendung von MoveIt! und ROS.

Die Steuerung soll einerseits die Bewegungsplanung eines selbst konfigurierten Roboterarms sowie eine kollisionsfreie Ausführung eines berechneten Pfades ermöglichen.

Dazu werden anfangs die wichtigsten Komponenten ausführlich vorgestellt, um ein Verständnis für die einzelnen Implementierungsschritte zu schaffen. Nachdem die Grundlagen der Systemkomponenten behandelt wurden, wird im weiteren Verlauf gezeigt, wie die einzelnen Komponenten zu implementieren sind. Hierbei werden die Zusammenhänge der Systemkomponenten genauer erläutert, um einen umfassenden Überblick über die Steuerung eines Roboterarms zu vermitteln. Abschließend werden die Resultate dieser Arbeit aufgeführt und behandelt. Außerdem wird ein Ausblick auf weiterführende Projekte und notwendige Weiterentwicklungen gegeben.

Diese Bachelorarbeit richtet sich vor allem an Studierende und Interessierte, die sich einen grundlegenden Überblick über die Implementierung einer Steuerung von Roboterarmen mit dem Robot Operating System verschaffen möchten.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

Abb. . . . . . Abbildung
API . . . . . Application Programming Interface
DHCP . . . . . Dynamic Host Configuration Protocol
ETTI . . . . . Elektrotechnik und Technische Informatik
IDE . . . . . Integrated Development Enviroment
IK . . . . . Inverse Kinematic
IP . . . . . Internet Protocol
KDL . . . . . Kinematics and Dynamics Libray
MDK . . . . . Microcontroller Development Kit
OMPL . . . . . Open Motion Planning Library
OS . . . . . Operating System
PC . . . . . Personal Computer
PWM . . . . . Pulsweitenmodulation
ROS . . . . . Robot Operating System
SRDF . . . . . Semantic Robot Description Format
UDP . . . . . User Datagram Protocol
URDF . . . . . Unified Robot Description Format
WE 4 . . . . . Wissenschaftliche Einrichtung 4
XML . . . . . Extensible Markup Language
XML-RPC . . . . . Extensible Markup Language-Remote Procedure Call
YAML . . . . . YAML Ain’t Markup Language

1 Einleitung

Diese Arbeit befasst sich mit der Entwicklung und Implementierung einer Steuerung für einen Roboterarm mit ROS. Als Grundlage hierzu dient eine Projektarbeit [3], die ein Konzept zur Steuerung eines Roboterarms mit dem Robot Operating System beschreibt.

Vorab sollen die Motive, die Absicht, sowie die Struktur der vorliegenden Arbeit kurz beschrieben werden.

1.1 Motivation

Im Dienstleistungssektor sowie im industriellen Bereich werden zunehmend Serviceroboter eingesetzt. Während Industrieroboter innerhalb einer festen Örtlichkeit betrieben werden und über fest installierte Schutzmechanismen verfügen, gestaltet sich die Umgebung von mobilen Servicerobotern dynamisch. Um die Sicherheit der Umwelt sowie der Menschen in der Umgebung zu gewährleisten, ist es unabdingbar, dass komplexe Bewegungsabläufe sicher geplant werden und auf die Veränderungen der Umgebung schnell und effektiv reagiert wird. Spätestens seit es Überlegungen gibt Pflegeroboter einzusetzen [20], nehmen die Anforderungen an Sicherheit und Zuverlässigkeit einen noch höheren Stellenwert ein.

Innerhalb der WE 4 für Daten- und Schaltungstechnik der Fakultät ETTI wird bisher erfolgreich an autonomen Fahrzeugen auf Basis des Robot Operating Systems geforscht. Bisherige Implementierungen von Roboterarmen basierten jedoch weitestgehend auf der direkten Ansteuerung der Servos, unter der Vorgabe von Winkelwerten. Berechnungen der Kinematik, Kollisionsvermeidung oder Pfadplanung fanden an dieser Stelle bisher kaum Anwendung. Aus diesem Grund wird in dieser Arbeit erstmals das Robot Operating System in Verbindung mit dem MoveIt!-Motion Planning Framework zur Steuerung von Roboterarmen eingesetzt. Dadurch soll die Forschung der wissenschaftlichen Einrichtung um die Fähigkeit der Steuerung von Roboterarmen mit ROS erweitert werden.

1.2 Aufgabenstellung und Anforderungen

Als Grundlage dieser Arbeit dient der Roboter “rob_arm_small“ sowie ein Mikrocontrollerboard, auf dem bereits die Interpretation von ROS-Steuerkommandos in PWM-Signale programmiert ist.

Realisiert wird die Steuerung mit dem MoveIt!-Motion Planning Framework (siehe Kapitel 3.2), das ein umfangreiches Werkzeug für die Steuerung von Roboterarmen sowie der mobilen Manipulation darstellt.

Ziel dieser Bachelorarbeit ist es, eine Implementierung für die Steuerung eines Roboterarms mittels ROS vorzunehmen und die Möglichkeiten, die MoveIt! bietet, auszunutzen.

Hieraus leiten sich folgende **Anforderungen** ab:

- Bewegungsplanung und -ausführung.
- Berechnung der Kinematik.
- Kollisionserkennung sowie -vermeidung.
- Manipulation von Objekten

Zunächst wird in dieser Ausarbeitung im Kapitel 2 ein Überblick über die genutzte Hard- und Software geschaffen. Anschließend werden im Kapitel 3 die Grundlagen der genutzten Komponenten und Frameworks vorgestellt. Insbesondere wird in diesem Kapitel auf das MoveIt!-Framework und das Package “actionlib“ eingegangen, die den Schwerpunkt dieser Bachelorarbeit bilden. Nach dem Erläutern der notwendigen Grundlagen wird im Kaptiel 4 die Implementierung der Steuerung vom Beginn bis zum aktuellen Stand dargelegt. Zudem werden in diesem Kapitel die Herausforderung dieser Arbeit näher beleuchtet. Abschließend werden im Kapitel 5 die Erkenntnisse dieser Bachelor-Thesis zusammengefasst sowie ein Ausblick über weitere Implementierungsschritte und mögliche Erweiterungen gegeben.

2 Systemaufbau

Das Gesamtsystem besteht aus mehreren Komponenten, die im Folgenden vorgestellt werden. Hierzu gehört zum einen die Hardware und zum anderen die, für die Implementierung notwendige, Software.

2.1 Hardware

Der Roboter “rob_arm_small“ dient als Entwicklungsplattform für die Steuerung. Aus diesem Grund werden die Spezifikationen und Einschränkungen des Roboters zuerst erläutert. Darauf folgend wird das Mikrocontrollerboard und dessen Aufgaben beschrieben.

2.1.1 Roboterplattform „rob_arm_small“

Der Roboter stammt von der Firma “CrustCrawler“ und trägt die Herstellerbezeichnung “SG6-UT“. Wie in Abbildung 2.1 ersichtlich, besitzt der Arm fünf Freiheitsgrade. Angetrieben werden die Glieder durch Servos der Firma “HiTEC“, die mit einer Spannung von 5V betrieben werden. Die Schwenkbereiche des Roboters sowie die zugehörigen Pulsweiten der Servos wurden in der vorangegangenen Projektarbeit [3] untersucht. Die aktuell gültigen Werte sind der Tabelle 2.1 zu entnehmen. Einen Sonderfall stellt an dieser Stelle der Greifer dar. Dieser wird ebenfalls durch einen Servo angetrieben. Über die Stellung dieses Servos wird der Abstand der beiden Greiferklammern gesteuert. In Tabelle 2.2 sind der Winkel des Servos, die zugehörige Pulsweite sowie die Griffweite des Grippers angegeben.

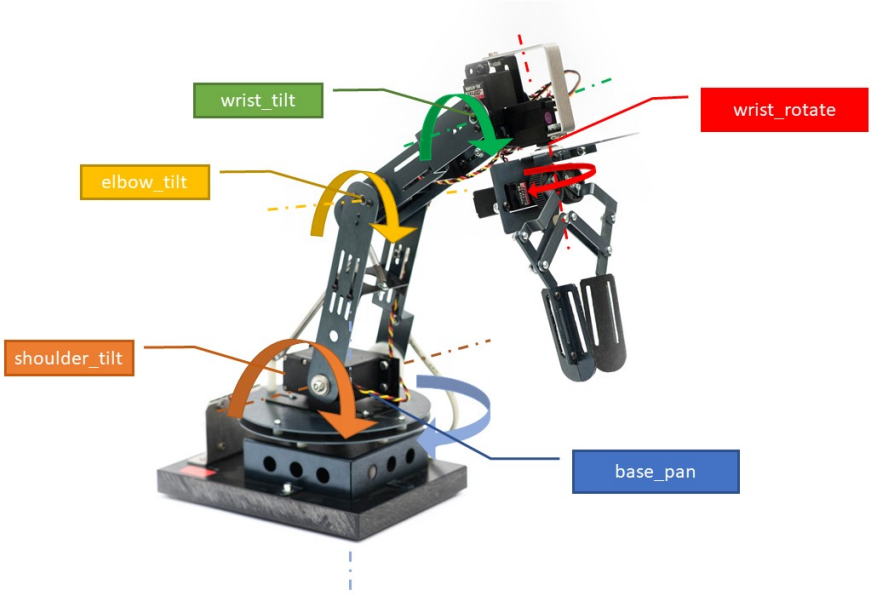


Abbildung 2.1: Freiheitsgrade des Roboterarms

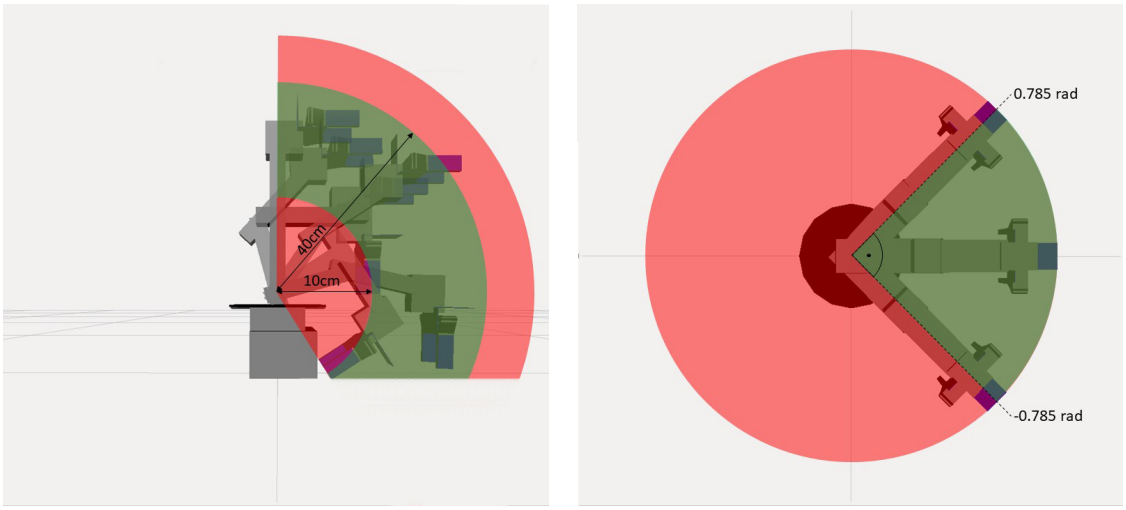
Joint	**Min. Winkel**			**Max. Winkel**		
	[rad]	[°]	[μs]	[rad]	[°]	[μs]
base_pan	-0.785	-45	1950	+0.785	+45	1050
shoulder_tilt	-0.261	-15	1854	+1.57	+90	1067
elbow_tilt	0.0	0	1906	+1.57	+90	1027
wrist_tilt	0.0	0	1934	+1.57	+90	1042
wrist_rotate	-0.785	-45	1959	+0.785	+45	1032

Tabelle 2.1: Winkelwerte der Gelenke

Joint	**Min. Winkel**			**Max. Winkel**		
	[rad]	[μs]	Öffnung [cm]	[rad]	[μs]	Öffnung [cm]
z_gripper	0.0	1555	6,4	1.0	1156	3,3

Tabelle 2.2: Griffweite des Greifers

Den Grafiken in Abbildung 2.2 können die zulässigen Operationsbereiche des Roboterarms entnommen werden. Für die Griffweite des Grippers wurde der Mittelpunkt des Greifers in vollständig geöffnetem Zustand gewählt. Dies hat den Grund, dass ein sicheres Zugreifen mit den Spitzen nicht vollständig gewährleistet werden kann. Infolge seiner Architektur kann der Roboterarm in den Bereichen von 10cm bis 40cm operieren (vgl. Abbildung 2.2(a)). Das Gelenk `shoulder_tilt` bildet für diese Werte den Ursprung. Wie aus Tabelle 2.1 und Abbildung 2.2(b) zu entnehmen ist, beschränkt sich der Schwenkbereich des Roboters auf $\pm 45°$.



(a) Zulässiger Arbeitsbereich in Y - Z Ebene (b) Zulässiger Arbeitsbereich in X - Y Ebene

Abbildung 2.2: Operationsbereiche des Roboterarms

2.1.2 Mikrocontrollerboard

Zur Umsetzung der ROS-Steuerkommandos dient ein vorab programmiertes STM32F746ZG Mikrokontrollerboard der Firma “STMicroelectronics“. Der Mikrokontroller nimmt dabei die Stellung der einzelnen Servos via Ethernet in Form von Winkelwerten entgegen und rechnet diese in die entsprechenden Pulsweiten um. Die berechneten Pulsweiten dienen anschließend den Timerbausteinen als Grundlage, die zugehörigen PWM-Signale zu erzeugen und diese an die betroffenen Servos weiterzuleiten. Zusätzlich wird zu Debugzwecken ein Telnet-Server bereitgestellt. Dieser ist unter der IP-Adresse des Boards erreichbar.

Zur Kommunikation zwischen PC und Mikrocontroller wird Ethernet genutzt. Hierbei kommt das verbindungslose UDP-Protokoll zum Einsatz.

2.2 Software

Die Entwicklung der Steuerung für Roboterarme basiert im Wesentlichen auf der Programmierung von Software. Um diese schreiben, testen und ausführen zu können, sind verschiedene Tools, Werkzeuge und Arbeitsumgebungen notwendig, die im Folgenden kurz vorgestellt werden.

2.2.1 Betriebssysteme

Als Betriebssystem dient Windows 10. Da ROS ein Linux-Betriebssystem als Grundlage benötigt, ist es notwendig, eine Linuxdistribution in einer virtuellen Maschine zu betreiben. In der WE4 ist seit Längerem ein solches System im Einsatz und wird kontinuierlich erweitert. Daher dient es in dieser Arbeit ebenfalls als Grundlage für die Entwicklung.

Virtualisiert wird an dieser Stelle ein Linux Ubuntu in der Version 16.04 LTS auf dem die ROS Version Kinetic Kame installiert ist.

2.2.2 Softwareentwicklungsumgebungen

Für die vorliegenden Bachelorarbeit war es notwendig, Programme zu erstellen oder vorhandene Software anzupassen. Zu diesen Zweck werden Softwareentwicklungsumgebungen eingesetzt. Neben den Möglichkeiten Programmcode zu kompilieren, stehen zudem umfangreiche Debugmöglichkeiten zur Verfügung, die die Fehlersuche erleichtern. Im Folgenden werden die wesentlichen Softwareentwicklungstools kurz vorgestellt.

Eclipse

Zur Programmierung der ROS-Nodes, Action Server und zum Anpassen der CMakeLists.txt dient die Open Source Softwareentwicklungsumgebung “Eclipse“. Diese IDE wurde von Frank Uhl im Rahmen einer Projektarbeit [29] so eingerichtet, dass diese mit dem ROS-Framework verwendbar ist .

Keil

Für die Anpassung und Debuggen der Software des Mikrocontrollers dient das “ARM Keil MDK“. Es bietet neben der μVision IDE und dem eben genannten Debugger einen ARM C/C++ Compiler.

2.2.3 Robot Operating System

Das Opensource-Projekt ROS [21] ist ein Framework zur Erstellung von Roboter-Software. Es beinhaltet verschiedene Werkzeuge, Bibliotheken und Frameworks. Ziel der Entwickler von ROS war es ein Framework, zur einfachen Entwicklung von Robotersoftware, zu schaffen.

Das Ziel von ROS ist, eine weltweite Community zu schaffen und auf diese Weise Spezialisten zusammenzuführen, um das Know-how aller an einem Ort zu bündeln. Im Grundlagenkapitel 3.1 wird genauer auf die Möglichkeiten, den Aufbau und den Umfang des ROS-Frameworks eingegangen.

2.2.4 MoveIt!-Motion Planning Framework

MoveIt! [16] ist eine Sammlung von Tools und Plugins, die der Entwicklung von Software für die mobile Manipulation durch Roboterarme dient. Die Fähigkeiten des Softwarepakets reichen von der Berechnung der Kinematik, über die Kollisionserkennung, bis hin zur Bewegungsplanung. Dem User steht ein umfangreiches Toolset zur Verfügung, das es ihm ermöglicht, individuell eine Steuerungssoftware für Roboterarme zu erstellen. Im Kapitel 3.2 werden die Möglichkeiten des Frameworks näher beleuchtet.

2.2.5 Systemaufbau

Abbildung 2.3 zeigt den schematischen der Aufbau des Systems. Der Grafik ist zu entnehmen, dass die Linuxdistribution innerhalb einer virtuellen Umgebung (hier VmWare-Player) gestartet wird. Die Netzwerkankopplung zum Host-Betriebssystem Windows ist im “Bridged-Modus“, sodass ROS direkten Zugriff auf die im Netzwerk verfügbaren Geräte verfügt.

Das Mikrocontrollerboard ist via Ethernet an das Labornetzwerk angeschlossen. Da der Mikrocontroller seine IP-Adresse dynamisch über DHCP bezieht, wurde in der Grafik auf eine Angabe der Netzwerkadresse verzichtet.

Die ROS-Node “hw_interface_node“ kommuniziert via UDP direkt mit dem Mikrocontroller. Dieser führt aufgrund der empfangenden Daten die Berechnung der Pulsweiten durch und stellt die aktualisierten PWM-Signale, an den Timer-Pins des Controllers, dem Roboter zur Verfügung.

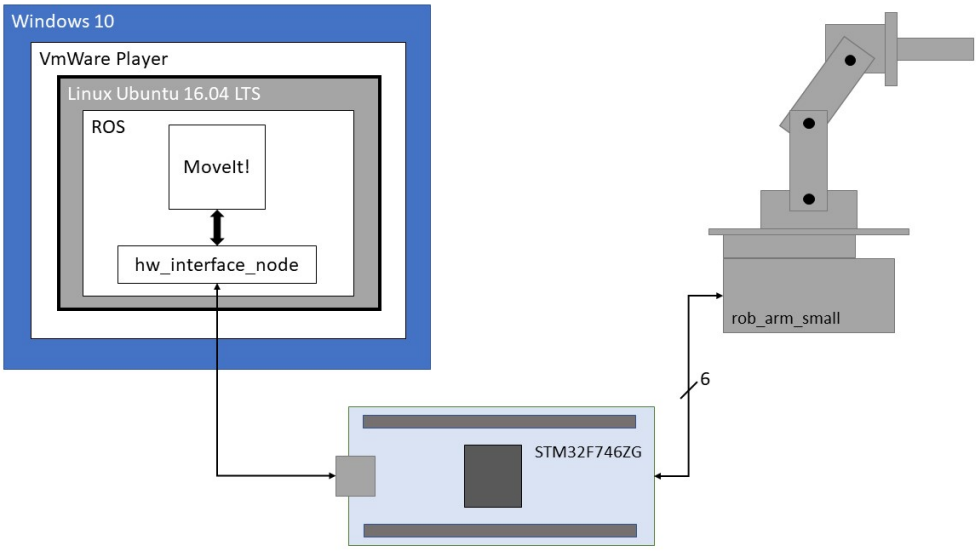


Abbildung 2.3: Schematischer Aufbau des Systems

3 Grundlagen

Im folgenden Kapitel sollen zunächst die Grundlagen zu den genutzten Systembausteinen, Plugins und Frameworks erläutert werden. Hauptaugenmerk liegt hierbei bei den für die Implementierung notwendigen Komponenten.

Im ersten Schritt wird das ROS-Framework vorgestellt. Aufbauend darauf das MoveIt!-Motion Planning Framework und anschließend weitere Systemkomponenten, die für die Implementierung erforderlich waren.

3.1 Robot Operating System

In diesem Abschnitt soll ein grundlegender Überblick über die Funktionalität und Arbeitsweise des ROS-Frameworks geschaffen werden, um folgende Kapitel besser nachvollziehen zu können.

ROS ist ein Open-Source Projekt, das ein Betriebssystem für Roboter darstellt. Auch wenn es kein eigenständiges Betriebssystem im traditionellen Sinn ist, adaptiert es einige Elemente eines klassischen OS. Es abstrahiert Hardware, verwaltet Prozesse und implementiert eine eigene Kommunikationsschicht. ROS agiert somit als Middleware zwischen Hardware und dem Linux-Betriebssystem.

Das Kernkonzept von ROS ist die Kommunikationsschicht. Über einen ROS-Master wird eine Punkt-zu-Punkt-Verbindung zwischen den Prozessen (ROS-Nodes /-Services) hergestellt, worüber Daten ausgetauscht werden können. Diese Kommunikationswege werden ROS-Topics genannt. Zum Übertragen der Nachrichten dienen ROS-Messages, die in ROS-Nodes oder ROS-Services generiert und verarbeitet werden.

Da die Kommunikation ausschließlich über diese Kommunikationsschicht läuft, die über XML-RPC organisiert wird, obliegt es dem Entwickler, welche Programmiersprache er nutzt, solange diese XML-RPC unterstützt [27].

Des Weiteren nutzt ROS statt eines monolithischen Systemaufbaus ein Mikrokernelprinzip. Das heißt, dass Systemkomponenten einzeln anstatt gemeinsam geladen werden, was zu einer höheren Ausfallsicherheit und Robustheit des Systems führt. Im Umkehrschluss bedeutet dies, dass der Absturz einer einzelnen Komponente nicht zwangsläufig den Ausfall des gesamten Systems bedeutet.

ROS-Master

Der ROS-Master bildet das Herzstück des Frameworks. Er stellt die Verbindungen zwischen den Verschiedenen Nodes und Services sicher. Da dieser nur als “Vermittler“ arbeitet, spielt es keine Rolle, auf welcher Plattform der ROS-Master läuft. Dies ermöglicht dem Nutzer mehrere Systeme miteinander zu koppeln.

ROS-Messages

ROS-Messages sind Datenstrukturen, die über ROS-Topics transferiert werden. Sie beinhalten primitive Datentypen. Neben den von ROS zur Verfügung gestellten Messages besteht die Möglichkeit, eigene Nachrichtentypen zu definieren und zu nutzen. Ebenso können die Messages beliebig ineinander verschachtelt werden.

ROS-Nodes

ROS-Nodes sind die Prozesse, in denen die eigentlichen Aufgaben ausgeführt werden. Sie lesen Sensordaten aus, steuern Aktuatoren an oder berechnen einen Pfad für die Bewegung eines Roboterarms.
Sobald eine Node gestartet wird registriert sich diese beim ROS-Master und meldet, welche Topics sie subscribed[1] oder pubslished[2].

ROS-Topics

ROS-Messages werden über eindeutig benannte Verbindungen transportiert, die Topics genannt werden. Diese Topics können von ROS-Nodes subscribed werden oder Nodes können auf Topics publishen, was bedeuetet, dass einerseits mehrere Nodes auf denselben Topics publishen sowie subscriben können (vgl. Abb.3.1 unten). Hierbei ist es unerheblich, ob eine Node von der Existenz einer anderen weiß. Aus diesem Grund agieren Topics nach dem “fire and forget“-Prinzip. Es erfolgt, ähnlich wie bei dem verbindungslosen Netzwerkprotokoll UDP, keine Rückmeldung des Zielsystems / -Node, ob eine Nachricht angekommen ist oder nicht.

[1] engl. abboniert
[2] engl. veröffentlicht

ROS-Services

Im Gegensatz zu ROS-Topics ermöglichen ROS-Services eine verbindungsorientierte Kommunikation. Das bedeutet, dass konkrete Anfragen einer Node in einer Service-Node verarbeitet und die Antwort an die anfragende Node zurückgeschickt wird (vgl. Abb.3.1 links). Erst danach werden weitere Programmschritte ausgeführt. Diese synchrone Kommunikation zwischen zwei Prozessen stellt zum einen sicher, dass Daten korrekt verarbeitet werden. Zum anderen schwächt dieses System jedoch die Stabilität des gesamten Systems, da bei Ausfall des Services ein Programm unter Umständen nicht mehr korrekt ausgeführt werden kann.

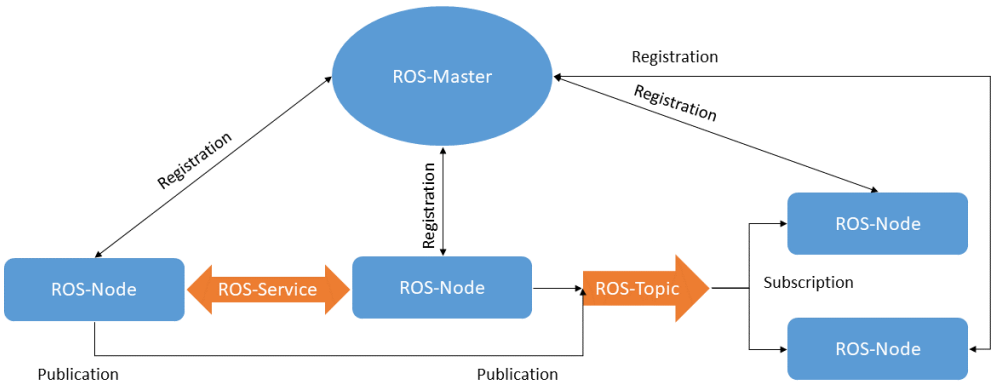


Abbildung 3.1: Schematische Darstellung der ROS-Kommunikationsarchitektur

ROS-Parameterserver

Der ROS-Parameterserver [24] dient zum Speichern und Austauschen von Daten zur Laufzeit. Da er nicht auf Geschwindigkeit ausgelegt ist, dient er hauptsächlich als Zwischenspeicher von statischen Inhalten, wie dem Robotermodell. Der Parameterserver nutzt ebenfalls XML-RPC und läuft innerhalb des ROS-Master.

3.2 MoveIt!-Motion Planning Framework

Das MoveIt!-Motion Planning Framework ist eine Toolsammlung für die mobile Manipulation in ROS. Es beinhaltet einen “fast inverse kinematics solver“ als Teil des Bewegungsplanungsalgorithmus. Des Weiteren Algorithmen für die Manipulation von Objekten, das Greifen, 3D-Wahrnehmung sowie Steuerung und Navigation. Neben der Implementierung der eben genannten Algorithmen bietet MoveIt! auch grafische Oberflächen für die Konfiguration neuer Roboter und das Anzeigetool RVIZ. In diesem Abschnitt soll die MoveIt!-Architektur (vgl. Abb.3.2) sowie das Basiskonzept genauer erläutert werden.

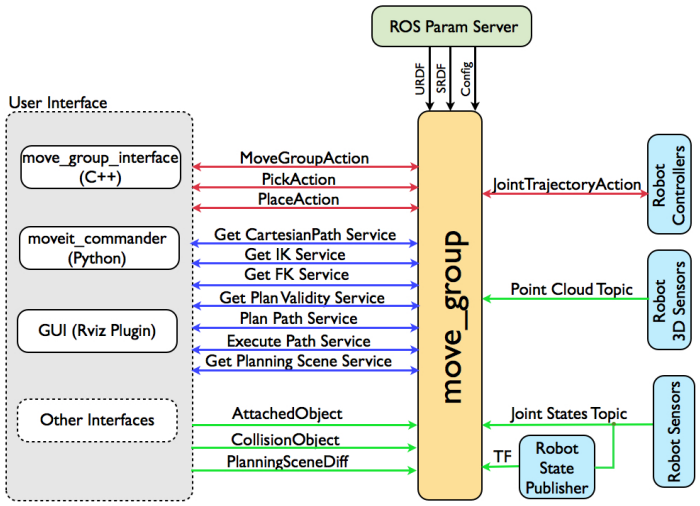


Abbildung 3.2: MoveIt! System Architektur[3]

move_group - Node

Im Zentrum der Gesamtarchitektur steht die `move_group-Node`. Sie dient als Bindeglied aller Komponenten und stellt verschiedene ROS-Actions und ROS-Services zur Verfügung.

Die `move_group-Node` lädt vom `ROS-param-server` die Daten der URDF und SRDF-Dateien, um das Robotermodell darzustellen und eine Grundlage für die Berechnungen zu haben. Ebenso prüft die Node, ob vom Standard abweichende MoveIt!-Konfigurationen vorliegen. Diese Konfigurationen können unter anderem Joint-Limits, Kinematikmodelle oder die Auswahl eines speziellen Bewegungsplaners sein.

[3](besucht am 26.05.2018): `http://moveit.ros.org/wordpress/wp-content/uploads/2013/12/Overview.0012.jpg`

User Interfaces

MoveIt! bzw. die `move_group-Node` bietet drei Wege, um auf die angebotenen Services und Actions zugreifen zu können.

C++-Schnittstelle: MoveIt! stellt hierfür das Paket `move_group_interface` zur Verfügung, das einen einfachen Zugriff auf die Funktionen der `move_group` bietet.

Python-Schnittstelle: Neben dem C++ Interface wird eine Python Schnittstelle angeboten. Der `moveit_commander` bietet dieselbe Funktionalität wie das C++ Interface.

RVIZ-Plugin: MoveIt! bietet für das ROS Visualisierungstool RVIZ das `Motion Planning Plugin` an. Hierbei handelt es sich um ein grafisches User Interface, das umfangreiche Möglichkeiten bietet.

Robot Interface

Die `move_group-Node` kommuniziert mit dem Roboter über ROS-Topics und ROS-Actions. Diese Kommunikation dient dazu, den aktuellen Zustand des Roboters zu ermitteln, Sensordaten abzufragen und Steuerinformationen an den Controller zu schicken.

Joint State Information: Die `move_group` hört auf den Topic `/joint_states`, um die aktuellen Zustandsinformationen, genauer die aktuelle Position der Joints, zu ermitteln. Publishen mehrere Nodes auf diesem Topic, kann die `move_group` diese Daten separieren und nur die für MoveIt! notwendigen Informationen weiterverwenden. Sollte der Topic `/joint_states` durch keine andere ROS-Node bedient werden, ist es notwendig den `joint_state_publisher` [12] manuell zu starten, da dies MoveIt! nicht selbstständig übernimmt.

Transform Information: MoveIt! nutzt die ROS-TF-Bibliothek [23], um globale Informationen über die Roboterpose und dessen Umwelt zu erhalten. Beispielsweise kann der ROS-Navigation-Stack die Transformation zwischen der Karte und dem Roboter-Koordinatensystem publishen. Die `move_group` wiederum kann daraus zum Beispiel die Gripper-Position ermitteln und korrekt abbilden. Hierfür ist der `robot_state_publisher` [22] notwendig, der, wie auch bei der Joint State Information, nicht automatisch gestartet wird und manuell gestartet werden muss.

Controller Interface: Ein Grundsatz von MoveIt! ist, so flexibel wie möglich zu sein. Daher ist die Implementierung der Ansteuerung allein dem Entwickler vorbehalten. MoveIt! kommuniziert hierfür über ein `FollowJointTrajectory Action-Interface`, um eine standardisierte Ansteuerung zu ermöglichen. Das bedeutet für den Entwickler, dass roboterseitig ein `FollowJointTrajectory Action-Server` implementiert werden muss, der von MoveIt! instantiiert wird.

Planning Scene: Um die Umwelt und den Roboter korrekt zu repräsentieren, nutzt `move_group` den `planning_scene_monitor`. Dieser fragt kontinuierlich den Status des Roboters sowie seiner Umgebung ab und generiert daraus ein Gesamtbild. Dies beinhaltet zum einen Objekte in der Umwelt und zum anderen die Pose des Roboters und verbundene Gegenstände.

Motion Planning

Die Bewegungsplanung innerhalb von MoveIt! befasst sich mit der Problematik, den Arm an eine bestimmte Position zu bewegen, ohne mit sich selbst oder einem Gegenstand in der Umwelt zu kollidieren oder Joint-Limits zu überschreiten. Hierfür stehen verschiedene Bibliotheken, wie zum Beispiel die standardmäßig genutzte OMPL [28], zur Verfügung. Auf das `motion_planning_interface` wird über, die von `move_group` angebotenen, ROS-Actions und ROS-Services zugegriffen. Im Folgenden wird auf die Schritte der Bewegungsplanung genauer eingegangen.

Motion Plan Request: Der Motion Plan Request beinhaltet die Zielinformationen, die an den Motion Planner gestellt werden. Dies kann zum einen ein Ziel in Form eines `JointSpaceGoal`, eines `PositionTarget` oder eines `PoseTarget` sein. Eine genaue Erläuterung zu den Zielformen folgt im Abschnitt 4.4.
Wird ein solcher Request verschickt und angenommen, wird automatisch bei der Pfadplanung geprüft, ob Hindernisse vorhanden sind. Im Rahmen der Kollisionsvermeidung werden Zusammenstöße mit vorhanden Objekten im Schwenkbereich sowie mit sich selbst vermieden und dementsprechend ein Pfad geplant, um diese zu verhindern.

Neben der Vorgabe eines Ziels besteht die Möglichkeit, verschiedene Einschränkungen für die Pfadplanung und Zielerreichung zu definieren.

Position Constraints: Ein Link darf einen definierten Bereich nicht verlassen.

Orientation Constraints: Ein Link darf definierte Rotationswinkel (Roll, Pitch, Yaw) nicht überschreiten.

Visibility Constraints: Ein definierter Punkt eines Links muss innerhalb des Erfassungsbereichs eines Sensors liegen.

Joint Constraints: Beschränkung eines Joints zwischen zwei Winkelwerten.

User-specified constraints: Der Nutzer kann eigene Beschränkunen definieren, die über einen `user-defined Callback` geprüft werden.

Motion Plan Result: Nach erfolgreicher Bewegungsplanung generiert die `move_group` einen Trajectory[4] dem der Arm folgt. MoveIt! nutzt hierbei die maximalen Geschwindigkeits- und Beschleunigungslimits für die Berechnung, um einen Pfad zu generieren, der diese Grenzen nicht verletzt.

Planning Scene

Die Planning Scene wird genutzt, um die Umwelt des Roboters sowie dessen eigenen Status zu repräsentieren. Dies wird durch den `planning_scene_monitor` gewährleistet. Dieser hört auf den `/joint_states-Topic`, erfasst Sensorinformationen über den `world_geometry_monitor` und nutzt Daten, die auf den `/planning_scene-Topic` gepublished werden.

Kinematics

MoveIt! nutzt für die Berechnung der inversen Kinematik standardmäßig KDL [19]. Dieses Plugin wird automatisch beim Ausführen des Setup Assistenten konfiguriert. Da MoveIt! eine Plugin-Infrastruktur besitzt, besteht die Möglichkeit einen eigenen Inverse-Kinematic-Solver zu implementieren. Die Berechnung der Vorwärtskinematik sowie der Jacobi-Matrizen erfolgt in der RobotState-Klasse.

Neben dem Standard-Kinematics-Plugin bietet MoveIt!, für die Implementierung eines eigenen Kinematic Solvers, zusätzlich das `IKFast Plugin` [9] an. Dies bietet die Möglichkeit, einen eigenen C++ - basierten IK-Solver zu generieren und einzubinden.

[4] engl. Flugbahn, Bewegungsbahn, Kurve

Collision Checking

Die Kollisionsprüfung von MoveIt! ist innerhalb der Planning Scene realisiert. Als Grundlage hierfür dienen `Collision-Objects`. Diese Elemente stellen Hindernisse sowie manipulierbare Objekte dar. Unterstützt werden dabei verschiedene Arten von Quellen. Über das `Planning Scene Interface` können primitive Formen, wie Boxen, Zylinder oder Kugeln, generiert sowie komplexe Meshes in die Welt geladen werden. Des Weiteren kann das Paket `Octomap` [18] in Verbindung mit einer 3D-Kamera genutzt werden. Auf diese Weise kann eine reale Darstellung der Umgebung in Form einer 3D-PointCloud für die Kollisionsprüfung generiert werden.

Trajectory Processing

Der Motion Planner generiert nur eine Bahnkurve, den der Roboter zurücklegen muss. Dieser Pfad beinhaltet keinerlei Informationen über das Timing. Über eine "Trajectory processing routine" werden die generierten Pfade zeitparametrisiert. Als Basis hierfür dient die Datei `joint_limits.yaml`. In dieser sind Beschränkungen bezüglich der Maximalgeschwindigkeit und Beschleunigung enthalten.

Setup Assistant

Der Setup Assistant ist ein grafisches Tool, das der Erstellung der MoveIt!-Konfigurationsdateien dient. Notwendig ist hierbei einzig eine Roboterbeschreibung in Form einer URDF-Datei.

3.3 Unified Robot Description Format

URDF ist ein einheitliches Roboter-Beschreibungsformat für ROS. Es basiert auf der XML-Syntax und nutzt spezielle Tags, die der Beschreibung des Roboters dienen. Durch diese Dateien kann ein visuelles sowie physikalisches Modell abgeleitet werden. Des Weiteren dienen diese Daten der Berechnung der Kinematik.

Nachfolgend soll auf die, für dieses Projekt wichtigsten, Elemente eingegangen werden. Umfangreichere Informationen sind unter [30] zu finden.

Link

Links repräsentieren die Glieder eines Roboters. Innerhalb dieses Tags können unter anderem Form, Abmessungen, Ausrichtung sowie Farbe parametrisiert werden. Weitere physikalische Eigenschaften lassen sich mit den Tags `<mass>` oder `<inertial>` festlegen. So kann die Masse eines Links angegeben oder, wie in Abbildung 3.3 gezeigt, der Schwerpunkt angepasst werden.

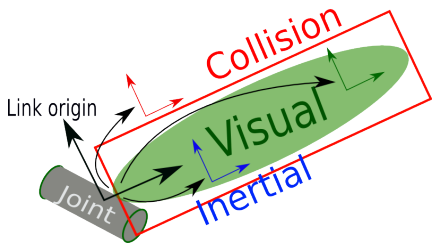


Abbildung 3.3: URDF Link[5]

Joint

Joints sind die Verbindungspunkte zwischen Links. Das Tag `<joint>` beschreibt die dynamischen und kinematischen Eigenschaften der Verbindung. Innerhalb dieses Tags können Aussagen über Limitierungen und Bewegungseigenschaften getroffen werden. Wesentlich für diese Arbeit ist der Typ `revolute`. Dieser gibt an, dass der Joint eine Rotationsbewegung nur innerhalb von festgelegten Begrenzungen durchführen kann. Hierfür dient das Tag `<limit>`. Insbesondere wird diese Eigenschaft bei der Bewegungsplanung wichtig. Für die Implementierung des Grippers war der Tag `<mimic>` erforderlich. Dieser sagt aus, dass ein Joint einem anderen Joint folgt. Hierfür muss der zu imitierende Joint angegeben werden. Ein Faktor ermöglicht eine Skalierung der imitierten Bewegung. Wie in Abb. 3.4 ersichtlich, verbinden Joints stets einen Elternlink mit einem Kindlink.

[5](besucht am 26.05.2018): `http://wiki.ros.org/urdf/XML/link?action=AttachFile&do=get&target=inertial.png`

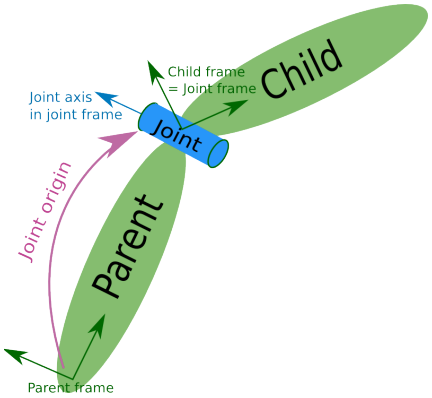


Abbildung 3.4: URDF Joint[6]

Robot

Wie einleitend erwähnt, besteht ein Robotermodell in der Regel aus mehreren Links, die mit Joints verbunden sind. Die Gesamtheit dieser Verbindungen werden unter dem Tag `<robot>` zusammengefasst und bilden somit das gesamte Modell. Abbildung 3.5 zeigt eine schematische Darstellung.

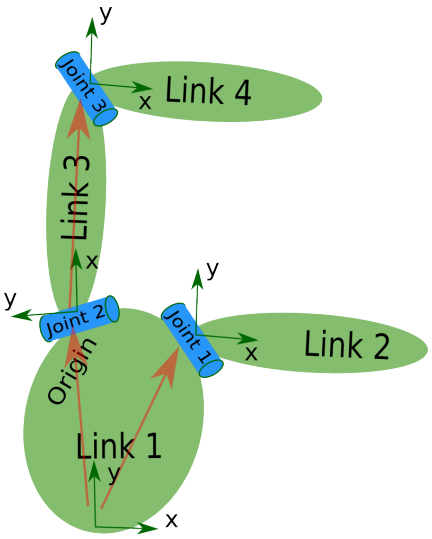


Abbildung 3.5: URDF Robot[7]

[6](besucht am 26.05.2018): `http://wiki.ros.org/urdf/XML/link?action=AttachFile&do=get&target=joint.png`

Im nachfolgenden Listing 3.1 ist ein Ausschnitt aus der verwendeten URDF-Datei dargestellt. Dabei zu erkennen ist die Gruppierung von Links und Joints durch das `<robot>` Tag. Des Weiteren ist in den Zeilen 3 bis 5 das Tag `<material>` erwähnt. Durch dieses Tag besteht die Möglichkeit, wiederkehrende Farbwerte unter einem Namen zu speichern und anschließend zu verwenden.

```
1  <robot name="RobArmSmall">
2
3          <material name="grey">
4                  <color rgba="0.5 0.5 0.5 1"/>
5          </material>
6
7
8          <link name="base_box">
9                  <visual>
10                         <geometry>
11                                 <box size="0.12624 0.1532 0.08861" />
12                         </geometry>
13                         <origin xyz="0.011 0.00 0.044305" rpy = "0 0 0"/>
14                         <material name="grey"/>
15                 </visual>
16         </link>
17
18         <joint name="base_link" type="fixed">
19                 <parent link="base_box" />
20                 <child link="base_top" />
21                 <origin xyz="0 0 0.08861" rpy="0 0 0" />
22         </joint>
23
24         <link name="base_top">
25                 <visual>
26                         <geometry>
27                                 <box size="0.10634 0.10634 0.04512" />
28                         </geometry>
29                         <origin xyz="0.00 0.00 0.02256" rpy = "0 0 0"/>
30                 <material name="grey"/>
31                 </visual>
32         </link>
33 </robot>
```

Listing 3.1: URDF-Syntax – Ausschnitt aus Datei: `rob_arm_small.urdf`

[7](besucht am 26.05.2018): `http://wiki.ros.org/urdf/XML/link?action=AttachFile&do=get&target=link.png`

3.4 ROS-Bibliothek „actionlib“

Im Abschnitt 3.1 wurde näher auf ROS-Services eingegangen. Diese senden eine Antwort auf eine vorher empfangene Anfrage. Gelegentlich ist es aber notwendig, dass eine solche Anfrage während ihrer Ausführung unterbrochen werden kann oder eine periodische Rückmeldung über die Entwicklung eines Prozesses erfolgt. Diese Möglichkeit bietet die ROS-Bibliothek `actionlib` [1]. Sie stellt eine standardisierte Schnittstelle für die Verarbeitung unterbrechbarer Tasks bereit.

Implementiert wird dieses System in einer Client/Server-Struktur, auf deren Konzept im Folgenden näher eingegangen und werden soll.

Client-Server Interaktion

Der Action Client und Action Server kommunizieren über ein `ROS Action Protocol`, welches oberhalb der Standard-`ROS-Messages` läuft. Server und Client stellen eine einfache API zur Verfügung, die es dem Nutzer ermöglicht, clientseitig Ziele anzufragen und diese serverseitig auszuführen (vgl. Abb. 3.6).

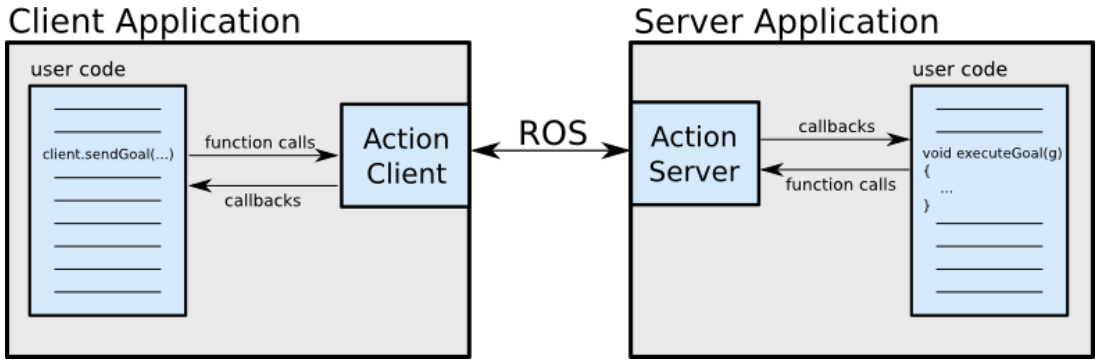


Abbildung 3.6: actionlib: Client-Server-Interaktion[8]

[8](besucht am 31.05.2018): `http://wiki.ros.org/actionlib?action=AttachFile&do=get&target=client_server_interaction.png`

Zustandsautomaten des Servers und Clients

Die `actionlib` arbeitet nach dem Prinzip eines zustandsgesteuerten Automaten. Da Server und Client jeweils eigene Zustandsmaschinen besitzen, müssen beide getrennt voneinander betrachtet werden. Seitens des Frameworks besteht die Forderung, dass ein Server implementiert wird. Aus diesem Grund wird zuerst dieser näher erläutert.

Sobald der Action Server ein Ziel eines Action Client erhält, wird für dieses ein Zustandsautomat generiert, der den Status des Ziels überwacht. An dieser Stelle ist zu erwähnen, dass für jedes Ziel ein neuer Zustandsautomat erstellt und dieser erst beim Erreichen eines Endzustandes wieder gelöscht wird.

Anhand der Abbildung 3.7 werden die Zustandsübergänge sowie die Zwischen- und Endzustände kurz erläutert.

Zustandsübergänge

setAccepted: Annehmen eines gültigen Zieles.

setRejected: Abweisen eines ungültigen Zieles.

setSucceeded: Benachrichtigen, dass ein Ziel erfolgreich bearbeitet wurde.

setAborted: Benachrichtigen, dass ein Fehler während der Verarbeitung auftrat und das Ziel nicht weiter bearbeitet wird.

setCanceled: Benachrichtigen, dass ein Ziel nach einem `cancel request` nicht weiter bearbeitet wird.

CancelRequest: Der Action Client informiert den Server, dass mit der Verarbeitung des Ziels gestoppt werden soll.

Zwischenzustände

Pending: Wartezustand des Ziels, das vom Action Server bearbeitet werden soll.

Active: Das Ziel wird aktuell verarbeitet.

Recalling: Das Ziel wurde noch nicht bearbeitet und ein `cancel request` wurde vom Action Server empfangen, der Abbruch wurde vom Server jedoch noch nicht bestätigt.

Preempting: Das Ziel wird aktuell verarbeitet und ein `cancel request` wurde empfangen, jedoch noch nicht vom Server bestätigt.

Endzustände

Rejected: Das Ziel wurde vom Action Server unverarbeitet zurückgewiesen.

Succeeded: Der Action Server hat das Ziel vollständig verarbeitet.

Aborted: Die Verarbeitung des Ziels wurde durch den Action Server und ohne externe Abbruchanfrage beendet.

Recalled: Das Ziel wurde durch ein neues Ziel ersetzt oder durch einen `cancel request` storniert, bevor der Action Server mit dessen Verarbeitung begonnen hat.

Preempted: Die Verarbeitung des Ziels wurde, durch ein neues Ziel oder einen `cancel request` an den Action Serer, unterbrochen.

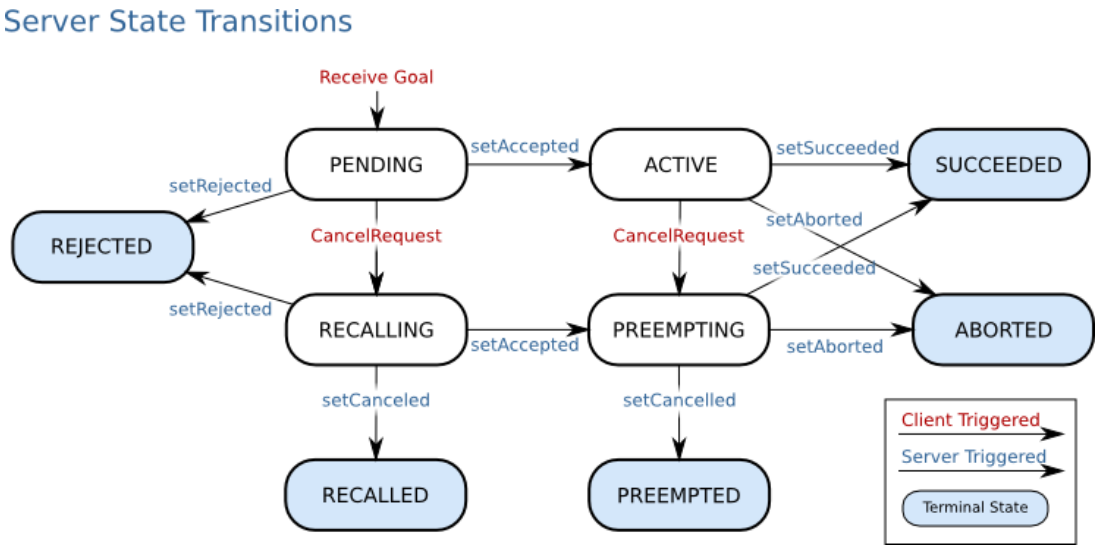


Abbildung 3.7: Action Server State Transitions[9]

Der Action Client, der seitens MoveIt! gestartet wird, besitzt wie oben angemerkt ebenfalls einen Zustandsautomaten. Während der Server-Zustandsautomat als Primärmaschine dient, wird der Automat der Clients als sekundärer Zustandsautomat bezeichnet. Seine Aufgabe ist es, den aktuellen Status des Action Servers zu überwachen.

[9](besucht am 11.06.2018): `http://wiki.ros.org/actionlib/DetailedDescription?action=AttachFile&do=get&target=server_states_detailed.png`

Aus diesem Grund ist die Mehrheit der Zustandsübergänge durch den Action Server gesteuert, was in Abbildung 3.8 durch die blauen Pfeile gekennzeichnet ist. Die Zustände des Clients haben dementsprechend dieselbe Bedeutung, wie die des Action Servers. Die roten Pfeile hingegen beschreiben die clientseitigen Zustandsübergänge. Im Wesentlichen handelt es sich hierbei um `cancel requests`, die einen Abbruch der Verarbeitung des Ziels bedeuten. Nicht eingezeichnet sind sogenannte "skipping states". Diese sind der ROS-Architektur geschuldet, da nicht vollständig davon ausgegangen werden kann, dass der Client sämtliche Statusupdates des Servers erhält.

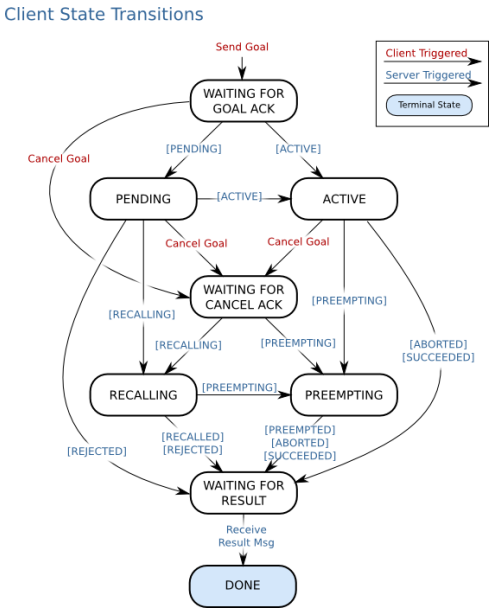


Abbildung 3.8: Action Client State Transitions[10]

[10](besucht am 11.06.2018): `http://wiki.ros.org/actionlib/DetailedDescription?action=AttachFile&do=get&target=client_state_transitions.png`

Action Messages

Das in 3.4 erwähnte `ROS Action Protocol` nutzt für die Kommunikation eigene Nachrichtentypen, sogenannte `Action-Messages`. In diesen Nachrichten werden ein Ziel (Goal), die Rückmeldung (Feedback) und das Ergebnis (Result) definiert. Für die genaue Definition der einzelnen Elemente dienen `.action-Dateien`. Vor der Beschreibung, wie eine solche Datei aufgebaut ist, soll auf die einzelnen Elemente näher eingegangen werden.

Goal: Um Aufgaben durch einen Action Server ausführen zu lassen, muss ein Ziel definiert werden. Dies kann zum einen eine bereits vorhandene ROS-Message sein, kann aber auch mit frei wählbaren Datentypen angelegt werden.

Result: Das Ergebnis wird vom Action Server an den Action Client geschickt. Es informiert den Client über die erfolgreiche Ausführung des Ziels und wird einmalig versendet. Neben der Information, dass das Ziel erreicht wurde, besteht die Möglichkeit, auch weitere Informationen, z.B. das Ergebnis eines Laserscans in Form einer PointCloud, zu übertragen.

Feedback: Die Rückmeldung dient für eine kontinuierliche Information des Clients über den aktuellen Zustand des Servers. Beispielsweise über die aktuelle Position des Roboters, während dieser einem vorgegebenen Pfad, dessen Ziel im Goal angegeben war, folgt.

Die Definition der oben genannten Elemente erfolgt in `.action-Dateien`. Der Aufbau dieser Dateien stellt sich wie folgt dar: Im ersten Schritt wird das Ziel festgelegt, anschließend folgen die Definitionen des Ergebnisses und des Feedbacks, jeweils getrennt von drei Bindestrichen (`---`). Eine `.action-Datei`, für das Zählen von Fahrzeugen, könnte wie in Listing 3.2 dargestellt aussehen.

```
1 \# Define the goal
2 uint32 count_cars_for_five_minutes
3 ---
4 \# Define the result
5 uint32 total_cars_counted
6 ---
7 \# Define a feedback message
8 float32 current_counted_cars
```

Listing 3.2: countCars.action

Die `.action-Dateien` müssen innerhalb des Packages im Ordner `<package>/action` abgelegt werden.

Damit aus den `.action-Dateien` beim Kompilieren `Action-Messages` generiert werden, müssen in der Datei CMakeLists.txt noch folgende Zeilen aus Listing 3.4 ergänzt werden. Hierbei ist darauf zu achten, dass diese vor dem Befehl `catkin_package()` ausgeführt werden.

```
find_package(catkin REQUIRED genmsg actionlib_msgs actionlib)
add_action_files(DIRECTORY action FILES countCars.action)
generate_messages(DEPENDENCIES actionlib_msgs)
```

Listing 3.3: ActionMessage CMakeLists.txt-Entries

Zusätzlich müssen innerhalb der `package.xml` folgende Abhängigkeiten ergänzt werden.

```
<build_depend>actionlib</build_depend>
<build_depend>actionlib_msgs</build_depend>
<run_depend>actionlib</run_depend>
<run_depend>actionlib_msgs</run_depend>
```

Listing 3.4: ActionMessage package.xml-Entries

Mit den oben genannten Einträgen werden beim Kompilieren die folgenden `Action-Messages` erzeugt. Diese werden anschließend vom Paket `actionlib` zur Kommunikation zwischen Action Server und Action Client genutzt.

- `countCarsAction.msg`
- `countCarsActionGoal.msg`
- `countCarsActionResult.msg`
- `countCarsActionFeedback.msg`
- `countCarsGoal.msg`
- `countCarsResult.msg`
- `countCarsFeedback.msg`

Weiterführende Informationen über die Funktionsweise der Server, Clients und des Transportlayers sind unter [8] zu finden.

Simple Action Server

Das MoveIt!-Framework setzt für die Nutzung von Roboterhardware die Implementierung verschiedener Action Server voraus. Das Package `actionlib` bietet hierfür eine Vielzahl an Möglichkeiten. In diesem Abschnitt soll die Variante näher betrachtet werden, die bei der Implementierung angewandt wurde. Weiterführende Methoden und Informationen zur Erstellung von Action Clients und Action Servern sind auf den Dokumentations- und Tutorialseiten [1] und [2] der `actionlib` zu finden.

Wie im vorherigen Absatz angesprochen, wurden die Action Server nach einem bestimmten Prinzip implementiert. Die in dieser Arbeit genutzte Variante verfährt mit der Callback-Methode [26]. Diese Art der Implementierung wird in diesem Abschnitt genauer betrachtet werden. Hierzu dient das im Anhang vollständig abgebildete Beispiel-Listing A.1. Die essenziellen Elemente werden hier auszugsweise erläutert.

Im ersten Schritt muss ein Action Server definiert werden. Dies geschieht mit dem Befehl aus Listing 3.5. Während der Konstruktion der Action-Klasse wird der Nodehandle an den Action Server übergeben und somit beim ROS-Master registriert.

```
51  ros::NodeHandle nh_;
52  actionlib::SimpleActionServer<simple_action_server_example::ExampleAction> as_;
```

Listing 3.5: Definition des Action Server

Im Konstruktor der Action-Klasse wird der Action Server generiert und die Callback-Methoden registriert. Der Action Server nimmt dabei folgende Argumente entgegen (vgl. Listing 3.6 Zeile 8): Einen Nodehandle, den Namen der Action und eine optionale `Execute Callback-Methode`, die in diesem Beispiel nicht genutzt wird.
In den darauffolgenden Zeilen werden die Callback-Methoden registriert. Der Aufruf in Zeile 12 stellt hierbei eine Besonderheit dar. Die `Feedback-Callback-Methode` subscribed einen "gewöhnlichen" Topic. Das bedeutet, dass diese Methode mit der Frequenz ausgeführt wird, mit der eine andere Node auf diesem Topic published. Daraus ergibt sich ein periodischer Aufruf und somit ein periodisches Feedback.

Über den Befehl in Zeile 13 wird der Action Server gestartet.

```
8   ExampleAction(std::string name) : as_(nh_, name, false), action_name_(name){
9     //Register Callback-Methos
10    as_.registerGoalCallback(boost::bind(&ExampleAction::goalCB, this));
11    as_.registerPreemptCallback(boost::bind(&ExampleActionAction::preemptCB, this));
12    sub_ = nh_.subscribe("/random_topic", 1, &ExampleAction::feedbackCB, this);
13    as_.start();
```

Listing 3.6: Konstruktor des Action Server

Nach der Registrierung der Callback-Methoden müssen diese implementiert werden. Die `Goal-Callback-Methode` dient zum einen der Annahme des Ziels und zum anderen dazu, die übertragenen Daten zu speichern (vgl. Listing 3.7). Hierbei wird der Status auf `active` gesetzt.

Sollte ein neues Ziel vorgegeben werden während das vorherige noch nicht abgearbeitet wurde, wird das alte verworfen und dessen Status auf `preempted` gesetzt. Das neue Ziel wird anschließend übernommen und verarbeitet.

```
19  void goalCB(){
20    //Accept new goal
21    goal_ = as_.acceptNewGoal()->samples; //The ->samples Statements depends on .action-
          File
22  }
```

Listing 3.7: Action Server: Goal Callback Methode

Da der Action Server eventbasiert arbeitet, ist es notwendig, dass zuvor angenommene Ziele ebenfalls unterbrochen werden können. Beispielsweise wenn ein falsches Ziel vorgegeben wurde. Für diesen Zweck kann eine `Preempt-Callback-Methode` implementiert werden, die den Aufruf `as_.setPreempted()` enthält. Im Listing 3.8 ist diese abgebildet.

```
24  void preemptCB(){
25    ROS_INFO("%s: Preempted", action_name_.c_str());
26    // set the action state to preempted
27    as_.setPreempted();
28  }
```

Listing 3.8: Action Server: Preempt Callback-Methode

Um dem Nutzer einen Überblick über die Entwicklung eines Prozesses zu geben, muss eine `Feedback-Callback-Methode` programmiert werden. Wie in Listing 3.6 Zeile 12 angesprochen, wird der Callback durch eine Nachricht auf einem "gewöhnlichen" Topic aufgerufen. Diese Methode nimmt die zu verarbeitenden Daten entgegen und führt die gewünschten Operationen aus. In der `Feedback-Callback-Methode` wird die eigentliche Aufgabe abgearbeitet.

Im ersten Schritt gilt es zu prüfen, ob ein gültiges Ziel vorliegt. Sollte dies nicht der Fall sein, wird von der weiteren Abarbeitung abgesehen.

Über den Befehl `as_.publishFeedback(feedback_)` (vgl. Listing 3.9 Zeile 37) wird regelmäßig Auskunft über den aktuellen Zustand des Prozesses gegeben. In den darauf folgenden Zeilen wird überprüft, ob das Ziel erreicht wurde. Ist das der Fall, wird der Zustand des Action Server über den Befehl `as_.setSucceeded(result_)` auf den Zustand `succeeded` gesetzt under der Client über die erfolgreiche Ausführung informiert. Das Ergebnis wird in der Variable `result_` abgespeichert. Sollte ein fehlerhaftes Ergebnis auftreten, muss der Action Client ebenfalls darüber informiert werden. Hierfür dient der in Zeile 42 angegebene Befehl `as_.setAborted(result_)`. Auch in diesem Fall wird das Ergebnis abgespeichert und kann für die Fehleranalyse ausgelesen und verarbeitet werden.

```
30  void feedbackCB(const std_msgs::Float32::ConstPtr& msg){
31    //Check if valid goal is active
32    if (!as_.isActive()){
33      return;
34    }
35
36    //Do something and publish periodic feedback
37    as_.publishFeedback(feedback_);
38
39    //Check if goal is satisfied
40    if(data_ > goal_){
41      //Set action state to aborted
42      as_.setAborted(result_);
43    }else if (data_ == goal_){
44      //Set action state to succeeded
45      as_.setSucceeded(result_);
46    }
47  }
```

Listing 3.9: Action Server: Feedback Callback-Methode

4 Implementierung der Steuerung für einen Roboterarm mit ROS

Während im vorherigen Kapitel die Grundlagen von MoveIt! und der notwendigen Packages erläutert wurden, widmet sich dieser Teil der Arbeit ganz der eigentlichen Implementierung.

Ziel ist es, einen umfassenden Überblick über die notwendigen Schritte und identifizierten Fehlerquellen zu geben. Für eine einfachere Nachvollziehbarkeit wird dieses Kapitel in derselben chronologischen Reihenfolge, wie auch die Implementierung stattfand, aufgebaut sein.

Am Ende dieses Kapitels werden die Zusammenhänge der einzelnen Komponenten erläutert, um ein Gesamtverständnis des Systems zu schaffen.

Unter der Voraussetzung, dass ROS in der Version Kinetic installiert ist [11], kann das MoveIt!-Framework mit folgendem Terminalbefehl installiert werden.

```
$ sudo apt-get install ros-kinetic-moveit
```

Anschließend müssen mit dem nachfolgendem Befehl die Umgebungsvariablen neu geladen werden.

```
$ source /opt/ros/kinetic/setup.bash
```

Umfangreiche Tutorials zu MoveIt! sowie dessen Installation sind auf der MoveIt!-Tutorial-Homepage [17] zu finden.

4.1 Grundlegende Implementierungsschritte

4.1.1 Entwurf des Robotermodells

Um einen neuen Roboter in die MoveIt!-Umgebung zu integrieren, ist ein Modell des Roboters notwendig. Als Grundlage hierfür dient das in Kapitel 3.3 erläuterte URDF-Beschreibungsformat. In diesem sind die Abmessungen des Roboters nachgebildet sowie die Einschränkungen bezüglich der Rotationsbewegungen der einzelnen Joints erfasst.

Um den Fokus auf die eigentliche Implementierung der Steuerung zu legen, dient für die weitere Nutzung ein vereinfachtes Robotermodell, welches in Abbildung 4.1 dargestellt ist.

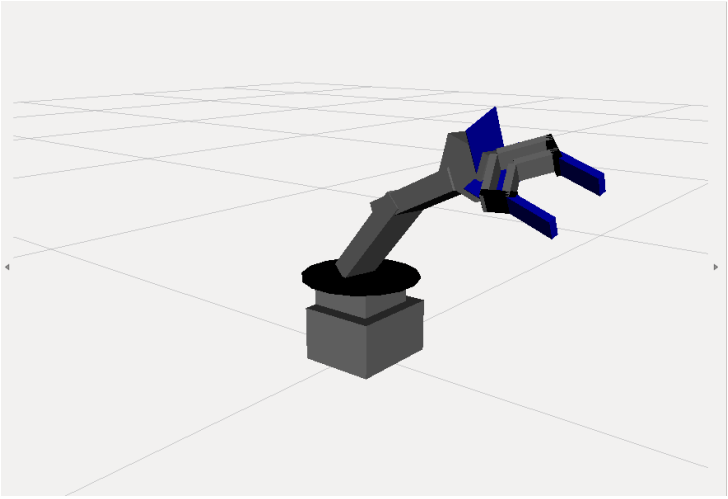


Abbildung 4.1: URDF Modell des Roboterarms

4.1.2 Erstellen eines Workspace

Damit ein neu erstelltes Package durch das Robot Operating System gefunden wird und im weiteren Verlauf die benötigten Nodes mit den Kommandos `rosrun` bzw. `roslaunch` gestartet werden können, ist das Erstellen eines Catkin-Workspace notwendig. In diesem Workspace werden anschließend alle weiteren Dateien abgelegt. Der Vorgang, einen solchen Workspace anzulegen, soll an dieser Stelle nur kurz beschrieben werden. Ein ausführliches Tutorial sowie weitere Informationen diesbezüglich sind auf der ROS-Website [5] zu finden.

Um einen Workspace zu erstellen, sind folgende Befehle in einem Terminalfenster einzugeben:

```
$ mkdir -p ~/<ROS_workspace>/src
$ cd ~/<ROS_workspace>/
$ catkin_make
```

Nach diesen Schritten ist der Workspace erzeugt, jedoch kann noch nicht mit den Befehlen `roslaunch` und `rosrun` darauf zugegriffen werden. Hierzu müssen zunächst die ROS-Umgebungsvariablen aktualisiert werden. Dies geschieht mit folgendem Befehl.

```
$ source devel/setup.bash
```

4.1.3 Setup Assistant

Für die Integration des Robotermodells ist die Erstellung weiterer Dateien notwendig. Um dem Entwickler aufwendiges Programmieren zu ersparen, bietet MoveIt! den, in Abbildung 4.2 gezeigten, Setup Assistenten an. Dieses grafische Userinterface leitet den Nutzer durch die notwendigen Konfigurationsschritte, um alle weiteren erforderlichen Dateien für die Nutzung des Roboters innerhalb von MoveIt! zu erstellen.

Starten des Setup Assistenten

Gestartet wird dieser mit folgendem Terminalbefehl.

```
$ roslaunch moveit_setup_assistant setup_assistant.launch
```

Es öffnet sich der Startbildschirm des Setup Assistenten (vgl. Abbildung 4.2). In diesem Fenster ist für die Erstellung eines neuen Konfigurationspackages “Create New Moveit Configuration Package“ zu wählen. Anschließend kann über die “Browse“-Schaltfläche zur URDF-Datei navigiert und über “Load Files“ die Datei geladen werden.

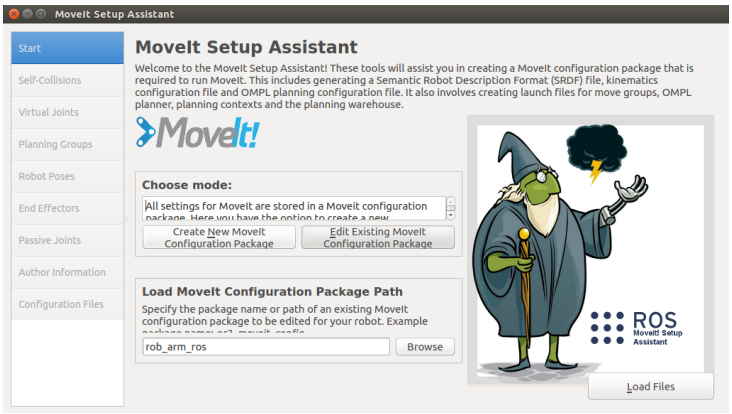


Abbildung 4.2: MoveIt! Setup Assistant

Berechnung der Selbstkollisionsmatrix

Der Setup Assistent bietet einen Generator für die Berechnung der Selbstkollisionsmatrix an. Dieser sucht nach Links, die dauerhaft aus der Kollisionsprüfung ausgenommen werden können. Dies dient vornehmlich der Verkürzung der Planungszeit des Motion Planners. Hierbei handelt es sich um Links, die sich dauerhaft in Kollision befinden, niemals kollidieren können oder um benachbarte Links einer kinematischen Kette. In Abbildung 4.3 ist das Fenster dargestellt.

Über den Regler “Sampling Density“ kann die Genauigkeit der Prüfung eingestellt werden. Genauer wird eine Reihe von zufälligen Roboterposen angenommen und daraus errechnet, welche Links kollidieren. Je geringer die Dichte gewählt wird, desto höher ist die Wahrscheinlichkeit, dass Links aus der Kollisionsprüfung ausgenommen werden, die nicht ausgenommen werden sollten. Über die Schaltfläche “Generate Collision Matrix“ kann die Berechnung gestartet werden.

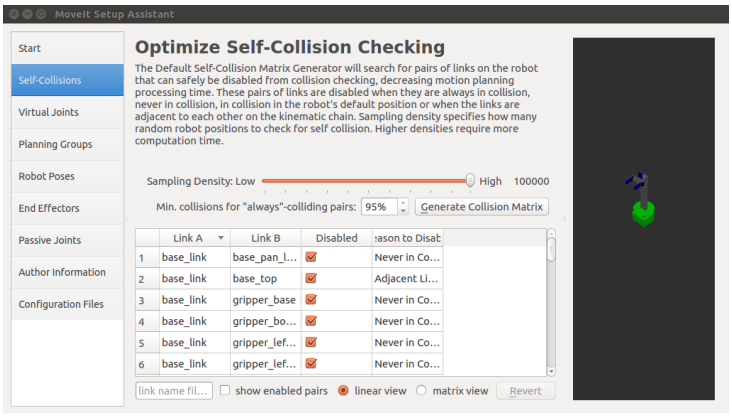


Abbildung 4.3: Berechnung der Selbstkollisionsmatrix

Hinzufügen virtueller Joints

Virtuelle Joints dienen vornehmlich der Positionierung des Roboters in der Planungswelt. Im vorliegenden Fall ist ein virtueller Joint ausreichend. Mit diesem wird der Link “base_link“ am Untergrund fixiert. Anhand des in Abbildung 4.4 gezeigten Fensters wird wie folgt vorgegangen.

- “Add Virtual Joint“ anklicken, um einen virtuellen Joint zu erstellen.
- Den eben erstellen Joint benennen.
- Als “Child Link“ wurde hier “base_link“ und als “Parent Frame Name“ `odom_combined` gewählt.
- Da der Roboter nicht mobil ist, wurde hier der JointType “fixed“ gewählt, um ihn am Ursprung des Weltkoordinatensystems zu fixieren.

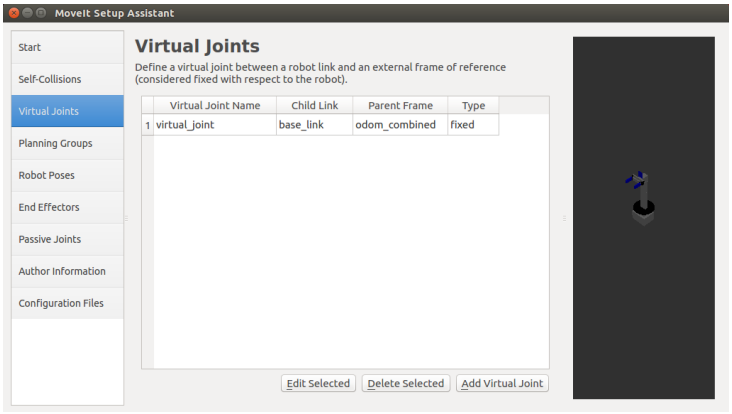


Abbildung 4.4: Hinzufügen virtueller Joints

Hinzufügen von Planungsgruppen

Planungsgruppen sind eine Bündelung von Joints des Roboters, die im späteren Verlauf für die Bewegungsplanung genutzt werden. Im Falle dieser Arbeit besteht der Roboter aus zwei Planungsgruppen. Zum einen aus dem Roboterarm selbst und zum anderen aus dem Gripper. Die Konfiguration der dieser Gruppen erfolgt in den in Abbildung 4.5 dargestellten Fenstern.

Zunächst muss eine Planungsgruppe, wie in Abbildung 4.5(a) abgebildet, definiert werden. Anschließend werden entweder die zur Gruppe gehörigen Joints ausgewählt (s. Abbildung 4.5(b)) oder eine `kinematic chain` definiert. Das geschieht, wie in Abbildung 4.5(c) aufgezeigt, durch die Auswahl des ersten und des letzten Joints der gewünschten Kette.

- Mit der Schaltfläche “Add Group“ öffnet sich das Erstellungsfenster der Planungsgruppen (vgl. Abb. 4.5(a)).

- Vergeben eines Namens für die Planungsgruppe, hier “roboter_arm“.

- Auswahl des Berechnungsalgorithmus für die Kinematik. In dieser Arbeit wurde der “kdl_kinematics_plugin/KDLKinematicsPlugin“ gewählt.

- Die Werte “Kin. Search Resolution“ sowie “Kin. Search Timeout“ wurden unverändert übernommen.

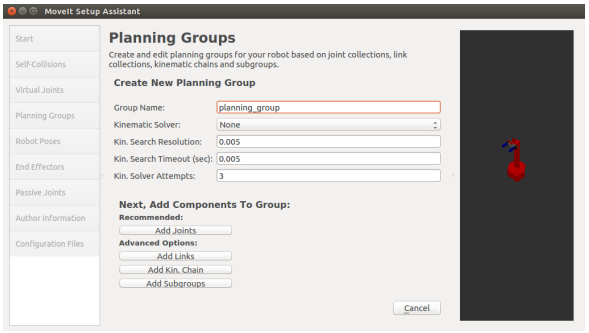
- Mit einem Klick auf die Schaltfläche “Add Joints“ wird das Auswahlfenster der Joints geöffnet (vgl. Abb. 4.5(b)).
  In diesem Fenster sind sämtliche Joints des Roboters, soweit diese in der URDF-Datei beschrieben wurden, aufgelistet. Hier sind alle Joints zu wählen, die in die jeweilige Planungsgruppe übernommen werden sollen. Für den Fall des vorliegenden Roboterarms sind das die Joints vom “virtual_joint“ bis zum Joint “z_gripper“.

- Alternativ kann eine kinematische Kette[11] definiert werden. Dies geschieht durch den Klick auf “Add Kin. Chain“ (vgl. Abb. 4.5(c))

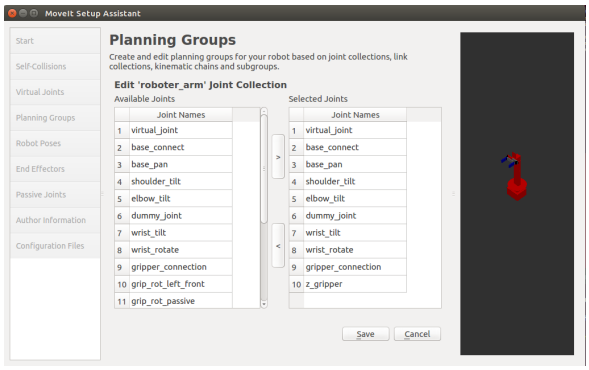
- Mit einem Klick auf die Schaltfläche “Save group“ wird die Planungsgruppe gespeichert.

Für die Planungsgruppe des Grippers wird analog verfahren. Einzig die Auswahl der Joints ist eine andere. Für diese Gruppe werden die Joints `wrist_tilt`, `wrist_rotate` und `z_gripper` ausgewählt.

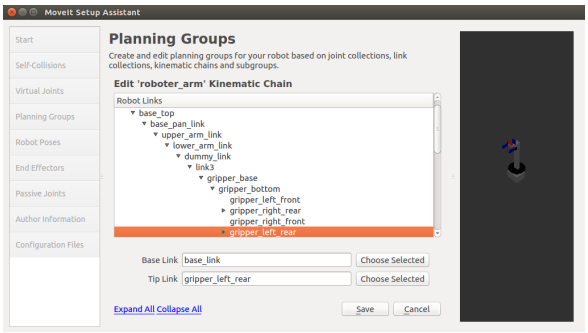
[11] Kette von aufeinanderfolgenden Gliedern und Joints



(a) Erstellen einer Planungsgruppe



(b) Auswahl der Joints



(c) Erstellen einer kinematischen Kette

Abbildung 4.5: Hinzufügen von Planungsgruppen

Roboterposen

Im nächsten Reiter können Roboterposen, wie zum Beispiel eine Homepose, definiert werden. Da in dieser Arbeit kein Nutzen davon gemacht wurde, wird darauf nicht näher eingegangen.

Endeffektor

Planungsgruppen dienen als Grundlage für die Bewegungsplanung. Dies gilt für die Gruppe `roboter_arm` sowie für die Gruppe `gripper`. Um mit dem Gripper jedoch spezielle Aufgaben, wie z.B. das Greifen von Objekten, durchführen zu können, ist es notwendig, dass dieser als Endeffektor gekennzeichnet wird. Sollte dieser nicht definiert sein, wird das letzte Glied der für die Planung ausgewählten Planungsgruppe als Endeffektor genutzt.

Ein Endeffektor stellt das Werkzeug eines Manipulators dar. Das können unterschiedlichste Werkzeuge sein und reichen von Greifern bis hin zu Schweißgeräten. Die Kennzeichnung eines Endeffektors geschieht in dem in Abbildung 4.6 abgebildeten Fenster. Die einzelnen Schritte zur Konfiguration des Endeffektors lauten wie folgt:

- Dem Endeffektor muss ein Name zugewiesen werden.
- Die unter Planungsgruppen definierte Gruppe für den Endeffektor muss ausgewählt werden. Im vorliegenden Beispiel ist dies die Gruppe `gripper`.
- Unter “Parent Link“ wird der Link angegeben, an dem der Endeffektor befestigt ist. Für den genutzten Roboterarm ist dies der `gripper_left_rear`.
- Optional kann eine “Parent Group“ angegeben werden. Hierbei wird dem Gripper als “Parent Link“ das letzte Glied der ausgewählten Planungsgruppe zugewiesen.

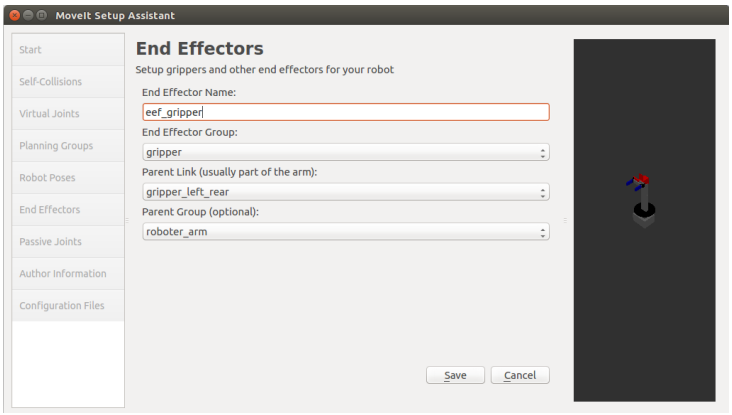


Abbildung 4.6: Hinzufügen eines Endeffektors

Passive Joints

Passive Joints sind Verbindungen, die nur indirekt Bewegungen ausführen. Das heißt, diese sind nicht direkt durch einen Motor angetrieben. Dies dient vornehmlich dem Kinematic-Solver, da dieser darüber informiert wird, dass er die betroffenen Joints nicht in die Planung einbeziehen muss.

Für den genutzten Roboterarm trifft dies auf einige Joints der Gripper-Baugruppe zu, da sie nur indirekt angesteuert werden. Die Auswahl erfolgt analog zur Auswahl der Gelenke der Planungsgruppen (vgl. Abbildung 4.7).

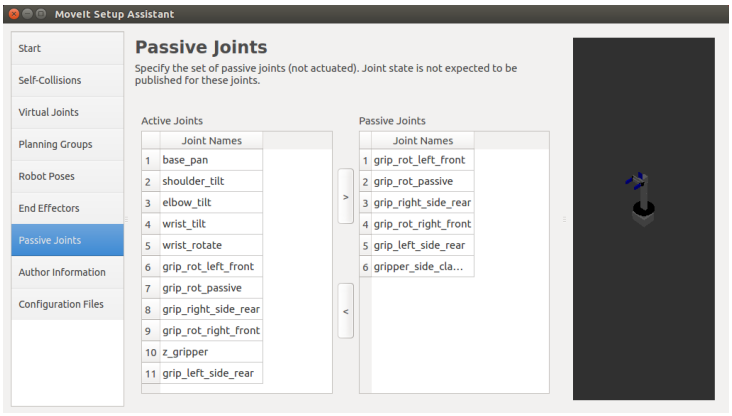


Abbildung 4.7: Auswahl passiver Joints

Autoreninformationen und Generierung der Konfigurationsdateien

Im letzten Schritt des Setup Assistenten müssen die Daten des Autors angegeben werden. Dies dient für eine etwaige Veröffentlichung der Konfigurationsdateien.

Durch einen Klick auf die Schaltfläche “Generate Package“, im Fenster “Configuration Files“ (vgl. Abbildung 4.9), werden, im oben angegebenen Verzeichnispfad, die MoveIt! Konfigurationsdateien generiert und stehen ab sofort zur Verfügung. In Abbildung 4.8 ist exemplarisch die aktuelle Verzeichnisstruktur dieser Arbeit abgebildet.

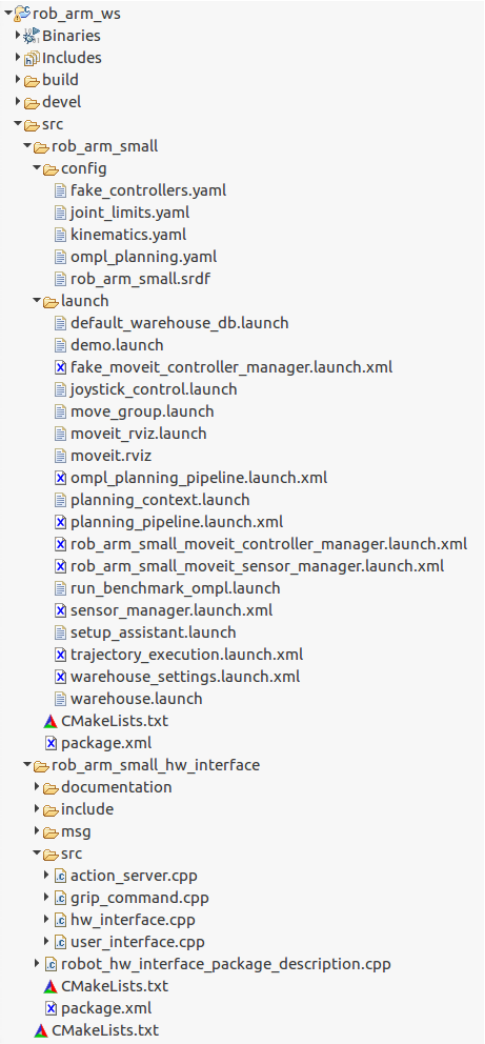


Abbildung 4.8: Verzeichnisstruktur des Projekts

Der Verzeichnispfad sollte folgende Form aufweisen:

```
~/<ROS_workspace>/src/<Package_name>/src/<robot_name>
```

Nach dem Erstellen der Konfigurationsdaten ist es notwendig, das Package, zu generieren und somit der ROS-Infrastruktur zugänglich zu machen. Dies geschieht mit folgenden Befehlen:

```
$ cd ~/<ROS_workspace>/<package_name>
$ catkin_make
$ source /devel/setup.bash
```

Für den Fall, dass Änderungen an den Konfigurationseinstellungen vorgenommen werden müssen oder die URDF-Datei geändert wurde, besteht die Möglichkeit den Setup Assistenten mit den generierten Daten erneut zu laden. Änderungen des URDF werden hierbei automatisch übernommen, da diese Datei neu eingelesen wird. Zu diesem Zweck wird bei der Erstellung des Pakets eine Launch-Datei erzeugt, die das ermöglicht. Gestartet wird diese mit folgendem Befehl:

```
$ roslaunch <robot_name> setup_assistant.launch
```

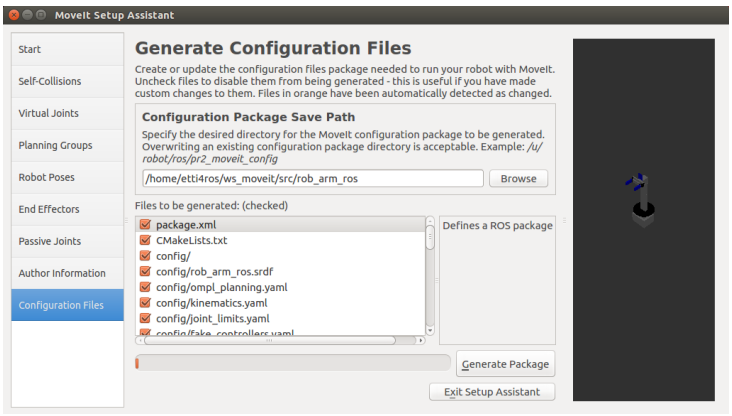


Abbildung 4.9: Erstellen der Konfigurationsdateien

Ein Teil der generierten Dateien sind Launch-Dateien [25], die zum gleichzeitigen Starten mehrerer ROS-Nodes dienen. Unter ihnen befindet sich die Datei `demo.launch`. Mit der Datei werden die notwendigen MoveIt!-Nodes und RVIZ gestartet. In dieser Umgebung, die eine reine Simulation darstellt, kann erstmalig geprüft werden, ob alle bisher vorgenommenen Einstellungen korrekt waren.

Bei der Untersuchung der `demo.launch` fällt die folgende Codezeile besonders auf.

```
<arg name="fake_execution"value="true/>
```

Der Eintrag trägt dafür Sorge, dass die Steuerbefehle an einen `fake_controller` weitergeleitet werden. Dieser erlaubt jedoch keine Ausführung der Befehle am Roboterarm an sich. Daher ist es notwendig, einen eigenen Controller zu programmieren, was zum folgenden Abschnitt führt.

4.1.4 Controller Manager

Die Aufgabe des Controller Managers besteht darin, beim Start von MoveIt!, die `move_group` darüber zu informieren, welche Action Server implementiert sind. Hierfür muss im Unterordner `/config` eine Datei mit dem Namen `controllers.yaml` angelegt werden. Sie beinhaltet eine Liste der Controller, die für die Steuerung des Roboters notwendig sind. In Listing 4.2 ist die für diese Arbeit genutzte Controllerliste abgebildet.

Damit der Inhalt der `controllers.yaml` in den Parameterserver von ROS geladen werden kann, muss die Datei `rob_arm_small_moveit_contoller_manager.launch` gemäß Listing 4.1 ergänzt werden.

```
<launch>
        <arg name="moveit_controller_manager"
                default="moveit_simple_controller_manager/MoveItSimpleControllerManager" />

        <param name="moveit_controller_manager" value="$(arg moveit_controller_manager)" />

        <rosparam file="$(find rob_arm_small)/config/controllers.yaml"/>
</launch>
```

Listing 4.1: rob_arm_small_moveit_controller_manager.launch

Die Definition eines Controllers, innerhalb der `controllers.yaml`, erfolgt nach einem definierten Schema, welches nachfolgend genauer erläutert wird.

name: Hier wird der Name des Controllers angegeben.

action_ns: Hier wird der Namespace des Action Server angegeben.

type: An dieser Stelle ist festzulegen, welcher Art der Action Server ist. Für die Ausführung der MoveIt!-Steuerkommandos muss es sich um den Typ `FollowJointTrajectory` handeln.

default: Mit diesem Flag wird angegeben, ob es sich bei diesem Controller um den Standard-Controller handelt.

joints: An dieser Stelle werden alle Joints angegeben, die für die Ausführung der jeweiligen Kommandos notwendig sind.

```
1  controller_list:
2  - name: "rob_arm_small_follow_joint_trajectory_controller"
3    action_ns: joint_trajectory_action
4    type: FollowJointTrajectory
5    default: true
6    joints: [base_pan, shoulder_tilt, elbow_tilt]
7  - name: "rob_arm_small_gripper_controller"
8    action_ns: gripper_trajectory_action
9    type: FollowJointTrajectory
10   default: true
11   joints: [wrist_tilt, wrist_rotate, z_gripper]
```

Listing 4.2: controllers.yaml

4.2 Action Server

Um die Steuerkommandos von MoveIt! auf dem Roboter ausführen zu können sind Action Server notwendig. MoveIt! instantiiert hierfür die Action Clients, die in der `controllers.yaml` definiert wurden. Daraus ergibt sich auch die Forderung, dass der Name bei der Registrierung des Action Servers identisch mit dem in der `controllers.yaml` sein muss (vgl. Listing 4.2 Zeile 2 und Listing A.2 Zeile 92). Der Nachrichtentyp, der in den Action Servern verarbeitet wird bzw. den MoveIt! verschickt, werden über das `type-Tag`in der Definition festgelegt.

In dieser Arbeit erwies es sich als sinnvoll, für den Roboterarm sowie den Gripper den Type `FollowJointTrajectory` zu verwenden. Dieser Nachrichtentyp enthält die Steuerkommandos für die jeweiligen Joints, in Form von Winkelwerten. Aufgrund dessen können diese direkt an den Mikrokontroller weitergeleitet werden.

Follow Joint Trajectory-Action Server

Der erste der beiden Action Server dient der Ansteuerung des Roboterarms. Das bedeutet, dass alle Gelenke angesteuert werden können. Dieser Zusammenhang ergibt sich aus der Definition der Planungsgruppen. In dieser Arbeit wird der Roboter via Ethernet angesteuert. Daraus folgt, dass der hier implementierte Action Server die Steuerinformationen der `move_group` in das passende Format für das Hardware Interface umwandelt und an dieses weiterleitet, bevor die Ausführung stattfindet. Je nach Systemaufbau kann die Ansteuerung des Roboters auch unmittelbar aus dem Action Server erfolgen.

Die `move_group` errechnet einen Pfad, zu einem vorgegebenen Ziel. Aufgrund der Planungseigenschaften von MoveIt! ergeben sich mehrere Zwischenziele, die zu erreichen sind, da beispielsweise ein Hindernis umfahren werden muss. Das bedeutet, der Roboterarm fährt nicht auf direktem Weg zur gewünschten Position, sondern folgt einer vorher berechneten Bahnkurve. Durch die Berechnung der Vorwärts- / Rückwärtskinematik, ergeben sich für die einzelnen Zwischenpositionen definierte Winkelwerte der Joints. Nach erfolgreicher Berechnung des Pfades wird eine Nachricht generiert, die den gesamten Pfad zum Ziel sowie dessen Zwischenpositionen, in Form von Winkelwerten, enthält. Das Nachrichtenformat `JointTrajectory Message` [6] und dessen Verarbeitung innerhalb des Action Servers wird nachfolgend näher erläutert.

In Listing 4.3 ist der Aufbau der `JointTrajectory Message` beschrieben. In ihr enthalten ist der Nachrichtentyp `JointTrajectoryPoint Message` [7], der in Listing 4.4 zusätzlich abgebildet ist.

```
1 std_msgs/Header header
2 string[] joint_names
3 trajectory_msgs/JointTrajectoryPoint[] points
```

Listing 4.3: JointTrajectory Message

```
1 float64[] positions
2 float64[] velocities
3 float64[] accelerations
4 float64[] effort
5 duration time_from_start
```

Listing 4.4: JointTrajectoryPoint Message

Wie in Listing 4.3 ersichtlich, beinhaltet die `JointTrajectory-Message` ein Array von Point-Nachrichten. Ein Arrayeintrag entspricht hierbei einer Zwischenposition, die der Arm auf dem Weg zum Ziel einnehmen muss. Innerhalb der Point-Nachricht sind die Winkelwerte der Joints in einem Array `positions[]` abgespeichert. Die Anzahl der Einträge hängt von der Definition der Joints innerhalb der `controllers.yaml` ab.

Der grundsätzliche Aufbau eines Action Servers wurde bereits im Kapitel 3.4 beschrieben. Daher wird an dieser Stelle nur die Verarbeitung des Ziels innerhalb der Feedback-Callback-Methode, anhand des Listings 4.5, näher erläutert. Das vollständige Listing befindet sich im Anhang unter A.2.

```
57  void feedbackCB(const sensor_msgs::JointState &fback) {
58    rob_arm_small::UDPconnectionMsg tmp;
59    tmp.data.resize(sizeof(robotArmOnlyMsg_t));
60
61    if (!as_.isActive()) {
62      return;
63    }
64
65    if (goal_.points.size() > 0) {
66      conv[0] = goal_.points[0].positions[0];
67      conv[1] = goal_.points[0].positions[1];
68      conv[2] = goal_.points[0].positions[2];
69
70      tmp.Message_Id = MSG_RCP_ROBARM;
71      tmp.Message_IDext = MSGEXT_JOINT_ARM_INFO;
72      tmp.bytes_len = sizeof(robotArmOnlyMsg_t);
73      memcpy(&tmp.data[0], &conv[0], sizeof(robotArmOnlyMsg_t));
74
75      pub_udp_.publish(tmp);
76      goal_.points.erase(goal_.points.begin());
77
78      if (goal_.points.size() == 0) {
79        //Set action state to succeeded
80        ROS_INFO("FollowJointTrajectory Action Server: SUCCEEDED!");
81        as_.setSucceeded();
82      }
83    }
84  }
```

Listing 4.5: Feedback-Callback-Methode des FollowJointTrajectory Action-Servers

Die aktuellen Winkelpositionen des Roboters werden alle 50ms auf den Topic `/joint_states` gepublished. Die `Feedback-Callback-Methode` des Action Server wird durch diesen Topic getriggert. Zunächst wird geprüft, ob aktuell ein Ziel bearbeitet wird und sich somit der Action Server im Zustand `active` befindet. Ist dies nicht der Fall, wird die Methode umgehend verlassen und es wird auf das nächste Eintreffen einer Nachricht auf dem `/joint_state`-Topic gewartet.

Liegt stattdessen ein Ziel vor, wird im nächsten Schritt die Länge des Pfades geprüft. Wie im oberen Abschnitt erwähnt, beinhaltet jeder Points-Eintrag eine Zwischenposition, wobei der letzte Eintrag die eigentliche Zielposition repräsentiert. Das nächste Zwischenziel des Pfades ist hierbei im Eintrag `points[0]` abgelegt.

Solange noch Zwischenziele vorhanden sind, das heißt die Länge des Arrays größer Null ist, können die Positionswerte übernommen und verarbeitet werden. Sobald keine Einträge mehr vorhanden sind, bedeutet das, dass das Ziel erreicht wurde und der Status des Action Servers kann auf `succeeded` geändert werden (vgl. Zeile 81 Listing 4.5).

Wie in Listing 4.4 ersichtlich ist, nutzt ROS den Datentyp `float64` was dem Datentyp `double` entspricht. Der Mikrocontroller kann jedoch nur Datentypen von maximal 32Bit Größe verarbeiten. Daher müsssen die Daten in ein für den Mikrocontroller verarbeitbares Format umgewandelt werden. Dies Geschieht durch das Speichern der Positionen in das Float-Array `conv[]`.

Für die Übertragung der umgewandelten Daten dient ein institutsinternes Nachrichtenprotokoll. Die, in der `ETTI4RCPmsg.h` definierte, Nachricht `UDPconnectionMsg` wird nach der Konvertierung befüllt und über den Topic `/UDPCommand` an das Hardware Interface `hw_interface_node` übertragen, das die Winkelpositionen anschließend über das Netzwerk an den Roboter versendet.

Da, wie oben erwähnt, das nächste Zwischenziel in `points[0]` gespeichert ist, kann dieser Array-Eintrag nach dem Verschicken gelöscht werden, damit das nächste Ziel auf die Position Null nachrücken kann.

Da in der `controllers.yaml` nicht alle Joints des Roboters für diesen Action Server definiert wurden, wird zudem bei der Bewegungsplanung der Gripper Command Action Server aktiv.

Gripper Command - Action Server

Der Gripper Command Action Server dient ausschließlich zur Steuerung des Greifers. Er ermöglicht das Nicken, Rollen sowie Zugreifen des Grippers, unabhängig der Bewegung des Roboterarms. Das bedeutet, dass die eigentliche Position des Arms unverändert bleibt, während der Greifer innerhalb seiner Grenzen bewegt werden kann. Sollte die Bewegungsplanung vorsehen, dass der Greifer für das Erreichen einer Position geneigt beziehungsweise gedreht werden muss, wird der Gripper Command Action Server zusätzlich zum Follow Joint Trajectory - Action Server getriggert.

Die Verarbeitung des Ziels erfolgt analog zu der des Follow Joint Trajectory - Action Servers, da beide denselben Typ aufweisen und somit auch der Nachrichtentyp identisch ist. Im Anhang ist zum Vergleich das Listing des Gripper Command Action Servers unter A.3 zu finden.

4.3 Hardware Interface Node

Für die endgültige Kommunikation zwischen ROS und dem Roboterarm ist ein Hardware Interface notwendig. Diese Schnittstelle, in Form einer ROS-Node, kommt in mehreren Robotern innerhalb der WE4 zur Anwendung und wurde für diese Arbeit von Herrn Dipl. Ing. Prodromos Sotiriadis bereitgestellt. Sie besitzt vier wesentliche Aufgaben.

Empfangen der Nachrichten der Action Server: Das Hardware Interface subscribed den Topic `/UDPcommand` und wartet auf eintreffende Nachrichten der Action Server.

Versenden der Nachrichten via UDP: Eingehende Nachrichten der Action Server werden als UDP-Pakete an die Zieladresse des Roboters versendet. Die IP-Adresse des Mikrocontrollers wird in der zugehörigen Header-Datei angegeben.

Empfang der aktuellen Winkelwerte des Roboters: Das Hardware Interface empfängt auf den Socket 5550 eingehende UDP-Pakete des Roboters. Diese enthalten den aktuellen Status der Joints.

Publishen der Winkelwerte: Die eingehenden Winkelwerte werden in eine `JointState`-Variable [13] geschrieben und anschließend über den Topic `/joint_states` gepublished.

Neben den genannten Hauptaufgaben wurde die Schnittstelle, um die Möglichkeit den Roboter unabhängig des MoveIt!-Frameworks anzusteuern, erweitert. Dies dient vornehmlich zu Debug- und Kalibrierungszwecken. Die Node stellt den Topic `/manual_control` zur Verfügung. Dieser verarbeitet `JointState - Nachrichten` und leitet diese unmittelbar über UDP an den Mikrocontroller weiter. Hieraus ergibt sich die Möglichkeit, über das Tool `rqt_gui` Nachrichten zu erzeugen und den Roboterarm unabhängig von MoveIt! zu steuern. Gestartet wird das Tool mit folgendem Aufruf.

```
$ rosrun rqt_gui rqt_gui
```

Durch die Implementierung des Controller-Managers, der Action-Server und dem Hardware Interface ist zunächst keine Steuerung des Roboterarmes möglich, da das Robotermodell noch nicht in den ROS-Parameterserver geladen wird. Hierfür müssen in der Datei `move_group.launch` folgende Zeilen aus Listing 4.6 ergänzt werden.

```
<include file="$(find rob_arm_small)/launch/planning_context.launch" >
        <arg name="load_robot_description" value="true" />
</include>
```

Listing 4.6: Ergänzung der move_group.launch

Noch lassen sich keine vom Nutzer definierten Ziele vorgeben. Über das MoveIt!-RVIZ-Plugin können jedoch Zufallsziele generiert, geplant und ausgeführt werden. Dies erlaubt eine Kontrolle der im Mikrocontroller programmierten Werte für Umrechnungsfaktoren zwischen Pulsweite und Winkel.

$$V = \frac{Pulsweitenänderung}{Winkeländerung} = \frac{x[\mu s]}{1[°]}$$

Um diese Faktoren genauer bestimmen zu können, wurden die `goal-Messages` ausgelesen und die Winkelwerte der Zielposition ermittelt. Nach Erreichen der Endposition wurden die Soll-Winkelwerte mit den tatsächlichen Winkeln des Roboters verglichen und bei Abweichungen eine Messreihe durchgeführt, um die Umrechnungsfaktoren zu korrigieren.

4.4 Bewegungsplanung mit „move_group_interface“

Die Realisierung bis zum jetzigen Zeitpunkt erlaubt, wie oben beschrieben, bisher ausschließlich das Vorgeben von Zufallszielen über das RVIZ-MoveIt!-Plugin. Aus diesem Grund muss das System um die Möglichkeit vom Nutzer definierte Ziele anzusteuern zu können erweitert werden. Hierfür dient das “C++ User Interface“, das im Abschnitt 3.2 bereits erwähnt wurde.

Das Plugin ermöglicht dem Programmierer, Ziele auf verschiedene Art und Weisen zu definieren. So besteht die Möglichkeit, ein “`JointSpaceGoal`“ zu erzeugen. Bei diesem werden die Winkelwerte der Joints des Roboters vorgegeben und anschließend übernommen werden. Ebenso können Ziele in Form einer definierten Pose (`PoseTarget`) angegeben werden. Diese beschreibt einerseits die Position im Raum sowie die Ausrichtung des Endeffektors. Des Weiteren kann ein Ziel durch kartesische Koordinaten (`PositionTarget`) beschrieben werden. In diesem Fall entfällt die Angabe der Orientierung. Nachfolgend werden die unterschiedlichen Möglichkeiten der Zielvorgabe erläutert, wobei die Angabe eines `PositionTarget` die aktuelle Implementierung darstellt und daher ausführlicher betrachtet wird.

PositionTargets

Die Vorgabe eines `PositionTarget` zeichnet sich dadurch aus, dass für den Endeffektor ein Ziel, in Form einer kartesischen Koordinate, im Raum angegeben wird. Anhand des Listings A.4 wird diese Form der Implementierung genauer beschrieben.

Die im Abschnitt 4.1.3 definierten Planungsgruppen sowie Endeffektoren bilden die Grundlage für die Berechnungen des Motion Planners. Da in dieser Arbeit die Planungsgruppen `roboter_arm` und `gripper` definiert wurden und die Bewegungsplanung für den gesamten Roboterarm berechnet werden soll, wird dem `MoveGroupInterface` die Gruppe `roboter_arm` übergeben (vgl. Listing 4.7). Aus dieser Information wählt der MotionPlanner automatisch den zugehörigen Endeffektor, in diesem Beispiel den Link `gripper_left_rear`.

```
10  static const std::string PLANNING_GROUP = "roboter_arm";
11  moveit::planning_interface::MoveGroupInterface move_group(PLANNING_GROUP);
```

Listing 4.7: Auswahl von Planungsgruppe

In Listing 4.8 ist dargestellt, wie dem `MoveGroupInterface` ein definierter Start zugewiesen werden kann. Das beinhaltet das Löschen etwaiger vorhandener Ziele, die Annahme der aktuellen Position als Startpose sowie das Nennen des Referenzkoordinatensystems - an dieser Stelle des `base_link`.

```
17  move_group.clearPoseTargets();
18  move_group.setStartStateToCurrentState();
19  move_group.setPoseReferenceFrame("base_link");
```

Listing 4.8: Festlegen definierter Startposition

Des Weiteren können die Toleranzen für die Zielerreichung sowie die Planungszeit skaliert werden (vgl. Listing 4.9).

```
21  move_group.setGoalTolerance(0.005);
22  move_group.setPlanningTime(20);
```

Listing 4.9: Skalieren der Zieltoleranz und Planungszeit

Neben dem in Listing 4.9 genannten Befehl `<group>.setGoalTolerance()` besteht die Möglichkeit, die Toleranzen für die Zielerreichung, genauer zu definieren. Dies ist im Wesentlichen davon abhängig, welche Art von Zielvorgabe der Programmierer wählt. Für den Fall, dass ein `JointSpaceGoal` vorgegeben wird, kann über die Funktion `<group>.setGoalJointTolerance()` die maximale Abweichung der Winkelwerte eingestellt werden. Wird hingegen ein `PoseTarget` vorgegeben können die Funktionen `<group>.setGoalPositionTolerance()` und `<group>.setGoalOrientationTolerance()` die maximal zulässigen Abweichungen für die Position sowie die Orientierung des Endeffektors angegeben werden. Weitere Informationen diesbezüglich sind der API [15] oder dem Tutorial [14] zu entnehmen.

Die Zielvorgabe und deren Ausführung ist in Listing 4.10 dargestellt. In dieser Implementierung werden die Koordinaten, in Form eines transponierten Spaltenvektors, als Argumente beim Programmaufruf angegeben.

$$\vec{v} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \implies \vec{v}\,(x\ y\ z)\top$$

Daraus ergibt sich folgender, beispielhafter Programmaufruf, der fordert, dass der Endeffektor sich an den Punkt mit den Koordinaten $(0.13[m]\,|\,0.11[m]\,|\,0.32[m])$ bewegen soll.

```
$ rosrun rob_arm_small user_interface_node 0.13 0.11 0.32
```

Da die Argumente als C-String interpretiert werden, müssen diese mit der Funktion `atof(double const char* str)` in double-Werte umgewandelt werden. Nur auf diese Weise erfolgt eine korrekte Übernahme der Zielkoordinaten (vgl. Listing 4.10).

Im darauf folgenden Aufruf `<group>.move()` werden die Berechnungen des Motion Planners gestartet. Dieser prüft, ob die angegebene Position erreichbar ist und berechnet anschließend einen kollisionsfreien Pfad zum Ziel, der über den `FollowJointTrajectory - Action Client` als `goal-Message` an den Action Server zur Ausführung übertragen wird.

```
24  move_group.setPositionTarget(atof(argv[1]), atof(argv[2]), atof(argv[3]), move_group.
        getEndEffectorLink());
```

Listing 4.10: Setzen des Zieles und Ausführung

Joint Space Goals

Joint Space Goals sind Ziele, bei denen Winkel für einen oder mehrere Joints vorgegeben werden können. Dies dient beispielsweise der Drehung des Roboters in der Z-Achse, ohne die Stellung des Arms dabei zu verändern. Die Generierung eines solchen Zieles wird anhand des Listings 4.11 erläutert.

In Zeile 1 des Listings 4.11 wird der aktuelle Status des Roboters in der Variablen `current_state` gespeichert. Anschließend wird aus der Statusvariablen die Stellung der einzelnen Joints in den Vektor `joint_group_positions` gespeichert.

Über die Änderung der Werte innerhalb des Vektors können den jeweiligen Joints neue Werte zugeordnet werden. Über den in Zeile 8 gezeigten Befehl wird das neue Ziel an den Motion Planner übertragen. Mit dem Befehl `move_group.move()` kann die Ausführung angestoßen werden.

```
1  moveit::core::RobotStatePtr current_state = move_group.getCurrentState();
2
3  std::vector<double> joint_group_positions;
4  current_state->copyJointGroupPositions(joint_model_group, joint_group_positions);
5
6  joint_group_positions[0] = -0.785;  //move Joint 'base_pan' to -45°
7
8  move_group.setJointValueTarget(joint_group_positions);
9  move_group.move();
```

Listing 4.11: Erstellen eines Joint Space Goals

Pose Targets

Eine weitere Möglichkeit, Ziele zu generieren, sind sogenannte `Pose Targets`. Diese enthalten zum einen die Position im Raum und zum anderen die Orientierung des Grippers, in Form eines Quaternions. Die Erstellung und Übertragung eines solchen Ziels wird nachfolgend anhand des Listings 4.12 erläutert.

Damit ein Ziel in Form einer Pose vorgegeben werden kann, muss im ersten Schritt eine Variable vom Typ `PoseStamped` angelegt werden (vgl. Zeile 1 Listing 4.12). Nach dem Anlegen der Variablen kann diese mit den notwendigen Informationen beschrieben werden. Dies erfolgt in den Zeilen 3 bis 7. Die Werte für die `.pose.position` stellen hierbei die Koordinate im Raum dar. Die Orientierung wird in diesem Beispiel durch die Berechnung eines Quaternions aus den Winkeln der gewünschten Zielorientierung angegeben.

```
1  geometry_msgs::PoseStamped goal_pose;
2
3  goal_pose.header.frame_id = "base_link";
4  goal_pose.pose.position.x = 0.0;
5  goal_pose.pose.position.y = 0.0;
6  goal_pose.pose.position.z = 0.0;
7  goal_pose.pose.orientation = tf::createQuaternionMsgFromRollPitchYaw(0.00, 1.57, 0.00);
8
9  move_group.setPoseTarget(goal_pose);
10
11 move_group.move();
```

Listing 4.12: Erstellen eines Pose Goals

4.5 Gesamtüberblick

Die Implementierung einer Steuerung für einen Roboterarm besteht, wie ersichtlich, aus einer Vielzahl verschiedener Komponenten. Ihre Aufgaben reichen dabei von der Verarbeitung von Daten, bis hin zur Vernetzung unterschiedlicher Komponenten.

Aufgrund der zum Teil oberflächlichen MoveIt!-Dokumentation und Tutorials war das Identifizieren von notwendigen Komponenten und deren Implementierungen nicht immer sofort ersichtlich, sodass bei nahezu jedem Entwicklungsschritt eine umfassende Recherche erforderlich war. Problematisch an dieser Stelle ist, dass viele der vorhanden Beispiele für ältere ROS-Versionen programmiert wurden und dementsprechend einige Funktionsaufrufe oder Klassen nicht mehr in der genutzten Version “Kinetic“ vorhanden sind. Insbesondere die Problematik der Ansteuerung eines realen Roboters stellte Anfangs eine Herausforderung dar, die nach langer Recherche, durch einen Foreneintrag der ROS-Community [10], bewältigt werden konnte.

Des Weiteren sind in der MoveIt!-Dokumentation kaum Aussagen über die Kommunikation der einzelnen Systemkomponenten zu finden. Daher sind in Abbildung 4.10 die Zusammenhänge der Komponenten in Form eines Schichtenmodells dargestellt, das nachfolgend genauer betrachtet wird. Beginnend bei der Vorgabe eines Ziels, wird der Weg durch die einzelnen Schichten bis zur Ausführung am Roboter sowie der Rückweg für die Darstellung und Rückmeldungen beschrieben.

Eine Zielvorgabe kann in der aktuellen Form der Implementierung über zwei Möglichkeiten geschehen: Zum einen ist das die Vorgabe eines Zufallsziels durch das MoveIt!-RVIZ-Plugin und zum anderen die Angabe einer Zielposition durch das MoveIt!-User-Interface. Der Zugriff auf den Motion Planner sowie die Übertragung der Zieldaten erfolgt durch Funktionsaufrufe aus den Programmen RVIZ bzw. dem User-Interface. Sobald MoveIt! alle notwendigen Daten gesammelt und einen Pfad errechnet hat, werden diese in Form einer `goal-Message` an die jeweiligen Action Server übertragen. Der Austausch dieser Daten erfolgt durch die in Abschnitt 3.4 beschriebenen Action Clients und - Server. Die Action Server werten das erhaltene Ziel aus und übertragen, in Verbindung mit der Hardware-Interface-Node, die Positionswerte der Servos über das Netzwerk an den Mikrocontroller. Im Mikrocontroller wiederum werden die Ziele in PWM-Signale umgerechnet und an den Timer-Pins dem Roboter für die einzelnen Servos zur Verfügung gestellt. Zeitgleich führt der Mikrocontroller die Berechnung der aktuellen Position jedes Servos durch und versendet diese gesammelt als `JointState-Message` über das Netzwerk zurück an den `/joint_states - Topic`. Dieser Topic dient einerseits zum Triggern der Feedback-Callback-Funktionen in den Action Servern und andererseits werden die Informationen des Topics genutzt, um die aktuelle Position des Roboters in RVIZ darzustellen.

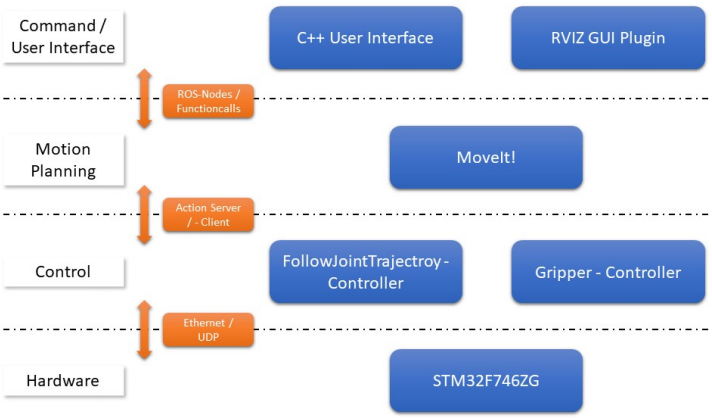


Abbildung 4.10: ROS Stack

Starten aller Systemkomponenten

Für die Steuerung des Roboterarms müssen die implementierten Komponenten gestartet werden. Dabei spielt die Reihenfolge kaum eine Rolle, solange der Start aller Komponenten zeitnah geschieht, da die `move_group` bei ihrem Start prüft, ob die Action Server bereits gestartet wurden. Ist dies nicht der Fall, wird dies im Terminalfenster angezeigt. Sollten die Controller nicht oder zu spät gestartet werden, fährt MoveIt! ohne die Verwendung der Action Server fort. Daher kann ein Neustart der `move_group` erforderlich sein, sollten diese benötigt werden.

Zuerst muss die Kernkomponente von MoveIt! gestartet werden. Dies geschieht mit folgendem Befehl.

```
$ roslaunch rob_arm_small move_group.launch
```

Da die `move_group-Node` beim Start prüft, welche Controller implementiert sind, ist es erfordlich, die Actionserver für die Steuerung des Roboterarms sowie des Grippers zu starten. Hierfür müssen die folgenden Kommandos, in jeweils einem Terminalfenster, eingegeben werden.

```
$ rosrun rob_arm_small_hw_interface action_server
```

```
$ rosrun rob_arm_small_hw_interface grip_command_server
```

Damit eine Kommunikation mit dem Roboter erfolgt, ist ebenfalls das Hardware Interface zu starten. Hierzu muss der nachfolgende Befehl ausgeführt werden.

```
$ rosrun rob_arm_small_hw_interface hw_interface_node
```

Der Start der Hardware-Interface-Node sowie der Action Server wurde in einem Launch-File zusammengefasst. Ausgeführt werden kann dieses durch folgenden Befehl. Hierdurch entfällt das Starten der einzelnen Komponenten.

```
$ roslaunch rob_arm_small_hw_interface hardware_interface.launch
```

Nach dem Start dieser Komponenten ist der Roboter bereit, Befehle zu empfangen und auszuführen. Hierzu kann zum einen RVIZ genutzt werden, was gleichzeitig eine Visualisierung ermöglicht, oder zum anderen das User Interface, um ein Ziel vorzugeben. Das Ausführen des User Interface wurde bereits in Abschnitt 4.4 behandelt. Zum Starten von RVIZ mit dem MoveIt!-Plugin ist die Ausführung des folgenden Befehls notwendig.

```
$ roslaunch rob_arm_small moveit_rviz.launch
```

5 Diskussion

In dieser Arbeit wurde erstmalig eine grundlegende Steuerung für einen Roboterarm mit dem Robot Operating System erfolgreich implementiert.

In den folgenden Abschnitten sollen die Herausforderungen und Schwierigkeiten der Implementierung sowie eine Analyse der Resultate dargestellt werden. Auf Basis der Ergebnisanalyse und Erfahrungen dieser Arbeit werden anschließend die Möglichkeiten von weiteren Implementierungsschritten sowie Folgeprojekten aufgezeigt.

5.1 Zusammenfassung

Ziel dieser Thesis war die Implementierung einer Steuerung für einen Roboterarm mit ROS. Aufgaben einer Steuerung beinhalten die Pfadplanung, Kollisionsvermeidung sowie das Manipulieren von Objekten. Zusätzlich soll diese Arbeit eine Grundlage für weitere Projekte im Bereich mobiler Manipulation und dem Arbeiten mit MoveIt! darstellen. Wie in Abschnitt 4.5 erwähnt, war die erste Schwierigkeit das Identifizieren notwendiger Komponenten für die Ansteuerung eines realen Roboters. Dieses Problem konnte durch umfangreiche Recherchen erfolgreich gelöst werden, sodass die Implementierung im Wesentlichen als Erfolg bezeichnet werden kann.

Weitaus schwieriger stellte sich die Problematik der Implementierung des User Interface dar. Die Ansätze der in den Tutorials genannten Implementierungen waren anfangs nicht auf den genutzten Roboter umsetzbar. Im Wesentlichen bestand der Ansatz darin, ein `PoseGoal` vorzugeben (siehe Abschnitt 4.4). Diese Art von Ziel zeichnet sich zum einen aus einer Position im Raum und zum anderen aus einer Orientierung, in Form eines Quaternions aus. Hier war es möglich die Pose eines Zufallszieles auszulesen und anschließend die entnommenen Werte wieder als Ziel vorzugeben. Die Eingabe eines frei gewählten Ziels wurde jedoch stets mit Fehlermeldungen quittiert. Dieses Problem war auf die unvorteilhafte Wahl der Planungsgruppen zurückzuführen. Anfänglich waren sämtliche Joints (aktive sowie passive) der Planungsgruppe `roboter_arm` zugewiesen, woraufhin die Spitze des Greifers als Endeffektor für die Planung ausgewählt wurde. In dieser Kombination war es dem Bewegungsplaner jedoch nicht möglich, einen gültigen Pfad zu berechnen. Erst durch das Entfernen der passiven Joints und daraus resultierend der Wahl des Link `gripper_left_rear` als Endeffektor konnten frei gewählte Ziele erreicht werden. Daraus resultierend ist es bei der aktuellen Implementierung ausreichend, ein Ziel in Form einer kartesischen Koordinate anzugeben, ohne eine Orientierung des Grippers vorgeben zu müssen. Eine weitere Erkenntnis, die bei der Recherche zu dieser Problematik erlangt

wurde, war, dass das RVIZ-Plugin Zufallsziele in Form von `JointSpaceGoals` generiert. Das bedeutet, dass RVIZ nicht die Position und Orientierung im Raum als Grundlage für die Zielgenerierung nutzt, sondern stattdessen die Winkel der einzelnen Joints für die Zielangabe.

Weitere Schwierigkeiten ergaben sich daraus, dass die Dokumentation von ROS und MoveIt! zum Teil sehr lückenhaft ist. Hier sei beispielhaft das Tutorial zur Controller Konfiguration [4] genannt. In dieser Anleitung wird angenommen, dass ein Action Service angeboten wird. Jedoch ist an keiner Stelle beschrieben, wie dieser zu implementieren ist.

Ein weiterer Punkt, der für Probleme bei der Implementierung sorgte, ist die Reihenfolge der Joints innerhalb der verschiedenen Nachrichtenformate. Während in den Nachrichten des `/joint_states Topic` die Joints alphabetisch geordnet sind, sind in den `goal Messages` der Action Server die Joints entsprechend der Reihenfolge der URDF-Datei angegeben. Diese Problematik erforderte eine genaue Analyse der versendeten bzw. erwarteten Daten.

Insgesamt ist das Ergebnis als zufriedenstellend zu betrachten und als Erfolg zu werten. Dass die Forderung “Manipulation von Objekten“ nicht umgesetzt werden konnte, war dem umfangreichen Rechercheaufwand, aufgrund der lückenhaften Dokumentation, geschuldet.

5.2 Ausblick

Die anfänglich gestellten Forderungen dieser Arbeit konnten nicht alle vollständig umgesetzt werden, sodass diese in zukünftigen Projekten weiter bearbeitet werden sollten. Dies bezieht sich zum einen auf die Implementierung eines `Pick and Place-Algorithmus`, der die Manipulation von Objekten ermöglicht. Zum anderen die Implementierung von bildverarbeitenden Elementen. Hierbei reichen die Möglichkeiten von der Nutzung einer einfachen Kamera für Mustererkennungsalgorithmen, bis hin zur Anbindung von Stereokameras die, in Verbindung mit dem Package `Octomap`, eine dreidimensionale Umgebungskarte erzeugen können.

Neben weiterführenden Projekten bieten sich auch Verbesserungen der aktuellen Implementierungen an. Zum einen sollte das aktuell genutzte URDF-Modell, welches eine grobe Näherung darstellt, durch ein präziseres ersetzt werden. An dieser Stelle sollte versucht werden, das Modell so zu gestalten, dass der Endeffektor, der für die Berechnung der Zielposition dient, mittig zwischen den Greiferklammern liegt. Durch diesen Schritt könnte eine präzisere Steuerung erreicht und fehlerhafte Berechnungen minimiert werden.

Des Weiteren stellt MoveIt!, das in Abschnitt 3.2 erwähnte, `IKFastPlugin` zur Verfügung. Mit diesem lässt sich ein Algorithmus implementieren, der eine schnellere Berechnung der

Vorwärts- / Rückwärtskinematik ermöglicht. Dadurch können Planungszeiten um ein Vielfaches der aktuellen Berechnungszeit verkürzt werden.

Neben den bereits genannten Projekten ist noch die Hardwareanbindung zu nennen. Aktuell erfolgt die Kommunikation im Netzwerk über das verbindungslose UDP-Protokoll. Hierbei werden empfangende Pakete nicht bestätigt, was wiederum bedeutet, dass dem Sender keinerlei Informationen darüber zur Verfügung stehen, ob ein Paket tatsächlich beim Empfänger angekommen ist. Dies ist insbesondere bei der Übertragung der aktuellen Roboterposition problematisch, da nicht davon ausgegangen werden kann, dass die aktuellen Daten die verarbeitet werden auch der aktuellen Position entsprechen.

Zur Lösung dieses Problems würde sich der Einsatz des verbindungsorientierten TCP-Protokolls anbieten. Bei diesem Protokoll wird jedes Paket quittiert und im Falle eines Verlusts die Übertragung wiederholt.

Literaturverzeichnis

[1] *actionlib Mainpage*. 31. Mai 2018. URL: http://wiki.ros.org/actionlib.

[2] *actionlib Tutorials*. 31. Mai 2018. URL: http://wiki.ros.org/actionlib/Tutorials.

[3] **Waldner Christian**: „Konzept für die Steuerung eines Roboterarms mit ROS“. Projektarbeit. WE4 ETTI Universität der Bundeswehr, 2. März 2018.

[4] *Controllers Configuration Tutorial*. 16. Juni 2018. URL: http://docs.ros.org/kinetic/api/moveit_tutorials/html/doc/controller_configuration/controller_configuration_tutorial.html.

[5] *Creating a workspace for catkin*. 10. Juni 2018. URL: http://wiki.ros.org/catkin/Tutorials/create_a_workspace.

[6] *Definition der JointTrajectory Message*. 9. Juni 2018. URL: http://docs.ros.org/api/trajectory_msgs/html/msg/JointTrajectory.html.

[7] *Definition der JointTrajectoryPoint Message*. 9. Juni 2018. URL: http://docs.ros.org/api/trajectory_msgs/html/msg/JointTrajectoryPoint.html.

[8] *Detailierte Dokumentation: Paket actionlib*. 31. Mai 2018. URL: http://wiki.ros.org/action/fullsearch/actionlib/DetailedDescription?action=fullsearch&context=180&value=linkto%3A%22actionlib%2FDetailedDescription%22.

[9] *IKFast Plugin*. 19. Juni 2018. URL: http://docs.ros.org/kinetic/api/moveit_tutorials/html/doc/ikfast/ikfast_tutorial.html.

[10] *Implement MoveIt! on Real Robot*. 9. Juni 2018. URL: https://answers.ros.org/question/192739/implement-moveit-on-real-robot/.

[11] *Installation von ROS*. 8. Juni 2018. URL: http://wiki.ros.org/kinetic/Installation/Ubuntu.

[12] *Joint State Publisher*. 18. Juni 2018. URL: http://wiki.ros.org/joint_state_publisher.

[13] *JointState Message*. 9. Juni 2018. URL: http://docs.ros.org/kinetic/api/sensor_msgs/html/msg/JointState.html.

[14] *Move Group C++ Interface*. 16. Juni 2018. URL: http://docs.ros.org/kinetic/api/moveit_tutorials/html/doc/move_group_interface/move_group_interface_tutorial.html.

[15] *MoveGroup Class Reference*. 11. Juni 2018. URL: http://docs.ros.org/jade/api/moveit_ros_planning_interface/html/classmoveit_1_1planning__interface_1_1MoveGroup.html.

[16] *MoveIt! Motion Planning Framework*. 17. Juni 2018. URL: moveit.ros.org.

[17] *MoveIt! Tutorial Homepage*. 8. Juni 2018. URL: http://docs.ros.org/kinetic/api/moveit_tutorials/html/.

[18] *OctoMap*. 19. Juni 2018. URL: https://octomap.github.io/.

[19] *Orocos Kinematics and Dynamics*. 17. Juni 2018. URL: http://www.orocos.org/kdl.

[20] *Pflege-Roboter sollen in Pflegheim in Garmisch-Patenkirchen getestet werden*. 16. Juni 2018. URL: https://www.br.de/nachrichten/pflege-roboter-sollen-in-pflegheim-in-garmisch-patenkirchen-getestet-werden-100.html.

[21] *Robot Operating System*. 17. Juni 2018. URL: www.ros.org.

[22] *Robot State Publisher*. 18. Juni 2018. URL: http://wiki.ros.org/robot_state_publisher.

[23] *ROS TF-Bibliothek*. 17. Juni 2018. URL: wiki.ros.org/tf.

[24] *ROS-Parameter Server*. 19. Juni 2018. URL: http://wiki.ros.org/Parameter%20Server.

[25] *roslaunch/XML*. 1. Juni 2018. URL: http://wiki.ros.org/roslaunch/XML.

[26] *Simple Action Server using the Goal Callback Method*. 31. Mai 2018. URL: http://wiki.ros.org/actionlib_tutorials/Tutorials/SimpleActionServer%28GoalCallbackMethod%29.

[27] *The general organisation of ROS*. 27. Mai 2018. URL: https://www.generationrobots.com/blog/en/2016/03/ros-robot-operating-system-2/.

[28] *The Open Motioin Planning Library*. 17. Juni 2018. URL: http://ompl.kavrakilab.org/.

[29] **Frank Uhl**: „Konzept für die Steuerung eines autonomen Roboterfahrzeugs mit ROS“. Projektarbeit. WE4 ETTI Universität der Bundeswehr, Apr. 2017.

[30] *XML Specifications for URDF*. 26. Mai 2018. URL: `http://wiki.ros.org/urdf/XML`.

Anhang

A Sourcecodes

```
#include <ros/ros.h>
#include <std_msgs/Float32.h>
#include <actionlib/server/simple_action_server.h>

  class ExampleAction{
  public:

    ExampleAction(std::string name) : as_(nh_, name, false), action_name_(name){
      //Register Callback-Methos
      as_.registerGoalCallback(boost::bind(&ExampleAction::goalCB, this));
      as_.registerPreemptCallback(boost::bind(&ExampleActionAction::preemptCB, this));
      sub_ = nh_.subscribe("/random_topic", 1, &ExampleAction::feedbackCB, this);
      as_.start();
    }

    ~ExampleAction(void){
    }

    void goalCB(){
      //Accept new goal
      goal_ = as_.acceptNewGoal()->samples; //The ->samples Statements depends on .action-
          File
    }

    void preemptCB(){
      ROS_INFO("%s: Preempted", action_name_.c_str());
      // set the action state to preempted
      as_.setPreempted();
    }

    void feedbackCB(const std_msgs::Float32::ConstPtr& msg){
      //Check if valid goal is active
      if (!as_.isActive()){
        return;
      }

      //Do something and publish periodic feedback
      as_.publishFeedback(feedback_);

      //Check if goal is satisfied
      if(data_ > goal_){
        //Set action state to aborted
        as_.setAborted(result_);
      }else if (data_ == goal_){
        //Set action state to succeeded
        as_.setSucceeded(result_);
      }
    }

    protected:
```

```
50
51    ros::NodeHandle nh_;
52    actionlib::SimpleActionServer<simple_action_server_example::ExampleAction> as_;
53
54    std::string action_name_;
55    int data_, goal_;
56    //Datatypes generated via ActionMessages
57    simple_action_server_example::ExampleFeedback feedback_;
58    simple_action_server_example::ExampleResult result_;
59    ros::Subscriber sub_;
60  };
61
62  int main(int argc, char** argv){
63    ros::init(argc, argv, "example_simple_actionserver");
64
65    ExampleAction example_simple_actionserver(ros::this_node::getName());
66    ros::spin();
67
68    return 0;
```

Listing A.1: Simple Action Server using Callback Method

```
#include <ros/ros.h>
#include <actionlib/server/simple_action_server.h>
#include <control_msgs/FollowJointTrajectoryAction.h>
#include <trajectory_msgs/JointTrajectory.h>
#include <sensor_msgs/JointState.h>

#include "../include/rob_arm_small/rob_arm_small_hw_interface.h"
#include "../include/rob_arm_small/ETTI4RCPmsg.h"

class RobotTrajectoryFollower {
protected:
  ros::NodeHandle nh_;

  // NodeHandle instance must be created before this line. Otherwise strange error may occur
      .
  actionlib::SimpleActionServer<control_msgs::FollowJointTrajectoryAction> as_;
  std::string action_name_;
  trajectory_msgs::JointTrajectory goal_;
  ros::Publisher pub_udp_;
  ros::Subscriber sub_;
  sensor_msgs::JointState jointStates;
  float conv [3];

public:

  RobotTrajectoryFollower(std::string name) :
      as_(nh_, name, false), action_name_(name) {
    pub_udp_ = nh_.advertise<rob_arm_small::UDPconnectionMsg>("/UDPcommand", 10);

    //Register callback functions:
    as_.registerGoalCallback(
        boost::bind(&RobotTrajectoryFollower::goalCB, this));
    as_.registerPreemptCallback(
        boost::bind(&RobotTrajectoryFollower::preemptCB, this));

    sub_ = nh_.subscribe("/joint_states", 1,
        &RobotTrajectoryFollower::feedbackCB, this);
    jointStates.position.resize(3);

    as_.start();
    ROS_INFO("FollowJointTrajectory action server started.");
  }

  ~RobotTrajectoryFollower(void) //Destructor
      {
  }

  void goalCB() {
    // Set action state to active
    goal_ = as_.acceptNewGoal()->trajectory;
  }

```

```
52    void preemptCB() {
53      // set the action state to preempted
54      as_.setPreempted();
55    }
56
57    void feedbackCB(const sensor_msgs::JointState &fback) {
58      rob_arm_small::UDPconnectionMsg tmp;
59      tmp.data.resize(sizeof(robotArmOnlyMsg_t));
60
61      if (!as_.isActive()) {
62        return;
63      }
64
65      if (goal_.points.size() > 0) {
66        conv[0] = goal_.points[0].positions[0];
67        conv[1] = goal_.points[0].positions[1];
68        conv[2] = goal_.points[0].positions[2];
69
70        tmp.Message_Id = MSG_RCP_ROBARM;
71        tmp.Message_IDext = MSGEXT_JOINT_ARM_INFO;
72        tmp.bytes_len = sizeof(robotArmOnlyMsg_t);
73        memcpy(&tmp.data[0], &conv[0], sizeof(robotArmOnlyMsg_t));
74
75        pub_udp_.publish(tmp);
76        goal_.points.erase(goal_.points.begin());
77
78        if (goal_.points.size() == 0) {
79          //Set action state to succeeded
80          ROS_INFO("FollowJointTrajectory Action Server: SUCCEEDED!");
81          as_.setSucceeded();
82        }
83      }
84    }
85  };
86
87
88  int main(int argc, char** argv) {
89    ros::init(argc, argv, "action_server");
90
91    RobotTrajectoryFollower RobotTrajectoryFollower(
92      "/rob_arm_small_follow_joint_trajectory_controller/joint_trajectory_action");
93    ros::spin();
94
95    return 0;
96  }
```

Listing A.2: action_server.cpp

```
#include "ros/ros.h"
#include <actionlib/server/simple_action_server.h>
#include <control_msgs/FollowJointTrajectoryAction.h>
#include <trajectory_msgs/JointTrajectory.h>
#include <sensor_msgs/JointState.h>

#include "../include/rob_arm_small/rob_arm_small_hw_interface.h"
#include "../include/rob_arm_small/ETTI4RCPmsg.h"

class GripperCmdAction {
protected:
  ros::NodeHandle nh_;

  // NodeHandle instance must be created before this line. Otherwise strange error may occur
      .
  actionlib::SimpleActionServer<control_msgs::FollowJointTrajectoryAction> gas_;
  std::string gripper_action_name;
  trajectory_msgs::JointTrajectory goal_;
  ros::Publisher pub_udp_;
  ros::Subscriber sub_;
  sensor_msgs::JointState jointStates;
  float conv[3];

public:

  GripperCmdAction(std::string name) :
      gas_(nh_, name, false), gripper_action_name(name) {
    pub_udp_ = nh_.advertise<rob_arm_small::UDPconnectionMsg>("/UDPcommand", 10);

    //Register callback functions:
    gas_.registerGoalCallback(
        boost::bind(&GripperCmdAction::goalCB, this));
    gas_.registerPreemptCallback(
        boost::bind(&GripperCmdAction::preemptCB, this));

    sub_ = nh_.subscribe("/joint_states", 1, &GripperCmdAction::feedbackCB,
        this);
    jointStates.position.resize(3);

    gas_.start();
    ROS_INFO("Gripper Action server started.");
  }

  ~GripperCmdAction(void) { //Destructor

  }

  void goalCB() {
    // Set action state to active
    goal_ = gas_.acceptNewGoal()->trajectory;
  }

```

```
52    void preemptCB() {
53      // set the action state to preempted
54      gas_.setPreempted();
55    }
56
57    void feedbackCB(const sensor_msgs::JointState &fback) {
58      rob_arm_small::UDPconnectionMsg tmp;
59      tmp.data.resize(sizeof(robotArmGripperMsg_t));
60
61      if (!gas_.isActive()) {
62        return;
63      }
64
65      if (goal_.points.size() > 0) {
66        conv[0] = goal_.points[0].positions[0];
67        conv[1] = goal_.points[0].positions[1];
68        conv[2] = goal_.points[0].positions[2];
69
70        tmp.Message_Id = MSG_RCP_ROBARM;
71        tmp.Message_IDext = MSGEXT_JOINT_GRIPPER_INFO;
72        tmp.bytes_len = sizeof(robotArmGripperMsg_t);
73        memcpy(&tmp.data[0], &conv[0], sizeof(robotArmGripperMsg_t));
74
75        pub_udp_.publish(tmp);
76        goal_.points.erase(goal_.points.begin());
77
78        if (goal_.points.size() == 0) {
79          //Set action state to succeeded
80          ROS_INFO("Gripper Action Server: SUCCEEDED!");
81          gas_.setSucceeded();
82        }
83      }
84    }
85  };
86
87
88  int main(int argc, char** argv) {
89    ros::init(argc, argv, "gripper_action_server");
90
91    GripperCmdAction GripperCmdAction(
92        "/rob_arm_small_gripper_controller/gripper_action");
93    ros::spin();
94
95    return 0;
96  }
```

Listing A.3: grip_command.cpp

```
#include <ros/ros.h>
#include <moveit/move_group_interface/move_group_interface.h>

int main(int argc, char** argv) {
  ros::init(argc, argv, "user_interface");
  ros::NodeHandle node_handle;
  ros::AsyncSpinner spinner(1);
  spinner.start();

  static const std::string PLANNING_GROUP = "roboter_arm";
  moveit::planning_interface::MoveGroupInterface move_group(PLANNING_GROUP);


  ROS_INFO_NAMED("user_interface", "Reference frame: %s", move_group.getPlanningFrame().
      c_str());
  ROS_INFO_NAMED("user_interface", "End effector link: %s", move_group.getEndEffectorLink().
      c_str());

  move_group.clearPoseTargets();
  move_group.setStartStateToCurrentState();
  move_group.setPoseReferenceFrame("base_link");

  move_group.setGoalTolerance(0.005);
  move_group.setPlanningTime(20);

  move_group.setPositionTarget(atof(argv[1]), atof(argv[2]), atof(argv[3]), move_group.
      getEndEffectorLink());

  move_group.move();

  ros::shutdown();
  return 0;
}
```

Listing A.4: user_interface.cpp

B Arbeitsaufwand

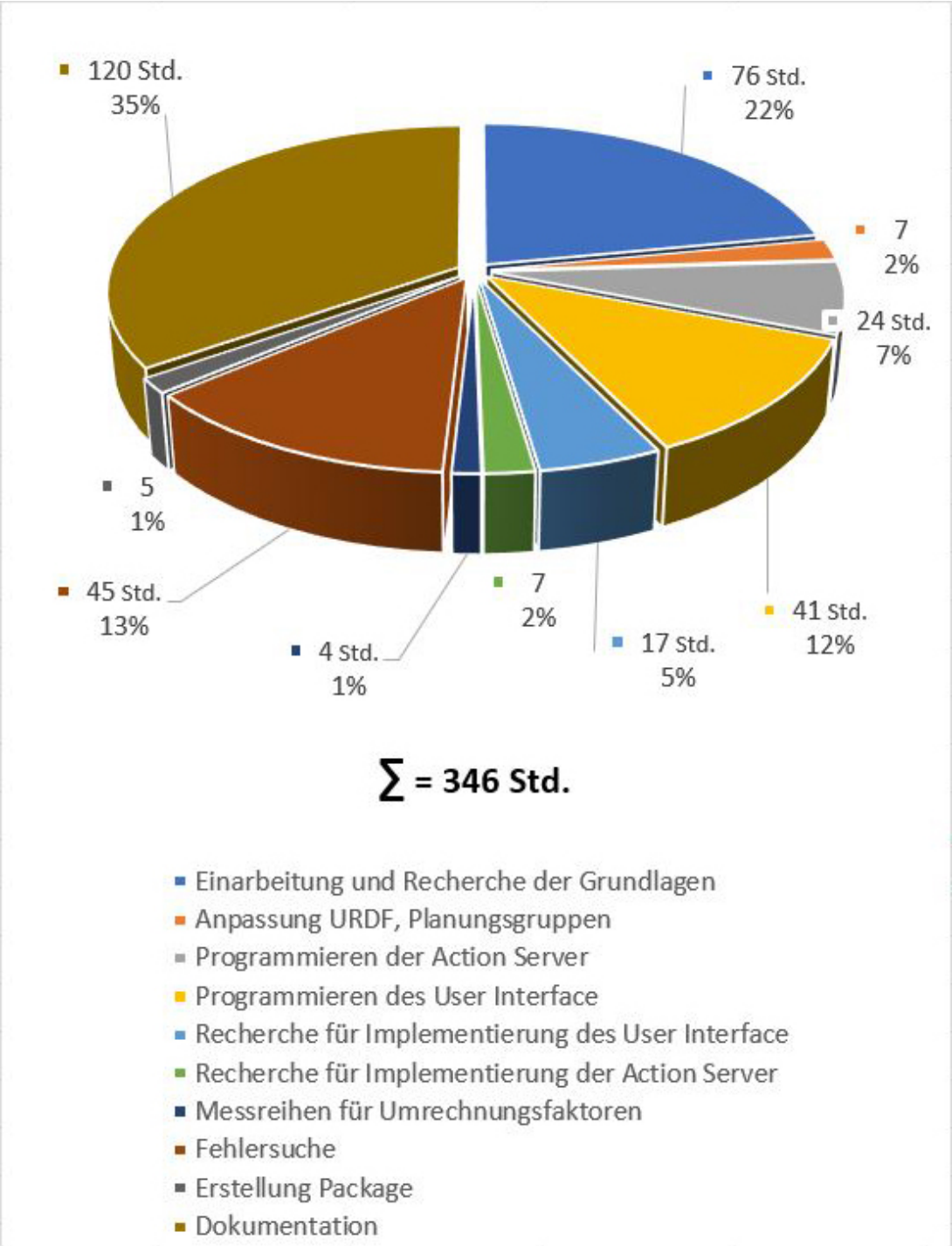


Abbildung B.1: Arbeitsaufwand

Index