Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

IMS Institut für
Mechatronische Systeme

VT2
FS17 MSE

Einsatz von ROS Industrial in einem Industrie 4.0 Demonstrator

Autor Luis Meier

Hauptbetreuung Dr. Marcel Honegger

Datum 31. Juli 2017

Abstract

Industrieroboter werden von diversen Herstellern angeboten, diese Roboter besitzen meist eine proprietäre Steuerung und Bahnplanung. Jeder Hersteller von Robotern und den entsprechenden Steuerungen bietet in der Regel ein sehr ähnliches Funktionspaket an. Der Aufruf dieser Funktionen und die dazu benötigte Programmiersprachen sind nicht bei allen Herstellern gleich. Bis anhin sind durch die Hersteller der Roboter keine Bestrebungen ersichtlich eine vereinheitlichte Programmiersprache und Ansteuerung der Roboter zu entwickeln. Mit dem Robot Operating System (ROS) und der dazugehörigen Erweiterung für Industrielle Roboter (ROS-Industrial) ist ein Framework vorhanden, welches es ermöglicht universell Software und Applikationen in der Robotik zu entwickeln.

Ziel dieser Arbeit ist es die Möglichkeiten und Einschränkungen von ROS-Industrial zu ermitteln, dazu soll eine Entwicklungsumgebung mit ROS-Industrial aufgesetzt werden und anschliessend eine Montageaufgabe mit ROS-Inudstrial realisiert werden. Als Montageaufgabe ist das Zusammenbauen eines Kugelschreibers, im Industie 4.0 Demonstrator vorgesehen. Der Industrie 4.0 Demonstrator ist ein Projekt der ZHAW und diverser Industriepartner. Es soll die vielen Aspekte von Industrie 4.0 in einem einzelnen Demonstrator veranschaulichen. Die Montageaufgabe soll auf den drei Robotern ABB IRB120, Stäubli TX2-60l und Universal UR3 realisiert werden.

Resultat dieser Arbeit ist eine fertig Aufgesetzte Arbeitsumgebung für ROS und ROS-Industrial, und eine Implementierung der Montage Aufgabe in den Industrie 4.0 Demonstrator. Es konnten jedoch nicht alle in der Aufgabenstellung geforderten Ziele erreicht werden. Hauptgründe dafür sind ein noch nicht angelieferter Roboter von Stäubli, Komplikationen bei der Ansteuerung der SMC Aktuatoren über EtherCAT und falsch bestellte Teile. Diese Komplikationen verhinderten, dass die implementierte Montageaufgabe vollständig geprüft und verbessert werden konnte.

Die Möglichkeiten von ROS-Industrial sind extrem Umfangreich, es werden durch die Open Source Community eine Vielzahl an Paketen mit Funktionen angeboten. Diese ermöglichen es in sehr kurzer Zeit eine sehr Komplexe Aufgabe zu lösen. Die Open Source Community ist zugleich aber auch ein Grund für diverse Einschränkungen von ROS und ROS-Industrial. Da viele Pakete nicht oder nur schlecht gewartet werden kommt es oft vor, dass nicht vollständig funktionsfähige Pakete implementiert werden müssen und diese allenfalls gepatcht werden müssen. Es kann zudem auch vorkommen, dass Nodes aufgrund von nicht nachvollziehbaren Gründen neu gestartet werden müssen.

Vorwort

Die Robotik faszinierte mich, seit ich das erste mal mit ihr in Kontakt kam. Ich konnte jedoch noch nie ein Projekt mit einem Industrieroboter miterleben. Das Thema ROS, ein so umfangreiches Softwareprojekt und C++ waren für mich absolutes Neuland und stellten mich oft vor grosse Herausforderungen. Einigie dieser Herausforderungen kosteten mich viele Nerven, mit ein paar wenigen bin ich bis zum Schluss nicht klar gekommen.

Erklärung betreffend das selbständige Verfassen einer Projektarbeit an der School of Engineering

Mit der Abgabe dieser Projektarbeit versichert der/die Studierende, dass er/sie die Arbeit selbständig und ohne fremde Hilfe verfasst hat. (Bei Gruppenarbeiten gelten die Leistungen der übrigen Gruppenmitglieder nicht als fremde Hilfe.)

Der/die unterzeichnende Studierende erklärt, dass alle zitierten Quellen (auch Internetseiten) im Text oder Anhang korrekt nachgewiesen sind, d.h. dass die Projektarbeit keine Plagiate enthält, also keine Teile, die teilweise oder vollständig aus einem fremden Text oder einer fremden Arbeit unter Vorgabe der eigenen Urheberschaft bzw. ohne Quellenangabe übernommen worden sind.

Bei Verfehlungen aller Art treten die Paragraphen 39 und 40 (Unredlichkeit und Verfahren bei Unredlichkeit) der ZHAW Prüfungsordnung sowie die Bestimmungen der Disziplinarmassnahmen der Hochschulordnung in Kraft.

Ort, Datum: Unterschriften:

…… ……

……..

……

Das Original dieses Formulars ist bei der ZHAW-Version aller abgegebenen Projektarbeiten zu Beginn der Dokumentation nach dem Abstract bzw. dem Management Summary mit Original-Unterschriften und -Datum (keine Kopie) einzufügen.

Inhaltsverzeichnis

Verzeichnisse

Abkürzungen

apt Advanced Packaging Tool

FCL Flexible Collision Library

GUI Graphical User Interface

HTTP Hypertext Transfer Protocol

IDE Integrated Development Environment

KDL Kinematics and Dynamics Library

OMPL The Open Motion Planning Library

SOEM Simple Open EtherCAT Master

TCP Tool Center Point

URDF Universal Robotic Description Format

VCS Version Control System

XML Extensible Markup Language

Glossar

Gimbal lock ist der Verlust eines Freiheitsgrades aufgrund von parallel liegenden Rotationsachsen.

Octomap ist eine Bibliothek, welche 3D-Rasterstrukturen implementiert um komplette Umgebungen zu modellieren.

XMLRPC ist eine Definition zum Methoden- und Funktionenabruf durch verteilte Systeme. Die Datenübertragung erfolgt über HTTP[1], die Darstellung der Daten per XML[2].

[1] Hypertext Transfer Protocol
[2] Extensible Markup Language

Abbildungsverzeichnis

Tabellenverzeichnis

Sourcecodeverzeichniss

1 Einleitung

Industrieroboter, wie sie von diversen Herstellern angeboten werden besitzen meistens eine proprietäre Steuerung. Die Hersteller dieser Steuerungen bieten meistens ein sehr ähnliches Funktionspaket an, jedoch wird über die Hersteller hinweg keine einheitliche Programmiersprache verwendet, es gibt zum Beispiel RAPID bei ABB, oder KRL bei KUKA. Bis anhin sind durch die Roboterhersteller keine Bestrebungen zu einer einheitlichen Programmiersprache in Sicht.
Mit dem Robot Operating System (ROS) ist ein Framework vorhanden, welches sich für die Entwicklung von universeller Software für Roboter anbietet. ROS wurde ursprünglich für den Teilbereich der Mobilrobotik entwickelt. Durch die Erweiterung ROS-Industrial steht jedoch ein weiteres Framework zur Verfügung, welches die Fähigkeiten von ROS auf die Anforderungen in der Industrie erweitert.

1.1 Ausgangslage

Im Rahmen zweier Bachelorarbeiten wurde im Frühlingssemester 2017 ein Industrie 4.0 Demonstrator konzeptioniert und entwickelt, welcher die diversen Aspekte von Industrie 4.0 aufzeigen können soll. Dazu können Kugelschreiber durch den Kunden - respektive den Messebesucher - in verschiedenen Zusammenstellungen konfiguriert werden, anschliessend werden diese Kugelschreiber durch den Demonstrator zusammengebaut.
Der Demonstrator wird in Zusammenarbeit mit mehreren Industriepartnern realisiert, die Industriepartner stellen diverse Komponenten für den Industrie Demonstrator zur Verfügung.

1.1.1 Projektorganisation

Das gesamte Projekt Industrie 4.0 Demonstrator wird durch mehrere Personen mit unterschiedlichen Zuständigkeiten realisiert. Die Planung des mechanischen Aufbaus der Anlage wird durch die beiden Bachelorstudenten Andrin Meister und Christian Hartmann durchgeführt. Claude Hasler entwickelt im Rahmen einer Verteifungsarbeit einen Minideltarobtoer, welcher für das Feeden der Kleinteile Zuständig ist.

1.2 Aufgabenstellung

Im Rahmen dieser VT2 soll eine konkrete Montageaufgabe für den Industrie 4.0 Demonstrator (SmartPro) mit einem Stäubli Industrieroboter realisiert werden, wobei als Programmiersprache `C++` und das ROS Industrial Framework eingesetzt werden. Nach Möglichkeit soll aber keine Stäubli spezifische Software entwickelt werden. Die Portierbarkeit dieser Anwendung soll schliesslich mit einem Wechsel des gewählten Roboters zu einem Universal Robots UR3 - respektive einem ABB IRB120 - gezeigt werden.
Dazu wurden folgende Arbeitspakete definiert:

- Aufsetzen einer Entwicklungsumgebung mit ROS Industrial.
- Simulation einer einfachen Montageaufgabe des Industrie 4.0 Demonstrators, wobei als Roboter ein Stäubli Industrieroboter, ein Universal Robots UR3 und ein ABB IRB120 verwendet werden sollen.
- Entwicklung und Umsetzung der Montageaufgabe mit dem Stäubli Industrieroboter.
- Portierung dieser Montageaufgabe auf einen Universal Robots UR3, inklusive Vergleichstest mit dem Stäubli Roboter.

- Evaluation von ROS Industrial, mit einer Beurteilung dessen Möglichkeiten und Einschränkungen für industrielle Anwendungen.

1.3 Verwendete Software

Für die vorliegende Arbeit wurden die unten aufgeführten Programme verwendets.

Arbeitsumgebung

- Ubuntu 16.04 LTS
- Microsoft Windows 10

IDE

- Visual Studio Code
- RoboWareStudio

CAD

- CATIA
- Meshlab

Dokumentation

- LaTeX mit TeXStudio 2.11.2
- yED Graph Editor

2 ROS Grundlagen

2.1 Überblick

Das Robot Operating System oder kurz ROS ist eine flexibles Framework zur Entwicklung von Software für Roboter. Es bietet eine Reihe von Funktionen, Werkzeugen und Bibliotheken. Mit diesen ist es möglich alle gängigen Operationen, welche man von einer Software von diesem Typ erwarten kann zu erledigen. Dazu gehören unter anderem Hardwareabstraktion, das Versenden von Nachrichten zwischen Prozessen und Paketmanagement. [1]
Der Kern von ROS ist unter der 3-Klausel-BSD-Lizenz lizenziert und somit komplett Open-Source. [2] ROS darf somit von jeder Person, oder Firma verwendet und auch weiterentwickelt werden, solange die drei Klauseln[1] der BSD-Lizenz eingehalten werden. ROS basiert genau aus diesem Grund auch aus diversen Teilprojekten, welche es seit der Entstehung von Anfang der 2000er Jahre vorangetrieben haben. [3]

2.1.1 Ziele von ROS

ROS setzt sich nicht zum Ziel ein Roboterframework zu sein, welches eine unzählige Vielfalt von Funktionen bereitstellt. Das Hauptziel von ROS ist es, dass Software welche für Roboter in der Forschung und der Entwicklung geschrieben wurde frei zugänglich ist und somit wiederverwendet werden kann. Aus diesem Grund ist ROS so aufgebaut, dass die einzelnen Komponenten respektive Funktionen lose aneinander gekoppelt werden können. Weitere Ziele von ROS sind es, dass die für ROS entwickelte Software nicht proprietär ist, sondern auch in anderen Roboterframeworks gebraucht werden kann. Gleichzeitig soll ROS unabhängig von der Programmiersprache sein, aus diesem Grund ist es möglich ROS in `C++, Python` und `Lisp` zu programmieren, zudem stehen experimentelle Bibliotheken zur Verfügung mit welchen `JAVA` und `Lua` eingebunden werden können. [3]

2.1.2 Betriebssysteme

Momentan ist ROS nur auf Unix basierten Betriebssystemen verfügbar. Die Software von ROS wird Hauptsächlich auf Ubuntu und Mac OS X getestet, es werden aber durch die Community Support für andere Linux Distributionen wie zum Beispiel Fedora, Gentoo und Arch Linux angeboten. Eine Portierung von ROS auf Microsoft Windows ist zwar möglich, wird aber im Moment noch nicht angeboten. [1]

2.1.3 Distributionen

Der Kern von ROS wird, ähnlich wie bei Linux Systemen, über Distributionen versioniert und verteilt. Diese Distributionen beinhalten alle für ROS gebrauchten Basispakete und Bibliotheken. Ziel dieser Distributionen ist es für Anwender eine stabile Basis zu schaffen, auf welcher aufgebaut werden kann. ROS versucht innerhalb einer Distribution keine grossen Änderungen zu vollziehen, damit plötzliche Inkompatibilitäten vermieden werden können.

[1] Siehe: `https://opensource.org/licenses/BSD-3-Clause`

Distro	Erscheinungsdatum	EOL Datum
ROS Lunar Loggerhead	23 May 2017	Mai 2019
ROS Kinetic Kame	23 May 2016	April 2021(Xenial EOL)
ROS Jade Turtle	23 May 2015	Mai 2017
ROS Indigo Igloo	22 Juli 2014	April 2019(Trusty EOL)

Tab. 2.1: Übersicht der noch gewarteten ROS Distributionen [4]

2.2 ROS Dateisystemebene

Die Dateisystemebene repräsentiert die Daten, welche auf der Festplatte abgespeichert sind. Die Dateisystemebene beinhaltet auch Konzepte, um den Aufbau von Packages einheitlich zu gestalten.

2.2.1 Packages

Packages sind der einfachste Organisationsbaustein in ROS, Ziel von Packages ist es für ROS-Anwender eine Struktur zu schaffen, welche einfach einsehbar und gut wiederverwendbar ist. Ein Package kann Nodes, unabhängige Bibliotheken, Datasets, Konfigurationsfiles, nicht ROS-Spezifischer Code und vieles anderes enthalten. In einem Package sind Typischerweise folgende Unterordner und Files zu finden.

include/package_name Headers von C++ Files

launch/ Ordner welcher .launch-Files enthält welche ein oder mehrere Nodes starten können

msg/ Ordner der .msg-Files enthält

srv/ Ordner der .srv-Files enthält

scripts/ Ordner der ausführbare Skripts enthält

CMakeLists.txt CMake Build File (siehe **??**)

Package.xml catkin Package File (siehe 2.6.3)

Changelog.rst Einige Pakete enthalten ein Änderungsprotokoll

2.2.2 Metapackages

Ein Metapackage ist eine Untergruppe eines Normalen Packages. Sie dienen nur dazu, eine Gruppe von verwandten Paketen darzustellen.

2.2.3 Repositories

Die meisten ROS-Packages werden durch ein VCS[2] verwaltet. Eine Gruppe von Packages, welche mit dem selben VCS verwaltet wird, wird in einem Repository zusammengefasst. Es ist auch möglich, dass ein Repository nur ein einzelnes Paket beinhaltet.

2.3 ROS Berechnungsebene

Die Berechnungsebene in ROS ist aufgebaut auf einem Peer-to-Peer Netzwerk aus einzelnen Prozessen, die Zusammenarbeit dieser einzelnen Prozessknoten ermöglicht es, dass komplexe Probleme gelöst werden können. Die Basis des auf dem TCP/IP-Protokoll aufgebauten Netzwerkes bilden Nodes, ROS-Master, Parameterserver, Messages, Topics, Services, Actions und Bags. All diese Komponenten tragen auf eine Art der Datenverarbeitung bei.

[2] Version Control System

2.3.1 Nodes

Ein ausführbarer Prozess wird in ROS als Node bezeichnet. Ein Node kann Tasks und Berechnungen ausführen, sowie über das ROS-Netzwerk kommunizieren. Jeder Node arbeitet individuell und ist mit einem TCP-IP-Socket verbunden. Dies erlaubt es, dass ein Absturz eines Nodes nicht einen Systemabsturz herbeiführt. Deshalb ist es auch zu empfehlen die Nodes so zu gestalten, dass jeder Node einen spezifischen Task erledigt und nicht ein Node mehrere Aufgaben hat, zudem wird durch die Aufteilung der Aufgaben die Komplexität des Codes tief gehalten. [5]

2.3.2 ROS-Master

Der ROS-Master stellt für alle Nodes im ROS-System Registrierungs- und Namensservices zur Verfügung. Die Aufgabe des ROS-Masters ist es dabei Publisher und Subscriber zu bestimmten Topics, sowie Services zu registrieren. Dies ermöglicht es anschliessend, dass sich die einzelnen Nodes finden können, um anschliessend Peer-to-Peer kommunizieren können. Zudem stellt der ROS-Master den Parameterserver zur Verfügung. Der ROS-Master muss für das Ausführen eines oder mehrerer Nodes bereits gestartet worden sein, dies erfolgt über den Bashbefehl `$ roscore`. Alternativ kann der ROS-Master auch zusammen mit dem Ausführen eines .launch-Files geschehen, dies geschieht automatisch beim ausführen des Befehls: `$ roslaunch 'package_name' 'file_name.launch'`. [6]

2.3.3 Parameterserver

Der Parameterserver ist ein Teil vom ROS-Master und kann durch die Nodes über eine in ROS integrierte Netzwerk-API beschrieben, respektive ausgelesen werden. Der Parameterserver ist nicht auf einen hohen Datendurchsatz ausgelegt, aus diesem Grund eignet er sich am besten, für die Speicherung von statischen Daten. Die gespeicherten Parameter sind dabei grundsätzlich Global sichtbar, dies ermöglicht es, dass die Konfigurationen einfach einsehbar sind und bei bedarf schnell geändert werden können. Auf dem Parameterserver werden zum Beispiel die Geometriedaten des Roboters abgelegt, welche so für alle Nodes sichtbar sind. Der Parameterserver ist mithilfe von XMLRPC implementiert. Es werden folgende Datentypen unterstützt: [7]

- 32-bit int
- boolean
- string
- float
- iso8601 Daten
- Listen
- base64 verschlüsselte Binärdaten

2.3.4 Messages

Datenpakete welche über das ROS-System gesendet werden, werden als Messages definiert. Ein solches Datenpaket kann ein oder mehrere Daten von den folgenden Standardtypen besitzen:

- boolean
- un-/signed int
- float
- string
- time (ROS speziefisch)
- durration (ROS speziefisch)

Zudem ist es möglich structures aus den oben genannten Datentypen zu senden.

2.3.5 Topics

Für die asynchrone Kommunikation werden in ROS Topics gebraucht, diese Funktionieren über Publisher und Subscriber. Ein Node kann sich als Publisher oder als Subscriber in einem Topic über den ROS-Master registrieren. Die Anzahl der Publisher respektive Subscriber in einem Topic ist nicht begrenzt. Ein Publishernode hat keine Information darüber, ob oder wie viele Subscriber in einem Topic registriert sind. Subscriber sehen jede von jedem Publisher auf dem Topic publizierte Nachricht. Die von den Publishern versendeten Daten müssen sich dabei an eine vordefinierte Vorlage der Message halten, somit ist es nicht möglich zusätzliche Daten über ein Topic zu senden, welche nicht in der Message definiert sind.

2.3.6 Services

Services dienen der synchronen Kommunikation in ROS, dazu wird ein Request/Response-Modell verwendet. Ein Node kann einen Service beim ROS-Master registrieren, damit dieser von anderen Nodes gefunden werden kann. Ein Node welcher auf einem bestimmten Service registriert ist erhält nur Daten als Antwort auf eine vorher getätigte Anfrage an den Node, welcher den Service anbietet. Die Antwort auf eine Anfrage ist dabei meist spezifisch auf die getätigte Anfrage. Auch hier ist es nötig vorgängig eine Message zu definieren welche Request und Response Dateien enthält. Ein solches .msg-File würde für einen Service, welcher den Satz des Pythagoras berechnet wie folgt aussehen:

```
# Define request
float32 a
float32 b
---
# Define response
float32 c
```

Codesnippet 2.1: Beispiel .srv-File

2.3.7 Actions

Actions ähneln sehr stark den Services der Unterschied dabei ist, das bei Actions während des Ausführens nicht blockieren. Somit ist es möglich während der Request vom Client Feedback über momentane Prozessdaten zu erhalten, der Client kann auch während der Ausführung einen Abbruch des Prozesses vom Server verlangen, gleichzeitig ist es möglich das Ziel aufgrund des erhaltenen Feedbacks anzupassen. Wie auch bei den Topics und Services muss auch bei den Actions vorgängig ein Messagefile erstellt werden, welches das Ziel, Resultat und das Feedback definiert. Actions kommunizieren über das ROS eigene "ROS Action Protocol", welches auf den ROS-Messages aufbaut. Die Clients und Server einer Action stellen für User eine API zur Verfügung

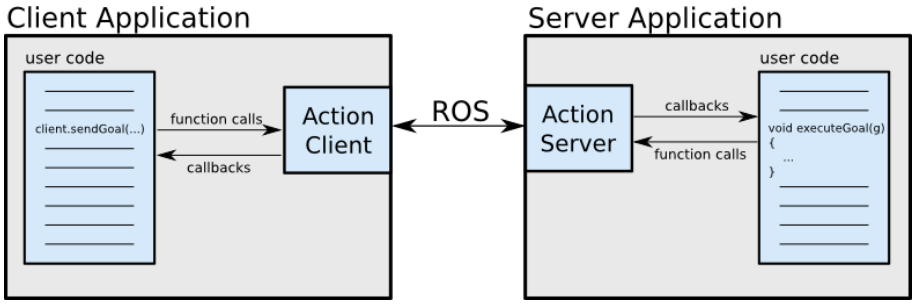

Abb. 2.1: Interaktion Client und Server
Interaktion Client und Server [8]

2.3.8 Bags

Bags werden in ROS gebraucht um die Daten, welche über eine Message verschickt werden zu speichern. Bags werden nicht aktiv für den Ablauf von Programmen benötigt, sind aber sehr hilfreich um Sensor- und Prozessdaten zu speichern und anschliessend auszuwerten.

2.4 ROS Communityebene

Für ROS ist, aufgrund des Open Source Aspekts, die Community ein sehr wichtiger Bestandteil des ganzen Projektes. Die ROS Community enthält ein extrem grosses Wissens- und Erfahrungspotential, dieses Wissen und die Erfahrungen wird von den einzelnen Gruppen der Community online zur Verfügung gestellt. Dazu stehen mehrere Möglichkeiten zur Verfügung:

- Repositories
- ROS-Wiki unter `http://wiki.ros.org/`
- Bug Ticket System
- Mailverteiler
- Die 'ROS Answers'-Website

Das ROS-Wiki ist eine vom ROS-Konsortium erstellte Website, welche nahezu für alles ein vollständige Dokumentationen und Tutorials, mindestens aber Kurzbeschriebe, bereithält. Die ROS Answers Website bietet allen Anwendern von ROS eine Plattform in Form eines Forums um Fragen zu stellen und diese zu beantworten. Mithilfe dieser beiden Webseiten lassen sich fast alle Probleme und Unklarheiten, welche mit ROS in Verbindung stehen meistens klären.

2.5 Standard Masseinheiten

In ROS muss immer wieder mit Längen und Winkeln gearbeitet werden. Aus diesem Grund ist in ROS ein Standardisiertes Einheitensystem vorhanden.

Grösse	**Einheit**	**Grösse**	**Einheit**
Länge	Meter	Leistung	Watt
Masse	Kilogramm	Spannung	Volt
Zeit	Sekunden	Temperatur	Celsius
Strom	Ampere	Magnetismus	Tesla
Winkel	Radiant	Frequenz	Herz
Kraft	Newton		

Tab. 2.2: Standard Masseinheiten in ROS
Standard Masseinheiten in ROS [9]

2.5.1 Quaternionen

In ROS werden Rotationen nicht mit Eulerwinkeln gerechnet sondern in Quanternionen. Der grosse Vorteil von Quanternionen gegenüber der Verwendung von Eulerwinkeln ist, das bei den Quanternionen kein Gimbal lock entstehen kann. Quanternionen sind wie folgt definiert:

$$\begin{aligned}[w, x, y, z] &= [cos(\Theta/2), sin(\Theta/2) \cdot nx, sin(\Theta/2) \cdot ny, sin(\Theta/2) \cdot nz] \\ \Theta &= \text{Drehwinkel} \\ [nx, ny, nz] &= \text{Achse der Rotation}\end{aligned}$$

2.6 Catkin

Catkin ist das Standard Build-Tool von ROS. Catkin kombiniert CMakemakros und Pythonskripts um dem normalen CMake einige neue Funktionalitäten hinzuzufügen. Catkin wurde erstellt um das vorherige Build-Tool rosbuild abzulösen, catkin ermöglicht es Pakete mithilfe von 'find package' zu finden und mehrere voneinander abhängige Projekte gleichzeitig zu compilieren.
ROS nutzt ein eigenes Build-Tool um es zu ermöglichen, dass alle Pakete in ROS, unabhängig von ihrer Programmiersprache, ohne Probleme compiliert werden können. Es wird mit catkin ermöglicht voneinander abhängige Pakete zu compilieren, welche mit unterschiedlichen Programmiersprachen geschrieben wurden, andere Tools benötigen und/oder andern Organisationsstrukturen unterliegen. [10]

2.6.1 Workspace

Catkin benötigt einen eigenen Workspace, welcher wie folgt initialisiert werden kann:
(Hier als Beispiel im Homedirectory im Ordner catkin_ws)

```
$ source /opt/ros/kinetic/setup.bash
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/
$ catkin_make
```

Codesnippet 2.2: Setup eines catkin Workspace

Anschliessend ist es zu empfehlen, dass die Variable 'ROS_PACKAGE_PATH' so gesetzt wird, dass der Workspace erkannt wird:

```
$ echo $ROS_PACKAGE_PATH
/home/youruser/catkin_ws/src:/opt/ros/kinetic/share
```

Codesnippet 2.3: Setzen der Umgebungsvariable

2.6.2 CMakeLists

Jedes Package in ROS benötigt zum compilieren durch catkin eine CMakeLists.txt Datei in ihrem Ordner. In dieser Datei werden diverse zum compilieren benötigte Einstellungen vorgenommen, unter anderem wird angegeben wo die Standard ROS-Packages zu finden sind, welche Packages zum compilieren Notwendig sind, welche der Files einen Node bilden und auch Abhängigkeiten können definiert werden. Zudem wird dem Compiler hier mitgeteilt, aus welchen Files Messages oder Services generiert werden sollen. Es ist zudem möglich und sehr zu empfehlen Flags zu setzen, welche erweiterte Fehlermeldungen bei Kompilierfehlern ausgeben. Dies vereinfacht ein Debuggen Nachfolgend sind einige der wichtigsten Zeilen aus einem CMakeFile.txt aufgeführt. Ein vollständiges CMakeFile kann im Anhang unter 2.1 gefunden werden. [10]

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  geometry_msgs
  moveit_core
  moveit_ros_planning
  moveit_ros_planning_interface
  pluginlib
  cmake_modules
  message_generation
  genmsg
)
```

Codesnippet 2.4: Für das compilieren benötigte ROS-Packages

```
add_executable(NodeName
  src/myROSNode.cpp
)
```

Codesnippet 2.5: Definieren eines Nodes aus einem .cpp-File

```
add_library(LibraryName
  src/myLibrary.cpp
)
target_link_libraries(targetName
  ${catkin_LIBRARIES}
  LibraryName
)
```

Codesnippet 2.6: Hinzufügen und verlinken von Bibliotheken

2.6.3 Package File

Das Package File package.xml wird von catkin gebraucht um Metadaten zu definieren. Diese beinhalten Versionsnummer, Paketname, Ersteller/Unterhalter und Lizenzen. Zudem muss angegeben werden welche anderen Packages vor dem compilieren bereits compiliert sein müssen (build_depend), welche Packages zum compilieren benötigt werden (buildtool_depend) und welche Packages während der Laufzeit (run_depend) des Packages benötigt werden. Folgend ein Minimalbeispiel einer package.xml Datei. [10]

```
<?xml version="1.0"?><package>
  <name>myPackage</name>
  <version>0.0.1</version>
  <description>My personal ROS-Package. Which does...</description>

  <maintainer email="Hans.Muster@example.com">Hans Muster</maintainer>

  <license>BSD</license>

  <author email="Hans.Muster@example.com">Jane Doe</author>

  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <run_depend>roscpp</run_depend>
  <build_depend>moveit_core</build_depend>
  <run_depend>moveit_core</run_depend>
  <build_depend>moveit_ros_planning_interface</build_depend>
  <run_depend>moveit_ros_planning_interface</run_depend>

  <!-- The export tag contains other, unspecified, tags -->
  <export>
  <!-- Other tools can request additional information be placed here -->
  </export>
</package>
```

Codesnippet 2.7: Minimalbeispiel einer package.xml Datei

2.7 URDF

Das URDF[3] wird gebraucht um die physikalischen Eigenschaften eines Roboters zu beschreiben. Ein URDF besteht aus einer Zusammenstellung von Links (Gliedern) und Joints (Gelenken) um eine Kinematische Kette zu bilden. (siehe Abb. 2.2)

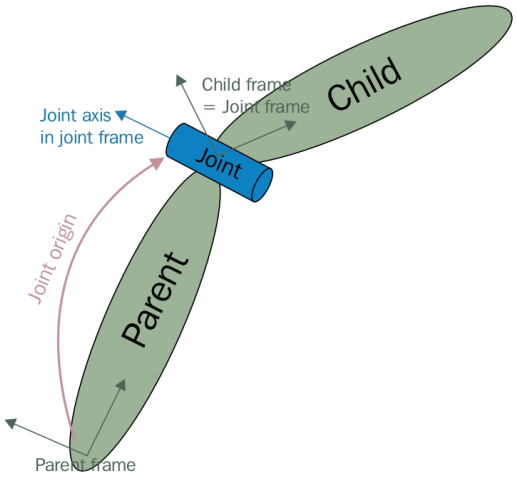

Abb. 2.2: Symbolische Darstellung einer Kinematischen Kette[3]

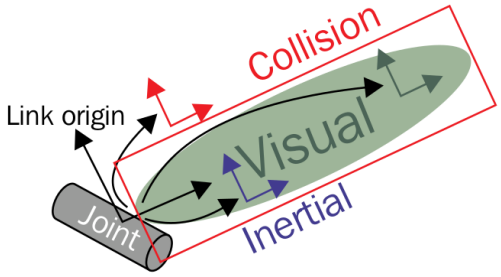

Abb. 2.3: Symbolische Darstellung eines Links[3]

2.7.1 Links

Für jeden Link muss dabei ein Koordinatensystem und ein Geometrisches Modell definiert sein. Optional ist es möglich Massenträgheitsmomente, Farben und eine simplere Geometrie für die Kollisionsüberwachung zu definieren. Für die Geometrien ist es zudem möglich, die 3D-Objekte über importierte .stl-Files zu generieren.

```
<link name="base_link">
  <inertial>
    <mass value="6.215"/>
    <origin rpy="0 0 0" xyz="-0.04204 8.01E-05 0.07964"/>
    <inertia ixx="0.0247272" ixy="-8.0784E-05" ixz="0.00130902" iyy="0.0491285" iyz="-8.0419E-06"
    ↪  izz="0.0472376"/>
  </inertial>
  <visual>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <geometry>
      <mesh filename="package://abb_irb120_support/meshes/irb120_3_58/visual/base_link.stl"/>
    </geometry>
    <material name="">
      <color rgba="0.7372549 0.3490196 0.1607843 1"/>
    </material>
  </visual>
  <collision>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <geometry>
      <mesh filename="package://abb_irb120_support/meshes/irb120_3_58/collision/base_link.stl"/>
    </geometry>
    <material name="">
      <color rgba="1 1 0 1"/>
    </material>
```

[3]Universal Robotic Description Format

```
  </collision>
</link>
```

Codesnippet 2.8: Beispiel eines voll definierten Links anhand der Basis des ABB IRB120

2.7.2 Joints

Für einen Gelenk (Joint) muss definiert werden um welche Achsen sich das Gelenk drehen darf und welche beiden Links das Gelenk verbindet. Optional können Endanschläge, maximale Geschwindigkeiten, Reibungs- und Dämpfungskonstanten definiert werden. Mögliche Definitionen von Drehachsen sind revolute, continuous, prismatic, fixed, floating und planar. Nachfolgend ist ein Codeausschnitt aufgeführt, welcher ein vollständig definiertes Gelenk beschreibt.

```
<joint name="joint_1" type="revolute">
  <origin rpy="0 0 0" xyz="0 0 0"/>
  <parent link="base_link"/>
  <child link="link_1"/>
  <limit effort="0" lower="-2.87979" upper="2.87979" velocity="4.36332"/>
  <axis xyz="0 0 1"/>
  <dynamics damping="0.0" friction="0.0"/>
</joint>
```

Codesnippet 2.9: Beispiel eines Rotationgelenkes

2.8 MoveIt!

MoveIt!ist eine Software zur Bewegungsplanung, Manipulation, 3D Wahrnehmung, Kinematik, Steuerung und Navigation. Es basiert auf einer Pluginstruktur, welche von ROS übernommen wurde. Dies ermöglicht es, dass MoveIt! sehr Ressourcenschonend ist und jeweils nur die Teile aktiv sind, welche auch für die jeweilige Anwendung nötig sind. Zudem ist es möglich die move_group einfach über neue Plugins zu erweitern. Der zentrale Node in MoveIt! ist der move_group Node, auf die Services und Actions dieses Nodes können über drei mögliche Arten zugegriffen werden: [11]

move_group_interface Das move_group_interface bietet eine API für `C++`, diese eignet sich vor allem für die Erstellung komplexer Anwendungen.

moveit_commander Das moveit_commander Package bietet eine API für `Python`, welche sich vor allem für das erstellen von einfachen Skripts eignet.

RVIZ Mithilfe eines Plugins für RVIZ, dem ROS eigenen Visualisierungspackage, kann die move_group direkt über ein GUI[4] gesteuert werden. Dies eignet sich vor allem für schnelles Visualisieren oder das Aufsetzen eines Roboters.

2.8.1 Konfiguration

Beim Start einer neuen move_group bezieht diese direkt beim Parameterserver von ROS benötigte Dateien. Diese beinhalten eine URDF Datei(siehe Abschnitt 2.7, welche die Physikalische Beschreibung des zu steuernden Roboters enthält. Ergänzend zum URDF wird ein SRDF benötigt, welches zusätzliche Informationen über den Roboter enthält, welche beim erstellen eines MoveIt!-Packages (siehe Abschnitt 5) definiert werden. Die beiden Files (URDF und SRDF) müssen zwingend auf dem Parameterserver von ROS abgelegt sein. Zusätzlich sucht die move_group auf dem Parameterserver noch nach weiteren, nicht zwangsweise vorhandenen, Konfigurationen. Diese können unter anderem maximale Gelenkgeschwindigkeiten, -beschleunigungen und -winkel enthalten sowie auch Daten über die Umgebung. [12]

[4] Graphical User Interface

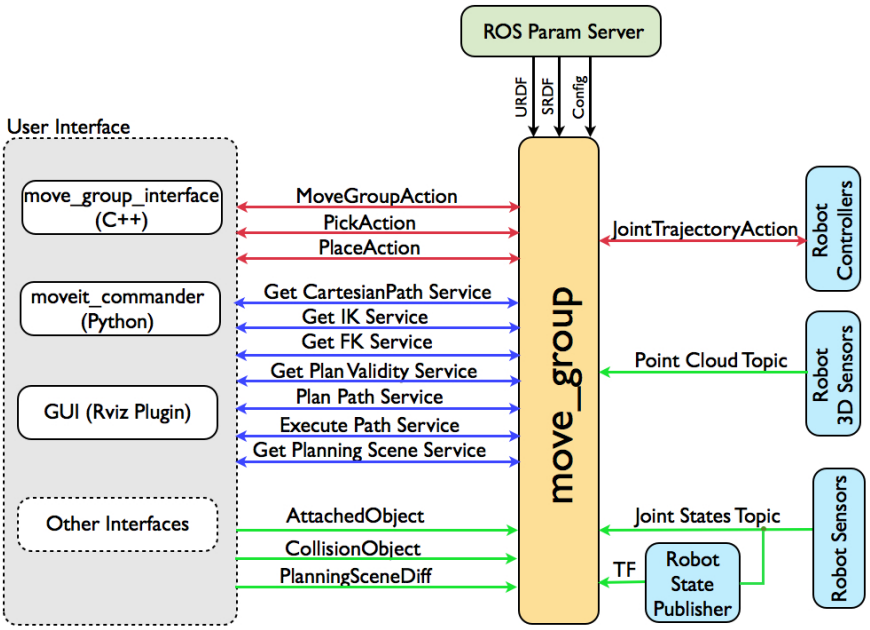

Abb. 2.4: move_group Architektur [12]

2.8.2 Roboterinterface

Über das Roboterinterface kommuniziert die move_group mithilfe von ROS Topics und Actions mit dem ausgewählten Roboter.

Joint State und Transformations Informationen

Zur Bestimmung der momentanen Position des Roboters hört die move_group auf dem Topic /joint_states, nach publizierten Gelenkpositionen. Diese Information geht auch über den Node Robot State Publisher welcher mithilfe eines Kinematischen Models die einzelnen Gelenkwinkel in eine genaue Position der jeweiligen Koordinatensysteme des Roboters umrechnet. Somit stehen der move_group jederzeit die einzelnen Gelenkwinkel sowie auch die Position jeder Achse des Roboters zur Verfügung. [3]

Controller Interface

Die move_group sendet Bewegungsbefehle über das FollowJointTrajectoryAction Interface, welches von ROS zur Verfügung gestellt wird, auf den entsprechenden Roboter Controller. Dabei ist zu beachten, dass auf dem Controller vorgängig ein entsprechender Actionserver implementiert werden muss, welcher die Anfragen in Steuerungsbefehle umwandelt. [3]

2.8.3 MoveIt! Setup Assistant

Der MoveIt! Setup Assistant ist ein sehr hilfreiches Tool welches ermöglicht jegliche Art von Roboter zu Konfigurieren so, dass dieser anschliessend mit MoveIt! respektive der move_group gebraucht werden kann. Dazu benötigt der Setup Assistant ein gültiges URDF-File, aus welchem ein SRDF-File und andere für MoveIt! und die Bewegungsplanung benötigten Files generiert werden. Dank dem GUI des Setup Assistants wird einem das erstellen des configuration packages sehr erleichtert. Dazu müssen folgende Schritte abgearbeitet werden: [13]

1. Starten und URDF laden
2. Generieren der Selbstkollisionsmatrix

3. Virtuelle Gelenke hinzufügen
4. Planungsgruppen erstellen
5. Roboterposen hinzufügen
6. Endeffektoren hinzufügen
7. Passive Gelenke hinzufügen
8. Konfigurationsfiles generieren

Der Setup Assistant kann über ein Linux Terminal mit folgendem Befehl gestartet werden:
`$ roslaunch moveit_setup_assistant setup_assistant.launch`
Eine ausführlichere Beschreibung anhand eines Beispiels ist im Abschnitt 5.3 aufgeführt.

2.8.4 Planningscene

Die Planningscene beinhaltet eine Repräsentation der echten Welt, in ihr ist der Roboter in seinem momentanen Zustand abgebildet, sowie auch alle Objekte welche sich momentan im Arbeitsraum des Roboters befinden. Die Planningscene wird für die Bestimmung von Kollisionsberechnungen gebraucht. [14]

2.8.5 Kollisionsüberwachung

Auch für die Kollisionsüberwachung wird eine Bibliothek über ein Plugin implementiert, standardmässig ist dies die FCL[5]. Dabei werden die folgenden Geometrischen Formate unterstützt: [14]

- Meshes: .stl oder .dae
- Primitive Shapes: Boxen, Zylinder, Kugeln, etc.
- Octomap: kann direkt für Kollisionsüberwachung gebraucht werden

Beim erstellen eines neuen MoveIt! Packages kann mithilfe des MoveIt! Setup Assistants eine Kollisionsmatrix erstellt werden, welche definiert welche Körper für die Berechnung der Kollision vernachlässigt werden können weil sie zum Beispiel gar nicht kollidieren können. Dies erspart einen grossen Teil an Rechenleistung und somit können Bahnplanungen schneller durchgeführt werden. [11]

2.8.6 Kinematischer Solver

Zur Lösung von Kinematikberechnungen sind in MoveIt! standardmässig zwei numerische kinematische Solver über ein Plugin implementiert. [11] Bei diesen Solvern handelt es sich um KDL und LMA beide basieren auf der KDL[6] von Orocos. [15] Zusätzlich wurde der Planer Trac-IK von Traclabs installiert, welcher laut den Angaben von Traclabs massiv schneller und zuverlässiger Lösungen finden soll, als die beiden Standardsolver. [16] Es muss je nach Roboter ausgetestet werden, welcher Solver bessere Ergebnisse liefert.

2.8.7 Bewegungsplanung

MoveIt! implementiert Bewegungsplaner über ein Plugin, dies ermöglicht es einfach zwischen mehreren Bewegungsplanungsbibliotheken zu wechseln, gleichzeitig ist es möglich einen eigenen Anwendungsspezifischen Bewegungsplaner in MoveIt! zu implementieren. Die Standard mässig implementierte Planungsbibliothek ist OMPL[7]. Mithilfe des MoveIt! Setup Assistant können die einzelnen Planer Konfiguriert werden. [11]

Um ein Grundverständnis für die Planer zu entwickeln wird in diesem Abschnitt kurz auf die in OMPL verfügbaren Planer eingegangen, aufgrund der Komplexität der Planer wird auf eine

[5] Flexible Collision Library
[6] Kinematics and Dynamics Library
[7] The Open Motion Planning Library

ausführlichere Beschreibung verzichtet. Grundsätzlich können die Planer welche in OMPL vorhanden sind in zwei Kategorien unterteilt werden. Es gibt geometrische Planer und Kontrollbasierte Planer. [17]

Geometrische Planer

Geometrische Planer berücksichtigen bei der Bahnplanung nur geometrische und kinematische Beschränkungen. Dabei trifft der Planer die Annahme, dass jeder vom Bahnplaner gefundene Weg in eine Bahn umgewandelt werden kann welche Dynamisch auch fahrbar ist. Dabei können diese Planer wieder in folgende Unterkategorien unterteilt werden: [17]

Single-query Planer Diese Planer bauen eine Baumstruktur auf, welche aus gültigen Bewegungen besteht und sich vom Start in Richtung Ziel bewegen. Einige Planer bauen die Baumstruktur gleichzeitig von Ziel und Start auf und treffen sich in der Mitte. Die einzelnen Planer dieses Typs unterscheiden sich hauptsächlich darin, wann und wo die Baumstruktur erweitert wird. [17]

Multi-query Planer Diese Art von Planern erstellt vorgängig eine Karte der ganzen Umgebung, welche anschliessend genutzt wird um zu bestimmen ob sich die berechneten Bewegungen innerhalb dieser Karte befinden und somit gültig sind. [17]

Optimierende Planer Optimierende Planer wählen gefundene Wege aufgrund von Optimierungskriterien, normalerweise ist dies die Distanz welche kurz gehalten werden soll. Es ist jedoch auch möglich, zum Beispiel die mechanische Arbeit zu minimieren. Obwohl das finden eines optimalen Weges von Vorteil ist, ist zu beachten, dass optimierende Planer die ihnen zur Verfügung gestellte Rechenzeit in der Regel komplett ausnutzen. Somit sind die gefahrenen Wege meistens kürzer als bei den Queryplanern es ist aber möglich, dass die gesparte Fahrzeit in längerer Rechenzeit verloren geht. [18]

Kontrollbasierte Planer

Kontrollbasierte Planer setzen nicht nur auf geometrische und kinematische Beschränkungen. Es werden Zustandsdarstellungen benutzt, um einzelne Positionen auf dem Weg zu definieren. Über Algorithmen wird bewertet wie hoch der Steuerungsaufwand ist um von Zustand zu Zustand zu gelangen, dieser soll möglichst klein gehalten werden. [17]

2.8.8 Trajektorienberechnung

Die aus der Bewegungsplanung berechneten Bahnen beinhalten noch keine Zeitinformationen, aus diesem Grund ist in MoveIt! ein Plugin zur Trajektorienberechnung implementiert. Dieses Plugin berechnet aufgrund der in den Konfigurationsdateien abgespeicherten erlaubten Gelenkgeschwindigkeiten aus der berechneten Bahn eine Trajektorie. Auch hier ist es wieder möglich, über ein Plugin eine eigene Version der Trajektorienberechnung zu implementieren. [11]

2.9 ROS Industrial

ROS-Industrial erweitert das von ROS zur Verfügung gestellte Framework rund um Industrierobotik und -automation. Die Erweiterung durch ROS-Industrial umfasst eine Vielzahl von Interfaces für diverse in der Industrie gebräuchliche Hardware, wie zum Beispiel Greifer oder Sensoren. Zudem sind in dem ROS-Industrial Repository[8] diverse Bibliotheken verfügbar, welche die spezifisch auf die Anwendung im Rahmen der Industrie zugeschnitten sind. ROS-I wird durch ein Konsortium [19] an Forschergruppen sowie auch von Firmen aus der Industrie unterstützt. Folgend sind einige Vorteile und Eigenschaften, der ROS-I Beschreibung aufgeführt: [20]

- Implementiert die bereits mächtigen Funktionen aus ROS

[8] https://github.com/ros-industrial/

- Ermöglicht neue Applikationen
- Vereinfacht die Roboterprogrammierung bis auf die Taskebene
- Reduziert Kosten
- Open Source

ROS-Industrial kann grundsätzlich in zwei Pakettypen unterteilt werden. Zum einen gibt es die ROS-I spezifischen Pakete, diese beinhalten alle generell in ROS-I brauchbaren Funktionen. Zum anderen gibt es die herstellerspezifischen Pakete, diese beinhalten die benötigten Beschreibungs- sowie Setupfiles, um einen Roboter eines bestimmten Herstellers in Betrieb zu nehmen.

2.9.1 Unterstützte Hersteller

ROS-I arbeitet eng mit diversen Forschungseinrichtungen sowie auch Herstellern zusammen und unterstützt bereits von Haus aus diverse Hardware von einigen Herstellern. Gleichzeitig gibt es eine sehr grosse Anzahl an Paketen und Bibliotheken, welche durch die Open Source Community erstellt wurden aber nicht offiziell - oder zumindest noch nicht - von ROS-I unterstützt werden.

Hersteller	**Controller(s)**	**Manipulator**	**MoveItPkg**
ABB	IRC5	IRB-2400	YES
		IRB-5400	NO
Adept	CX, CS	Viper 650	NO
Fanuc	R-30iA / R-30iB	LR Mate 200iC (all)	YES
		LR Mate 200iD	YES
		M-10iA	YES
		M-16iB/20	YES
		M-20iA(/10L)	YES
		M-430iA/(2F, 2P)	YES
		M-900iA/260L 5	NO
Motoman	DX100	SIA10D/F	NO
		FS100	NO
		DX200	NO
		YRC1000	NO
Universal Robot	CB2/CB3 10	UR 5	YES
		UR 10	YES

Tab. 2.3: Offiziell unterstützte Hardware [21]

2.9.2 Robot Support Package

Robotsupportpackages werden von ROS-Industrial verwendet um alle Dateien an einem Ort zu sammeln, welche nötig sind um einen Industrieroboter zu betreiben, respektive eine MoveIt! Konfiguration dafür zu erstellen. Die Strukturen, Namensgebungen sowie auch Funktionen in einem ROS-I Package unterliegen in diverser Konvention.

Ordnerstruktur

Übersichtshalber ist der Inhalt solcher Packages definiert, es wird auch empfohlen beim erstellen solcher Packages sich an die Ordnerstruktur sowie an die Namenskonventionen zu halten. Die Ordnerstruktur sieht in folgendermassen aus:

```
RoboterName_support
├── config
├── launch
├── meshes
├── test
└── urdf
```

Launchfiles

Durch ROS-Industrial werden einige .launch-Files vorgegeben, welche in jedem Support Package vorhanden sein müssen. Diese werden teils automatisch generiert, andere müssen selber geschrieben werden. Nachfolgend eine Auflistung der .launch-Files und eine kurze Beschreibung ihrer Funktion.

load_roboterName.launch Lädt alle roboterspezifischen Konfigurationsdateien, welche von MoveIt! benötigt werden auf den ROS Parameterserver.

test_roboterName.launch Startet RVIZ und ermöglicht es die Konfiguration der Gelenke zu überprüfen. Dafür werden die beiden Nodes joint_state_publisher und robot_state_publisher gestartet.

robot_state_visualize_roboterName.launch Visualisiert den Zustand eines echten oder simulierten Roboters in RVIZ. Es können dem Robotercontroller dabei keine Befehle erteilt werden.

Das letzte .launch-File ist das robot_interface welches eine bidirektionale Kommunikation mit dem Robotercontroller startet. Dieses .launch-File gibt es in zwei unterschiedliche Versionen, abhängig vom Controller. Diese unterscheiden sich darin, wie ROS-I mit dem Robotercontroller kommuniziert.

robot_interface_download_roboterName.launch Das Download Interface sendet die gesamte berechnete Trajektorie in einem Paket an den Controller, dieser führt diese Anschliessend aus. Beim Erhalt eines neuen Trajektorienpakets wird der Roboter gestoppt und die alte Trajektorie verworfen.

robot_interface_streaming_roboterName.launch Beim Streaming Interface werden die generierten Trajektorienpunkte direkt an den Controller gesendet, welcher leicht Zeitverzögert die ankommenden Trajektorienpunkte ausführt.

2.9.3 Standardisierte Links

Gemäss ROS-I sind drei Links definiert, welche in jedem Package gleich benennt sein müssen. Einerseits der base_link welcher die Basis des Roboters bildet, an welcher dieser befestigt wird. Der tool0 Link ist der letzte Link eines Roboters, er befindet sich somit im Zentrum des TCP[9], seine Orientierung entspricht der Orientierung des Standard Toolframes des Roboters. Der dritte definierte Link ist der flange_frame, gemäss der Definition von ROS-I entspricht seine Position dem Befestigungspunkt des jeweiligen Endeffektors. Die Orientierung ist genormt mit der x-Achse in Vorwärtsrichtung, y nach links und z nach oben.

2.9.4 URDF's und xacro

Die physikalischen Beschreibungen eines industriellen Roboters werden nicht in einem URDF verfasst, sondern in einem .xacro-File. Diese sind vom Syntax her identisch, jedoch erlauben es .xacro-Files Macros zu verwenden. Mithilfe von Macros können Tags generiert werden, welche es ermöglichen andere .xacro-Files zu suchen und Importieren. Zudem ist es möglich jedem Gelenk und Glied eines Roboters ein Präfix zu geben. Dies verhindert bei mehrfacher Verwendung eines .xacro-Files Namenskonflikte, falls ein xacro mehrfach verwendet werden will. Aus einem .xacro-File muss kann anschliessend wieder ein URDF generiert werden.

[9] Tool Center Point

3 ROS: Setup und Tools

Damit mit ROS und ROS-I gearbeitet werden kann müssen zuerst einige Installationen und Setups vorgenommen werden. Diese werden in diesem Abschnitt erläutert. Gleichzeitig werden einige ROS spezifische Tools erläutert, die das Arbeiten und auch Debuggen erleichtern können.

3.1 Auswahl Distribution

Für die Implementierung des Basis ROS-Systems, sowie für ROS-Industrial musste eine geeignete ROS-Distribution (siehe Abschnitt 2.1.3) gewählt werden. Die Wahl viel Dabei auf die Distribution ROS Kinetic Kame. Diese Distribution wurde gewählt, da sie im Vergleich zu ihrer Vorgängerversion (ROS Jade Turtle) einige Verbesserungen mit sich bringt. Gleichzeitig ist es von Vorteil, dass die Distribution schon ca. ein Jahr im Einsatz ist und somit ein Grossteil der verfügbaren Packages von ROS schon auf die Version Kinetic porträtiert wurden.

3.2 Auswahl IDE

Grundsätzlich kann Sourcecode in ROS, wie jede andere Software auch, mit einem Texteditor und einem passenden Compiler generiert werden. Für ROS stehen eine Vielzahl an IDEs[1] zur Verfügung. Aufgrund von bereits vorhandener Erfahrung wurde Visual Studio Code verwendet, welches mit einem Plugin erweitert werden kann um CMakeFiles und auch ROS spezifische Files besser zu unterstützen.

3.3 Installation

Die Installation von ROS und ROS-I können in Ubuntu über apt[2] installiert werden. Eine detaillierte Installationsanleitung ist auf dem ROS-Wiki[3] zu finden. Hier wird nur auf die wichtigsten Details eingegangen, da die Installationsanleitung eher umfangreich ist. Wichtig bei der Installation ist, das für ROS und ROS-Industrial die selben Distributionen installiert werden, zudem ist es zu empfehlen alle ROS und ROS-I Packages zu installieren. Dies verhindert spätere Komplikationen durch nicht vorhandene Packages. Nach der Installation von ROS sollten direkt die Abhängigkeiten von ROS initialisiert (`rosdep init`) und upgedatet (`rosdep update`) werden.

[1] Integrated Development Environments
[2] Advanced Packaging Tool
[3] ROS: `http://wiki.ros.org/kinetic/Installation`
ROS-I:`http://wiki.ros.org/Industrial/Install`

3.4 Workspace

Nach der Installation muss ein Workspace eingerichtet werden welcher für Catkin konform ist. Dies über die IDE möglich, geht aber am einfachsten über folgende Konsolenbefehle:

```
$ source /opt/ros/kinetic/setup.bash  # Source ROS setupscript
$ mkdir -p ~/catkin_ws/src        # erstellen benötigter Ordner
$ cd ~/catkin_ws/
$ catkin_make              # compilieren
$ source devel/setup.bash        # Source Workspace setupscript
```

Codesnippet 3.1: Konsolenbefehle zur Generierung eines Catkin-Workspace

Im Workspace sollten sich nun die Unterordner build und devel befinden, gleichzeitig sollte sich im Ordner src die Datei CMakelists.txt befinden. Das Setupscript eines Workspaces muss beim starten eines neuen Terminals immer neu eingelesen werden, es bietet sich an diese Zeile im File `~/.basrc` hinzuzufügen, damit dies nicht mehr notwendig ist.

3.5 Nützlichetools

In ROS gibt es eine Vielzahl nützlicher Konsolenbefehle und Tools hier eine kurze Auflistung der in dieser Arbeit meist gebrauchten.

3.5.1 rqt

Bei rqt handelt es sich um ein sehr mächtiges Tool in ROS, welches es ermöglicht auf fast alle Datenströme (messages, topics, services) zuzugreifen und diese zu visualisieren. Das Standartmenü kann über die Konsole mit dem Befehl `rqt` gestartet werden, dazu muss jedoch bereits ein ROS-Master aktiv sein. Mithilfe diverser Plugins können weitere nützliche Funktionen von rqt genutzt werden. Eine der Hilfreichsten ist dabei rqt_graph welche mit dem gleichlautenden Konsolenbefehl gestartet werden kann. Mit rqt_graph können alle aktiven Nodes und die auf den Topics publizierten Nachrichten visualisiert werden. Somit kann schnell und einfach überprüft werden, ob ein Node läuft und ob Topics vorhanden sind.

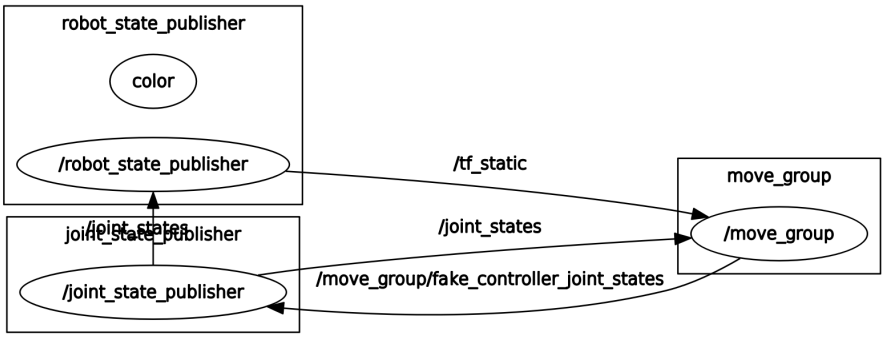

Abb. 3.1: Ausgabe von rqt_graph

3.5.2 Node

rosnode list Gibt eine Liste aller laufenden Nodes aus

rosnode info nodename Gibt Informationen über einen laufenden Node aus

rosnode ping nodename Pingt einen laufenden Node

rosnode kill nodename Beendet einen laufenden Node

3.5.3 Topics

rostopic list Gibt eine Liste mit allen registrierten Topics aus

rostopic echo /topicname Gibt alle auf dem Topic publizierten Nachrichten aus

rostopic hz /topicname Gibt die Frequenz an mit der Publiziert wird

rostopic pub /topicname data Publiziert auf einem Topic die mitgegebenen Daten

3.5.4 Parameterserver

rosparam list Gibt eine Liste aller gespeicherten Parameternamen aus

rosparam get /paramname Gibt den gespeicherten Variabelwert zurück

rosparam set /paramname Über-/schreibt einen neuen Wert auf den Parameterserver

rosparam delete /paramname Löscht den angegebenen Parameter vom Server

4 Industrie Demonstrator

In der Bachelorarbeit von Andrin Meister und Christian Hartmann wurde ein Grossteil des Aufbaus und der Funktionalitäten des Industrie 4.0 Demonstrators definiert. In den folgenden Abschnitten wird kurz auf die für den Montageprozess mit dem Sechsachsroboter der Firma Stäubli wichtigsten Punkte eingegangen.

4.1 Physikalische Schnittstellen

Der Industrie Demonstrator ist modular Aufgebaut, es gibt 3 trennbare Module, die Montagezelle mit dem Industrieroboter befindet sich in der mittleren Zelle. Der Arbeitsraum des Roboters ist nahezu nicht durch andere Systeme im Demonstrator beschränkt. Die einzigen physikalischen Interaktionen mit dem gesamt System bestehen aus dem Greifen der Teile aus den Teileträgern, welche auf Linearschlitten montiert sind. Auch im Mittelteil der Anlage wird ein Portalroboter verbaut, welcher sich in der Nähe des Sechsachsroboters bewegt.

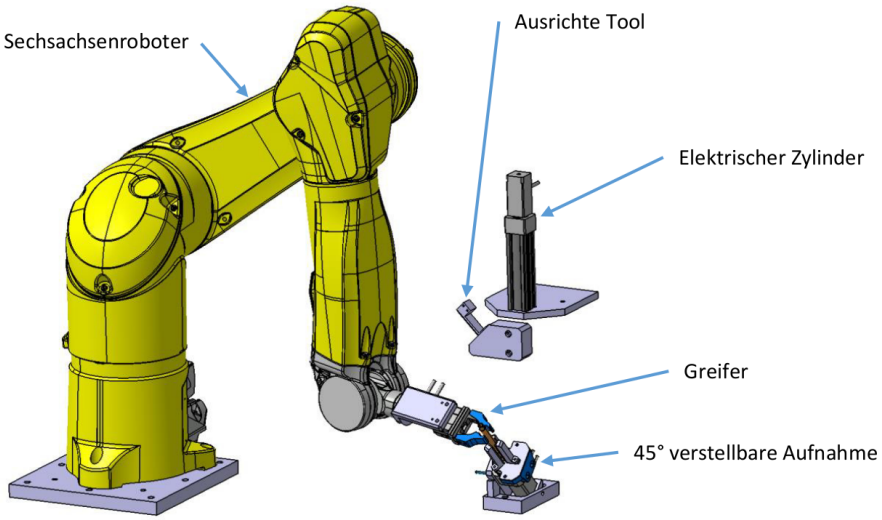

Abb. 4.1: Hauptelemente der Montagestation [22]

4.2 Software Schnittstellen

Der Grossteil der Anlage wird über Siemens SPS Steuerungen angesteuert, welche über PROFINET kommunizieren. Eine Einbindung dieses Protokolls in eine Linux/ROS Umgebung ist sehr komplex. Aus diesem Grund wurde mit den restlichen am Industrie 4.0 Demonstrator beteiligten Personen definiert, dass der der Linux PC gegen Aussen ein Webserver zur Verfügung stellt. Der zentralen Steuerung der Anlage soll es dabei möglich sein, einen Montagevorgang auszulösen sowie den Status des Roboters abzufragen.
Von ROS aus muss zudem die Ansteuerung des Dreibackengreifers, der Presse, sowie des Greifers am Roboterarm geschehen. Bei all diesen Aktuatoren handelt es sich um Servomotoren der Firma SMC welche mit Controllern mit einem EtherCAT Interface angesteuert werden können.

4.3 Kugelschreibermontage

Der Kugelschreiber, welcher zusammengebaut werden soll, besteht aus insgesamt aus fünf Teilen. Diese Einzelteile liegen definiert im Schlitten des Transportsystems, welche an einer vordefinierten Stelle im Arbeitsbereich des Roboters anhalten. Diese Teileträger befinden sich während des ganzen Montageprozesses an der selben Position. Der Montageablauf besteht aus folgenden Schritten:

1. Vorderhülle in Dreibackengreifer positionieren
2. Vorderhülle mithilfe des ausrichte Tools in korrekte Position drehen
3. Feder in Vorderhülle einführen
4. Miene in Vorderhülle einführen
5. Arretierung aufsetzen
6. Dreibackengreifer inkl. Kugelschreiber in senkrechte Lage drehen
7. Deckel positionieren und aufpressen
8. Dreibackengreifer inkl. Kugelschreiber in 45°Lage drehen
9. Kugelschreiber entnehmen

Der Grundsätzliche Ablauf für die Montage des Kugelschreibers ist durch die Geometrie des Kugelschreibers gegeben, wie genau diese Teile gegriffen werden und wie die Montagestation aufgebaut ist wurde in der Bachelorarbeit von Andrin Meister und Christian Hartmann definiert und sind somit für diese Arbeit vorgegeben. Für genauere Details zur Auswahl und Definition ist die Bachelorarbeit hinzuzuziehen.

5 Implementierung MoveIt! Package

Im folgenden Kapitel wird anhand des Stäubli Roboters TX2-60l erläutert, wie das MoveIt! Package für ROS-I erzeugt wird. Dieses Vorgehen wurde mit leichten Abänderungen auch auf den IRB120 von ABB und den UR3 von Universal Robots angewendet. Es ist wäre auch möglich andere Roboter so in den Industrie Demonstrator einzubinden.

Zur Erzeugung des MoveIt! Packages wird ein .xacro File benötigt. Für den IRB120 und UR3 sind diese Pakete bereits im ROS-I Gitrepository vorhanden, wobei das Package für den IRB120 unter dem Status experimentell aufgeführt wird. Die für den Stäubliroboter generierten Files werden im Ordner /staubli_expermiental/staubli_tx260l_support abgelegt. Für den Stäubli Roboter musste ein Package aufgesetzt werden. Dazu wurden folgende Schritte durchgeführt:

1. Erstellen Visual- und Kollisionsmodelle
2. Erstellen .xacro File
3. Generieren MoveIt! Package
4. Anpassungen für ROS-I

5.1 Erstellen Visual- und Kollisionsmodelle

5.1.1 Roboter und Greifer

Die CAD-Modelle des TX2-60l wurden von der Website von Stäubli bezogen, anschliessend musste an den Modellen gemäss den ROS/ROS-I Konventionen einige Anpassungen gemacht werden. Diese beinhalten vor allem Anpassungen am Koordinatensystem. Die ROS-Konventionen bestimmen, dass relativ zu den Körpern die x-Achse nach vorne Zeigt, die y-Achse nach links und die z-Achse nach oben.
Damit an Rechenleistung gespart werden kann, werden für die Kollisionsberechnung vereinfachte CAD-Modelle verwendet. Diese wurden mithilfe der Software Meshlab, welche gratis verfügbar ist, erstellt. Meshlab bietet diverse Tools und Funktionen um .stl-Files, sowie auch andere Tesselationsfiles zu bearbeiten. Für die Kollisionsfiles werden die CAD-Files mithilfe des Tools 'Convex Hull' vereinfacht (Vergleich Abbildung 5.2 und 5.1). Die erstellten Modelle werden in den entsprechenden Unterordnern abgelegt.

Das selbe Vorgehen wurde auch für den Greifer durchgeführt, bei diesem wurde die Convex Hull jedoch nur auf den hinteren Teil des Greifers angewendet.

5.1.2 Anlage

Aufgrund der Grösse und Komplexität der Anlage ist auch das CAD-Modell sehr gross. MoveIt! hat starke Schwierigkeiten sehr grosse Files zu laden, aus diesem Grund wurde der visuelle Teil der Anlage mithilfe von Meshlab stark vereinfacht. Somit konnte die Dateigrösse von 240MB auf circa 50MB verringert werden, diese Grösse lässt sich einigermassen gut importieren und visualisieren.
Für das Kollisionsmodell wurde nur der mittlere Teil der Anlage verwendet und stark vereinfacht. Ein Grossteil wird dabei einfach als Box dargestellt (siehe Abbildung 5.4).

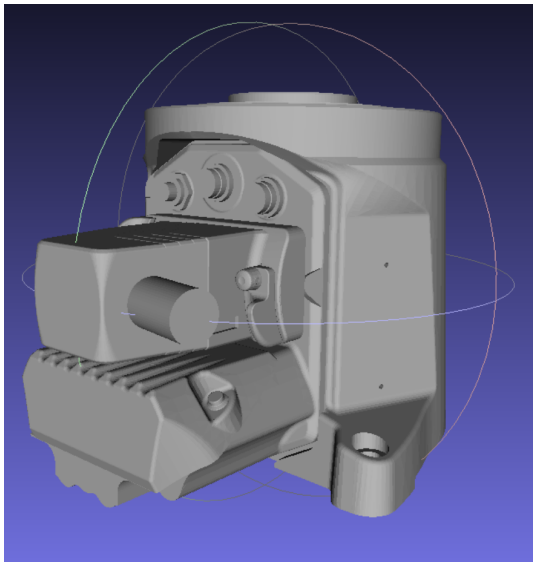

Abb. 5.1: Visuelles Modell der Base TX2-60l

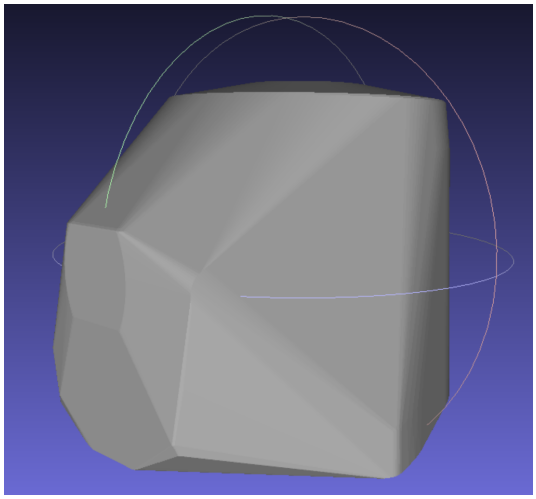

Abb. 5.2: Kollisions Modell der Base TX2-60l

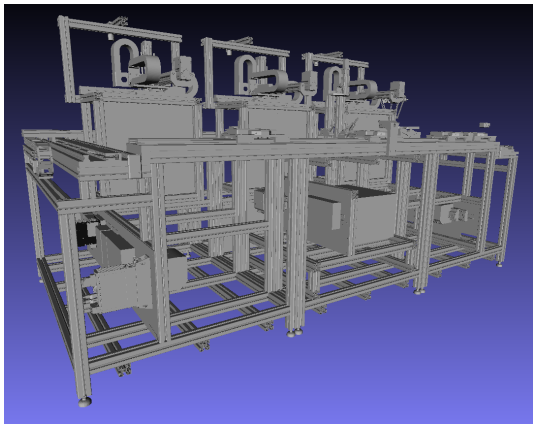

Abb. 5.3: Visuelles Modell

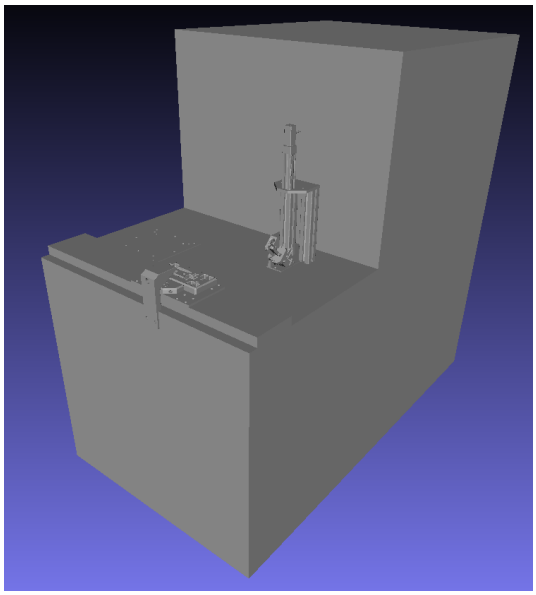

Abb. 5.4: Kollisions Modell

5.2 xacro Files

Für die Einbindung des Roboters in die Montageanlage wurden vier unterschiedliche xacro-Files erstellt. Das erste beschreibt den Roboter, das zweite den Greifer und das dritte beschreibt die Anlage. Diese drei Files werden in einem übergeordneten File, welches die ganze Anlage beschreibt eingebunden. Dies ermöglicht es den Roboter in der Montagezelle einfach auszuwechseln.

5.2.1 Roboter

Zur Beschreibung des Roboters wird ein xacro-macro File erstellt, die dafür benötigten Masse werden aus dem Datenblatt des TX2-60l bezogen. Für die Massenträgheitsmomente der einzelnen Glieder konnten keine Angaben gefunden werden. Auch für den maximalen Kraftaufwand (Effort) eines Gelenkes wurden keine Angaben Gefunden. Aus diesem Grund wurden die selben Werte wie

die des Vorgängermodells TX-60l verwendet, da dieser in etwa die gleichen Abmessungen wie sein Nachfolger besitzt. Falls die Bewegungsabläufe mit dem echten Roboter nicht Konform sind müssen allenfalls diese Werte angepasst werden. Im folgenden wird je ein Beispiel für ein Gelenk und ein Glied des Roboters gezeigt, das komplette File ist im Anhang unter 2.3 zu finden.

```
<link name="${prefix}base_link">
  <visual>
    <origin xyz="0 0 0" rpy="0 0 0" />
    <geometry>
      <mesh filename="package://staubli_tx260l_support/meshes/tx260l/visual/base_link.stl" />
    </geometry>
    <xacro:material_staubli_ral_melon_yellow />
  </visual>
  <collision>
    <origin xyz="0 0 0" rpy="0 0 0" />
    <geometry>
      <mesh filename="package://staubli_tx260l_support/meshes/tx260l/collision/base_link.stl" />
    </geometry>
  </collision>
  <inertial>
    <mass value="5.76415" />
    <origin xyz="-0.010284 -0.000676 0.087340" rpy="0.0 0.0 0.0" />
    <inertia ixx="0.000025" ixy="-0.000001" ixz="-0.000002" iyy="0.000034" iyz="-0.0000001"
    ↪  izz="0.000029" />
  </inertial>
</link>
```

Codesnippet 5.1: Definition des Baselinks TX2-60l

```
<joint name="${prefix}joint_2" type="revolute">
  <origin xyz="0.0 0.130 0.375" rpy="0 0 0" />
  <parent link="${prefix}link_1" />
  <child link="${prefix}link_2" />
  <axis xyz="0 1 0" />
  <limit lower="-2.22529" upper="2.22529" effort="130.0" velocity="6.719" />
  <dynamics damping="0.0" friction="0.0" />
</joint>
```

Codesnippet 5.2: Definition der zweiten Achse TX2-60l

Anschliessend muss ein neues File generiert werden, welches das vollständig definierte tx260l_macro.xacro einschliesst. Damit aus diesem ein URDF mithilfe der folgenden Kommandozeileneingabe erstellt werden kann.

```
rosrun xacro xacro.py tx260l.xacro > tx260l_generated.urdf
```

Codesnippet 5.3: Generierung des URDF aus xacro

Schlussendlich müssen sich im Unterordner folgende drei Files befinden.

```
urdf
├── tx260l_generated.urdf
├── tx260l_macro.xacro
└── tx260l.xacro
```

5.2.2 Greifer

Der Greifer besteht, wie auch der Roboter, aus einem Visuellen und einem Kollisionsmodell. Zudem besitzt der Greifer zwei unterschiedliche Greiferframes auf welche die move_group planen kann, somit ist es möglich den Greifer richtig zu Positionieren um die Teile zu greifen. Die Positionen der beiden Frames wurden in der Abbildung 5.5 in Form zweier Kugeln visualisiert.

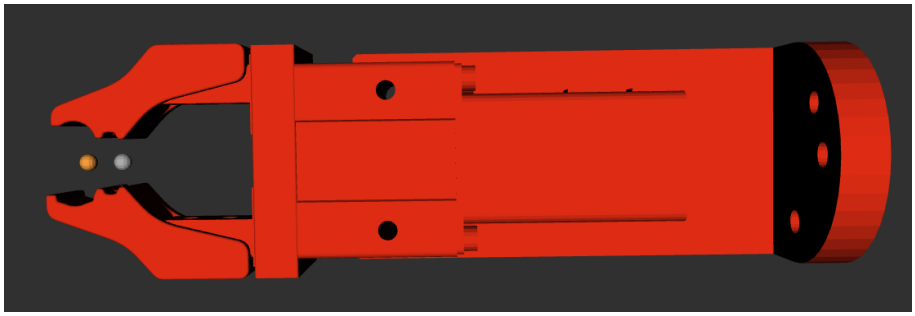

Abb. 5.5: Visualisierung der Greiferframes

5.2.3 Unterbau

Im xacro der Anlage wird nur das Visuelle Modell der Anlage geladen, da ansonsten die Ladezeiten beim starten der move_group extrem hoch sind und diese oft abstürzt. Das Kollisionsobjekt wird sobald die move_group gestartet ist eingebunden (siehe Abschnitt 6.2.3). Falls bei einem xacro kein Kollisionsobjekt definiert wird, wird automatisch der visuelle Teil auch für die Kollisionsberechnung verwendet. Aus diesem Grund wurde als Kollisionsobjekt eine Kugel mit Radius 0.1 *mm* definiert und so positioniert, das diese sich in der Base des Roboters befindet.

```
<collision>
  <origin rpy="0 0 0" xyz="0 0 0.005" />
  <geometry>
    <sphere radius="0.0001" />
  </geometry>
</collision>
```

Codesnippet 5.4: Dummy Kollisionsobjekt der Anlage

5.2.4 Gesammte Anlage

Die generierten Einzelteile der Anlage werden anschliessend in einem einzelnen .xacro-File eingebunden, als Ankerpunkt für den Greifer, Roboter und den Unterbau wird ein leerer Link mit dem Namen World definiert.

```
<?xml version="1.0" ?>
  <robot name="cell_tx260l" xmlns:xacro="https://ros.org/wiki/xacro">
  <xacro:include filename="$(find cell_support)/urdf/gripper_definition.xacro"/>
  <xacro:include filename="$(find cell_support)/urdf/unterbau.xacro" />
  <xacro:include filename="$(find staubli_tx260l_support)/urdf/tx260l_macro.xacro"/>

  <link name="world"/>

  <!-- init robots -->
  <xacro:gripper_definition prefix="" />
  <xacro:unterbau prefix=""/>
  <xacro:staubli_tx260l prefix="" />

  <!-- Joints -->
  <joint name="world_to_tx260l" type="fixed">
    <parent link="world" />
    <child link="base_link"/>
    <origin xyz="0 0 0" rpy="0 0 0"/>
  </joint>
  <joint name="world_to_unterbau" type="fixed">
    <parent link="world" />
    <child link="unterbau_base"/>
    <origin xyz="0 0 0" rpy="0 0 0"/>
```

```
  </joint>
  <joint name="tool0_to_gripper" type="fixed">
    <parent link="tool0" />
    <child link="gripper_base" />
    <origin xyz="0 0 0" rpy="0 1.57 0" />
  </joint>
</robot>
```

Codesnippet 5.5: .xacro-File der gesammten Anlage

5.3 MoveIt! Konfiguration

Das erstellen der MoveIt! Konfiguration muss für alle Roboter welche in der Anlage eingebaut werden sollen durchgeführt werden, die Einstellungen sind dabei für alle Roboter identisch. Nachfolgend sind alle vorgenommenen Einstellungen aufgelistet, welche im Setup Assistant vorgenommen wurden.

Self-Collisions Als erstes wird der Setup Assistant gestartet und das .xacro-File geladen, anschliessend kann im Tab Self-Collisions die Kollisionsmatrix erstellt. Dabei kann die Sampling Density auf High gestellt werden.

Virtual Joints Als nächstes muss ein Virtual Joint definiert werden, dieses verbindet die simulierte Welt mit dem geladenen .xacro-File, als Child Link wird dabei der definierte Ankerlink World gewählt, das Parent Frame ist World, und der Joint Type ist fixed. Als Name wurde hier jeweils FixedBase gewählt, dieser kann jedoch frei gewählt werden.

Planning Groups Es müssen zwei Planungsgruppen erstellt werden, eine small_gripper und eine gripper. Je eine für die unterschiedlichen Frames des Greifers. Beide Planungsgruppen werden als Kinematische Kette definiert mit Start beim base_link des Roboters und als Ende das entsprechende Frame. Als Kinematischersolver kann aus drei Standard Solvern gewählt werden und dem zusätzlich installierten Solver von TRACLabs. Für alle Roboter und Kinematischen Ketten wird der Solver 'trac_ik_kinematics' von TracLab gewählt. Es hat sich in Tests gezeigt, dass der Solver von TRACLabs, im Vergleich zu den Standardsolvern, fast immer eine Lösung findet. Die Einstellungswerte des Solvers wurde dabei auf den Standartwerten belassen.

Robot Poses Im Tab Robot Poses können Positionen des Roboters definiert werden, welche im GUI der move_group direkt ausgewählt werden können. Hier wurden zwei Posen definiert, eine mit allen Gelenken in Nullposition und eine Warteposition.

End Effectors Im Abschnitt End Effectors muss nichts eingestellt werden. Hier würden Endeffektoren, die von der move_group gesteuert werden definiert werden.

Passive Joints Im Abschnitt Passive Joints muss nichts angepasst werden. Hier würden Gelenke definiert werden, welche nicht angetrieben sind.

Author Information Es müssen Namen und Emailadressen des Erstellers diese Packages angegeben werden, ansonsten lässt sich das Setup nicht beenden.

Configuration Files In diesem Setup muss der Pfad angegeben werden, wo das Package abgespeichert werden soll, der Paketname muss, gemäss den ROS-Konventionen, dabei auf '_moveit_config' enden.

5.4 Anpassungen ROS-I

Für ROS-Industrial müssen nach dem Erstellen des Packages mit dem Setup Assistant einige Dateien hinzugefügt, respektive abgeändert werden.

controllers.yaml Als erstes muss die Datei 'controllers.yaml' im config Ordner erzeugt werden, welche den ROS-Node 'Controller Manager' zur Verfügung stellt der von der move_group gebraucht wird. Die Datei ist für alle Roboter gleich. Es muss jedoch beachtet werden, dass die Bezeichnungen der Gelenke mit deren des Roboters übereinstimmen müssen.

```
controller_list:
      - name: ""
      action_ns: joint_trajectory_action
      type: FollowJointTrajectory
      joints: [joint_1, joint_2, joint_3, joint_4, joint_5, joint_6]
```

Codesnippet 5.6: controller.yaml

controller_manager.launch Das .launch-File um den Controller zu starten ist nach dem Erstellen der Konfiguration zwar vorhanden jedoch ist es leer. Das File muss für jeden Roboter spezifisch angepasst werden. Es benötigt den Pfad zum jeweiligen 'controller.yaml'. Nachfolgend eine Vorlage, bei welcher der Teil in den eckigen Klammern ergänzt werden muss.

```
<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 [robot_moveit_config])/config/controllers.yaml"/>
</launch>
```

Codesnippet 5.7: Vorlage '..controller_manager.launch'

moveit_planning_execution.launch Als letztes muss ein .launch-File erzeugt werden, welches alle für ROS-I benötigten Nodes startet. Eine Vorlage dieses Files ist im Anhang unter 2.2 zu finden. Auch in diesem File müssen einige Teile spezifisch auf die Roboter angepasst werden.

5.5 ROS

Damit ROS-I mit dem Roboter Kommunizieren kann muss auf den Robotersteuerungen Anpassungen vorgenommen werden.

ABB IRB 120 Für den ABB IRB120 ist es nötig einen ROS Server auf dem Controller des Roboters zu installieren. Es müssen einige neue Tasks auf der Steuerung erstellt werden sowie ein RAPID-Programm auf die Steuerung geladen werden. Eine sehr detaillierte Anleitung dazu ist auf den ROS-Wikipages[1] zu finden.

Stäubli TX2-60l Da der Stäubli Roboter bei Abschluss dieser Arbeit noch nicht geliefert wurde können keine konkreten Angaben über die nötigen Anpassungen auf dem Controllers gemacht werden. Es stehen Driver für die Steuerung CS8 zur Verfügung, gemäss Informationen von Stäubli wird der Roboter mit einer Steuerung vom Typ CS9 geliefert. Bei Erhalt des Roboters und der Steuerung müssen diese mit ROS getestet werden und allenfalls der Server für die Steuerung angepasst werden. Die benötigten Files sowie eine Anleitung in Form eines Readmes befinden sich im Ordner /staubli_expermintal/staubli_val3_driver.

[1] http://wiki.ros.org/abb/Tutorials/InstallServer

Universal UR3 Für den Universal Robot UR3 muss keine zusätzliche Software installiert werden, da Universal Robot Standardmässig auf ihrer Steuerung bereits ROS Messages verarbeiten kann. Somit genügt es auf der Robotersteuerung das Netzwerk zu konfigurieren respektive kontrollieren ob dieses eingeschaltet ist.

6 Implementierung Montage

In diesem Kapitel wird auf die Implementierung aller notwendiger Softwarekomponenten eingegangen, damit im Industriedemonstrator Kugelschreiber montiert werden können.

6.1 Grundüberlegungen

Der Roboter muss fähig sein einem Montageablauf zu folgen, um Kugelschreiber montieren zu können. Dieser wird in einem eigenen Package cell_core implementiert. Dieses Package erzeugt die move_group und kann Kollisionsobjekte in der Planungsszene generieren. Ein zweites Package ist für die Ansteuerung der SMC Aktuatoren zuständig. Dieses Package smc_grippers ist die Schnittstelle zwischen der ROS-Kommunikation und der Kommunikation über EtherCAT. Für die Schnittstelle zur restlichen Anlage steht der in Abschnitt 4.2 bereits erwähnte HTTP-Server zur Verfügung. Dieser befindet sich im Package http_server.

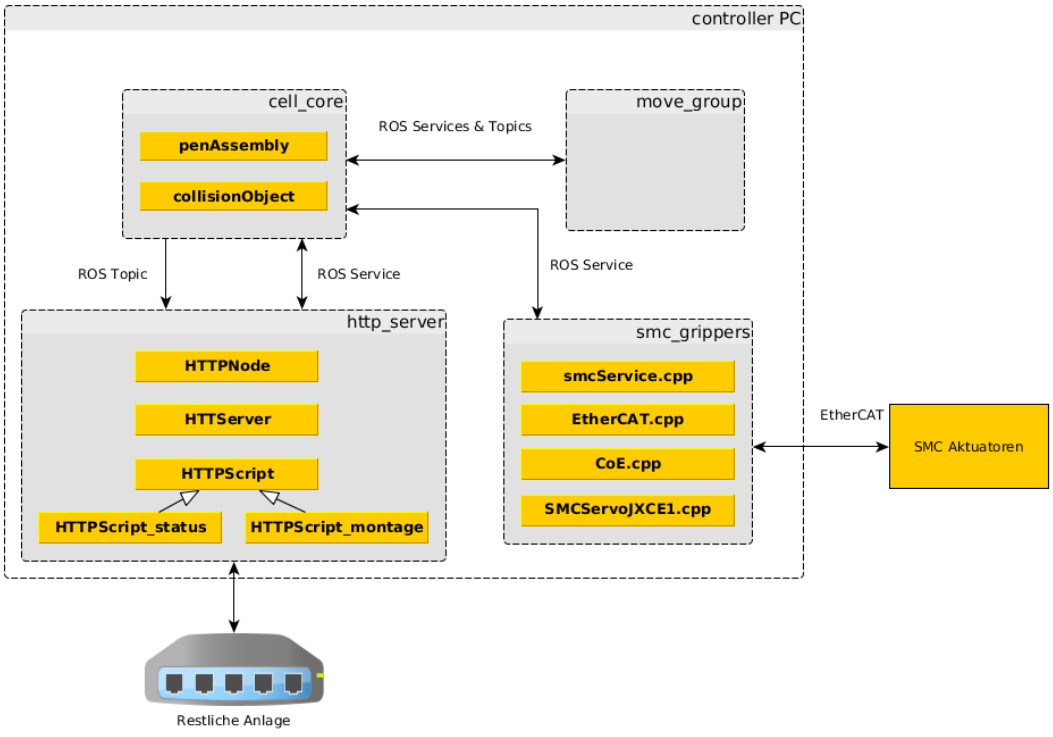

Abb. 6.1: Übersicht Packages und Kommunikation

6.2 cell_core

Das Package cell_core ist das Kernelement der Implementierung von ROS. Das Package konstruiert die move_group und kommuniziert anschliessend mit derselben um den momentan in der Anlage verbauten Roboter in Bewegung zu bringen. Das Package besteht aus einem Node 'penAssembly' und einer Klasse 'collisionAdder'. Das Package publiziert beim ROS-Master zwei Services und ein Topic, mit welchen es möglich ist mit dem Package zu kommunizieren.

6.2.1 Topics

Der Node publiziert regelmässig(10 Hz) den momentanen Anlagenstatus auf dem Topic /penAssembly/status. Die Publizierte Nachricht enthält die zwei booleschen Variablen idle und error. Welche je nach Zustand der Anlage gesetzt werden.

6.2.2 Services

Das Package cell_core bietet zwei Services an, der erste Service startet den Zusammenbau eines Kugelschreibers. Der zweite Service ermöglicht es den Roboter an einen bestimmten Punkt und Orientierung zu bewegen.

moveRobotToPose Zum jetzigen Stand der Arbeit registriert sich bei diesem Service kein Node. Der Service wurde nur gebraucht um den Roboter über die Konsole mithilfe von `$ rosservice call /penAssembly/moveRobotToPose "x: ,y: ,z: ,oW: ,oX: ,oY : ,oZ: "` zu einer bestimmten Position und Orientierung zu bewegen. Die Methode beachtet bei der Planung der Bewegung Kollisionsobjekte. Der Service Antwortet mit einem Boolean ob die Position erreicht wurde (true) oder nicht (false). Zum Aufrufen des Services müssen die Koordinaten sowie die Orientierung angegeben werden. Der Service ruft die Methode 'moveRobotCallback' auf.

montage_service Der Service montage_service ermöglicht es den Start der Montage auszulösen. Dabei muss der Nachricht der Offset der Wagen in Metern, sowie die gewünschte Ausgabestelle für den fertig montierten Kugelschreiber mitgegeben werden. Momentan ist die Ausgabe an verschiedene Stellen nur implementiert es existieren in der Anlage aber noch keine effektiven Ausgabestellen.

```
#request
float64 x
float64 y
float64 z
float64 oW
float64 oX
float64 oY
float64 oZ
---
#response
bool status
```

Codesnippet 6.1: moveRobotToPose Message

```
#request
float64 Offset
int64 Ausgabestelle
---
#response
int64 status
```

Codesnippet 6.2: montage_service Message

6.2.3 penAssembly

Initialisierung Beim starten des penAssembly Nodes wird als erstes ROS initialisiert und anschliessend ein asynchroner Spinner gestartet. Der asynchrone Spinner ist nötig, damit bei aktiver Planung von Trajektorien andere Anfragen immer noch verarbeitet werden können. Anschliessend wird die move_group gestartet und initialisiert, gefolgt vom Publizieren der beiden Services und des Topics. Als letztes wird ein Objekt der Klasse collisionObject erstellt und das Kollisionsobjekt der Anlage auf dem planning_interface publiziert. Wenn dieses Setup durchgeführt wurde wechselt der Node in einen Endlosloop und publiziert mit einer Frequenz von 10 Hz den Status der Anlage. Gleichzeitig wird auf den beiden publizierten Topics auf eine Anfrage gewartet.

```
int main(int argc, char **argv)
{
  ros::init(argc, argv, "penAssembly");
  ros::NodeHandle nh;
```

```
  ros::AsyncSpinner spinner(0); // Define Multithreadedspinner
  spinner.start();

  // Define MoveGroup and PlanningSceneInterface
  group.reset(new moveit::planning_interface::MoveGroupInterface("gripper"));
  //group->setPlannerId("RRTConnect");
  group->setEndEffectorLink("grasping_frame");
  group->setPlanningTime(1.5);
  group->setGoalJointTolerance(0.0001);
  group->setNumPlanningAttempts(3);
  group->setGoalOrientationTolerance(0.0001);
  group->setMaxVelocityScalingFactor(0.1);

  // Advertise Services at ROS-Master
  ros::ServiceServer service = nh.advertiseService("/penAssembly/montage_service",
  ↪  montageCallback);
  ros::ServiceServer moveservice = nh.advertiseService("/penAssembly/moveRobotToPose",
  ↪  moveRobotCallback);
  ROS_INFO("Service rdy!");

  // Publish the Status updater
  ros::Publisher penAssembly_pub = nh.advertise<cell_core::status_msg>("/penAssembly/status",
  ↪  1000);

  sleep(15); // to make sure move_group is up
  collisionObject coAdder;
  coAdder.addCell(group);
  sleep(10);  // only used because the large model takes some time to load

  ROS_INFO("Node Ready!");
  // Publish the State of the Assembly
  ros::Rate loop_rate(10); //Freq of 10 Hz
  while (ros::ok()) //aslong as node is alive
    {
      cell_core::status_msg msg;
      msg.idle = idle_;
      msg.error = error_;
      penAssembly_pub.publish(msg);
      loop_rate.sleep();
    }
}
```

Codesnippet 6.3: Initialisierung penAssembly Node

Methoden

Der Node bietet mehrere Methoden, zwei der Methoden werden durch die Services aufgerufen, die restlichen Methoden werden durch diese beiden Callbackmethoden aufgerufen. Alle in penAssembly.cpp implementierten Methoden ermöglichen es dem Roboter sich auf eine bestimmte Art und Weise zu bewegen.

montageCallback Die Methode montageCallback wird vom Service montage_service aufgerufen. Die Methode überprüft zu Beginn, ob die mitgegebenen Werte korrekt sind und ob die Anlage sich im idle-Modus befindet. Falls die Kriterien erfüllt sind startet die Anlage die Montage eines Kugelschreibers, ansonsten wird kein Kugelschreiber montiert und ein Fehler zurückgegeben. Innerhalb der Methode ist der Ablauf für die Montage des Kugelschreibers definiert. Der Roboter fährt jeweils mit freier Trajektorienplanung an ein Offset Punkt vor der eigentlichen Pose. Anschliessend wird der restliche Weg mit einem linearen Fahrbefehl gefahren.

initPoses Die Methode init Poses wird bei jedem Start eines Montage Prozesses aufgerufen, diese Initialisiert alle für die Montage benötigten Posen des Roboters. Ihr wird der Offset aus dem Servicecall mitgegeben, damit die Position des Schlittens korrigiert werden kann.

moveLinear Diese Methode bewegt den Roboter auf einer geraden Linie zwischen zwei Punkten. Der Startpunkt entspricht dabei immer der momentanen Position des Roboters, als Endpunkt kann entweder der Fahrweg von Startpunkt zu Zielpunkt in Metern in x,y,z mitgegeben werden, oder eine Pose zu welcher gefahren werden soll. Die Methode benötigt zudem den Plan des MoveGroupInterfaces. Die Trajektorie zwischen dem Start- und Endpunkt wird mithilfe der Methode computeCartesianPath berechnet, welche Teil der move_group ist. Der Trajektorie wird nur ausgeführt, falls der Planner mehr als 90 % des vorgegebenen Weges einhalten kann. Falls dies der Fall ist gibt die Methode true zurück.

```
bool moveLinear(double x, double y, double z, moveit::planning_interface::MoveGroupInterface::Plan
↪   &plan)
{
    std::vector<geometry_msgs::Pose> waypoints_tool;
    // Getting the current pose
    geometry_msgs::PoseStamped tempStampPose = group->getCurrentPose(group->getEndEffectorLink());
    geometry_msgs::Pose test_pose = tempStampPose.pose;
  // Defining Endpose
    test_pose.position.x += x;
    test_pose.position.y += y;
    test_pose.position.z += z;
    waypoints_tool.push_back(test_pose);

    moveit_msgs::RobotTrajectory trajectory_msg;
    double fraction = group->computeCartesianPath(waypoints_tool,
                                                  0.001, //eef_step
                                                  5,     // jump_threshold
                                                  trajectory_msg,
                                                  true); //avoid collisions
    plan.trajectory_ = trajectory_msg;
    ROS_INFO("Visualizing Cartesian Path (%2f%% acheived)", fraction * 100.0);
    if (fraction >= 0.90)
    {
        group->execute(plan);
        return 1;
    }
    else
    {
        return 0;
    }
    return -1;
}
```

Codesnippet 6.4: moveLinear mit x,y,z Offset

movePen Die Methode movePen wird gebraucht um den Kugelschreiber von der 45 °Lage in die Senkrechte Position aufzurichten.

rotateZ Die Methode rotateZ wird aufgerufen um den Kugelschreiber mit dem Werkzeuge auszurichten. Sie ist die einzige Methode, welche anstatt auf Positionen zu Planen direkt auf Gelenkwinkel plant. Der Methode muss ein Winkel in Radiant mitgegeben werden, um welchen sich die sechste Achse in positiver Richtung drehen soll.

```
void rotateZ(moveit::planning_interface::MoveGroupInterface::Plan &plan, double angle)
{
  std::vector<double> group_variable_values;
```

```
  // Saves the current Joint values in the vector group_variable_values
  group->getCurrentState()->copyJointGroupPositions(group->getCurrentState()->getRobotModel()->ge
  ↪  tJointModelGroup(group->getName()),
  ↪  group_variable_values);
  // Defining goal JointValue of joint6
  group_variable_values[5] += angle;
  group->setJointValueTarget(group_variable_values);
  // plan and execute
  group->plan(plan);
  group->execute(plan);
}
```

Codesnippet 6.5: Methode rotateZ

moveToPose Die Methode moveToPose bewegt den Roboter an die angegebene Pose, es ist möglich der Methode einen Offset in x,y und z mitzugeben. Falls der Planer beim ersten versuch keine Lösung findet wird die Planungszeit auf 10 s erhöht und ein neuer Planungsversuch gestartet. Falls immer noch keine Lösung für die Bewegung gefunden wird gibt die Methode false zurück.

6.2.4 collisionAdder

Die Klasse collisionAdder ermögtlicht es Kollisionsobjekte in die Planungsszene zu laden, dazu bietet sie vier Methoden an.

addCell Die Methode addCell wird kurz nach dem erstellen des Objekts beim Initialisieren des Nodes aufgerufen. Sie fügt der Planungsszene das Kollisionsmodell der Anlage hinzu, welches direkt aus dem .stl-File generiert wird.

```
void collisionObject::addCell(boost::shared_ptr<moveit::planning_interface::MoveGroupInterface>
↪  &group)
{
    // Generating Collision object from Mesh
    Eigen::Vector3d scaling_vector(0.001, 0.001, 0.001); // Scaling Vector
    moveit_msgs::CollisionObject co;
    co.header.frame_id = group->getPlanningFrame();
    co.id = "WorkCell";
    shapes::Mesh *m =
    ↪  shapes::createMeshFromResource("package://cell_support/meshes/1000_Anlage_collision.stl",
    ↪  scaling_vector);
    ROS_INFO("Mesh Loaded");

    shape_msgs::Mesh mesh;
    shapes::ShapeMsg mesh_msg;
    shapes::constructMsgFromShape(m, mesh_msg);
    mesh = boost::get<shape_msgs::Mesh>(mesh_msg);

    // Define position and orientation
    co.meshes.resize(1);
    co.mesh_poses.resize(1);
    co.meshes[0] = mesh;
    co.mesh_poses[0].position.x = -1.075;
    co.mesh_poses[0].position.y = 0.023;
    co.mesh_poses[0].position.z = -0.021;
    co.mesh_poses[0].orientation.w = 0.707;
    co.mesh_poses[0].orientation.x = 0.0;
    co.mesh_poses[0].orientation.y = 0.0;
    co.mesh_poses[0].orientation.z = 0.707;

    // Push onto vector and publish it on the szene
    co.meshes.push_back(mesh);
```

```
    co.mesh_poses.push_back(co.mesh_poses[0]);
    co.operation = co.ADD;
    std::vector<moveit_msgs::CollisionObject> collision_objects;
    collision_objects.push_back(co);
    planning_scene_interface.addCollisionObjects(collision_objects);
    return;
}
```

Codesnippet 6.6: addCell Methode

addBody Die Methode addBody publiziert einen Kollisionskörper am momentan aktiven Planningframe des Roboters und befestigt diesen direkt daran. Die Methode besitzt ein Switchstatement, welches einen Übergabewert auswertet. Mit diesem kann zwischen der vorderen Hülle (1) dem Deckel (2), der Miene (3) und dem Werkzeug (4) ausgewählt werden. Die Kollisionskörper werden wie bei der Methode addCell direkt aus den .stl-Files der Teile erstellt.

detatchBody Diese Methode entfernt ein zuvor publiziertes Kollisonsobjekt vom Roboter, entfernt es jedoch noch nicht aus der Plannungsumgebung. Auch hier können die Objekte wieder über den übergebenen Integer ausgewählt werden.

removeBody Die Methode entfernt ein sich in der Planungsszene befindendes Objekt ganz. Die Methode benötigt die selben Übergabewerte wie die beiden vorhergegangenen Methoden.

6.2.5 launch

Das Package enthält zwei launch-Files, das Erste startet alle benötigten Nodes für den Industriedemonstrator, verbindet sich aber nicht mit einem angeschlossenen Roboter. Das andere .launch-File startet auch alle benötigten Nodes, versucht aber zudem eine Verbindung mit dem angeschlossenen Roboter herzustellen (momentan ABB IRB120). Es wurden auch launch-files für den Stäubliroboter generiert, bei diesen konnte jedoch nicht getestet werden ob sie eine Verbindung mit dem Roboter herstellen, da der Roboter noch nicht geliefert wurde.

6.3 http_server

Der HTTP-Server basiert auf dem im SVN-Repository `https://triest.zhaw.ch/svn/iio/` vorhandenen Sourcecode. Es wurde zusätzlich ein ROS Node erstellt, welcher die für den Betrieb des HTTP-Servers nötigen Objekte generiert. Der Node erstellt zwei unterschiedliche Server auf den Ports 8080 und 8081, auf dem Port 8080 kann die Montage eines Kugelschreibers ausgelöst werden. Dazu muss die folgende URL mit den entsprechenden Werten aufgerufen werden:
`http://160.85.95.166:8080/cgi-bin/montage?offset=0.0&ausgabestelle=1`
Der Offset muss in Metern angegeben werden wobei positive Zahlen einen Offset in Fahrtrichtung der Wagen entsprechen.

Mit dem Webserver auf Port 8081 kann der Status des Roboters abgefragt werden, dazu kann die folgende URL aufgerufen werden:
`http://160.85.95.166:8080/cgi-bin/status`
Bei beiden Aufrufen ist allenfalls die IP-Adresse zu ändern.

```
#include <http_server/HTTPNode.h>

int main(int argc, char **argv)
{
    ros::init(argc, argv, "HTTP_server");
    ros::NodeHandle nh;
    ros::AsyncSpinner spinner(2); // Define Multithreadedspinner
```

```
    spinner.start();

    HTTPServer *httpServer = new HTTPServer(8080);
    HTTPServer *httpServer2 = new HTTPServer(8081);
    httpServer->start();
    httpServer2->start();
    httpServer->add("montage", new HTTPScript_montage());
    httpServer2->add("status", new HTTPScript_status());

    ROS_INFO("Http-Server is running!");

    ros::waitForShutdown();
}
```

Codesnippet 6.7: Main des HTTP-Servers

6.3.1 launch

Der Webserver kann über das launch-File 'httpNode.launch' gestartet werden, das launchfile Startet im falle eines Absturzes des HTTP-Nodes diesen direkt neu.

6.4 smc_grippers

Die Controller für die SMC Aktuatoren müssen über EtherCAT angesteuert werden. Durch die ROS Community sind einige Packages vorhanden, welche dazu gedacht sind EtherCAT Geräte anzusteuern. Diese basieren in der Regel entweder auf dem SOEM[1] der OpenEtherCAT Society, oder auf dem EtherCAT Treiber des PR2 von Willow Garage. Es wurde mit mehreren dieser verfügbaren Packete versucht eine erfolgreiche Kommunikation mit den EtherCAT Kontrollern von SMC aufzubauen. Diese Versuche blieben jedoch erfolglos. Aus diesem Grund wurde durch Herrn Dr. Marcel Honegger ein Treiber für die EtherCAT Kontroller von SMC geschrieben. Diese Sourefiles wurden anschliessend in ein ROS Node eingebunden, welcher für jeden Aktuator von SMC einen eigenen Service anbietet.

```
bool smcService::gripperCallback(smc_grippers::gripper_service::Request &req,
↪  smc_grippers::gripper_service::Response &res)
{
    gripper->enable();
    int pos = req.position;
    gripper->setTargetPosition(pos);
    gripper->disable();
    return 1;
}
```

Codesnippet 6.8: Callbackmethode des Greifers

Im momentanen Stand der Anlage ist noch keine Kommunikation mit den Kontrollern von SMC möglich, da es vom Linux PC nicht möglich ist, dass die Slaves in den EtherCAT status Operational wechseln.

6.4.1 Services

Die drei Services sind identisch aufgebaut und benötigen alle nur die gewünschte Position in Millimeter. Die Services antworten immer mit true, ausser die Services sind nicht erreichbar.

[1] Simple Open EtherCAT Master

6.4.2 launch

Der SMC Node kann mithife des launch-File 'smcService.launch' gestartet werden.

7 Resultate

7.1 Simulation Montageaufgabe

Die Montageaufgabe kann in RVIZ simuliert werden. Im momentanen Stand der Arbeit kann es Vorkommen, dass keine Lösungen für das Anfahren von gewissen Positionen gefunden werden.

7.2 Umsetzung Montageaufgabe

Die Montageaufgabe konnte in der Simulation mit den von ROS zur Verfügung gestellten Tools implementiert werden. Da die beiden Bachelorstudenten einige Teile für die Montagezelle falsch gezeichnet haben und diese somit nicht montiert werden konnten mussten die Teile neu modelliert und anschliessend 3D-gedruckt werden. Zusätzlich kamen Probleme mit der Genauigkeit der CAD-Modelle hinzu und das mit dem ABB IRB120 in der vorgesehenen Konfiguration der Anlage nicht alle Montagepunkte erreicht werden konnten. All diese Faktoren führten dazu, dass für ausführliche Tests der Simulation sowie realen Anlage die Zeit fehlte. Der Komplette Sourecode sowie die erstellten Packages und CAD-Files sind auf dem pool unter:`//shared.zhaw.ch/pools/t/T-IMS-SmartPro/Projekt/ROS-Industrial` zu finden.

7.3 Portierung Montageaufgabe

Da der Staubli Roboter erst nach Abschluss der vorliegenden Arbeit geliefert wird wurde die Montageaufgabe auf dem Roboter von ABB vom Typ IRB120 realisiert. Über eine mögliche Portierung der Montageaufgabe auf den Roboter von Stäubli können keine fundierten Resultate abgegeben werden. Es konnte ein Driver für den Controller vorbereitet werden, ob dieser auf der echten Steuerung wirklich funktioniert muss nach Erhalt des Roboters geklärt werden.

7.4 Evaluation ROS-Industrial

ROS und ROS-Industrial sind zwei extrem mächtige Tools, sie haben ihre Vorteile aber auch ihre Nachteile. In diesem Abschnitt wird auf die wichtigsten Eigenschaften von ROS-I eingegangen.

7.4.1 Verfügbarkeit von Paketen

ROS und ROS-Industrial werden zu einem sehr grossen Teil durch die Community erweitert und gewartet. Auf diversen Repositorys verteilt befinden sich eine sehr grosse Anzahl an Packages, welche diverse Funktionen und/oder Bibliotheken enthalten. Viele dieser Packages werden einigermassen gut gewartet und aktualisiert, fast genau so viele Packages sind jedoch fast verwahrlost.
Oft wird zum Implementieren einer Funktion auf diese grosse Anzahl an Funktionen und Bibliotheken zurückgegriffen, oder aber es werden andere OpenSource Ressourcen eingebunden wie zum Beispiel die OMPL. Dies bietet einige Vor- und Nachteile, einerseits sind viele Probleme schon bereits irgendwo entstanden und es bestehen Lösungen oder zumindest Lösungsansätze. Die Funktion, Stabilität und Wartung solcher Pakete ist jedoch überhaupt nicht gewährleistet und muss allenfalls selbst in die Hand genommen werden.

7.4.2 Zuverlässigkeit und Stabilität

Die von ROS-Industrial angebotenen Funktionen sind alle relativ zuverlässig, ausser der Descrates Planer. Dieser konnte selten eine vernünftige Lösung für die gestellten Trajektorienbahnen finden

und stürzte oft ab. Die Nodes welche generiert werden sind in sich auch stabil und bei einem Absturz eines Nodes wird aufgrund der Systemarchitektur von ROS ein kompletter Systemausfall verhindert. Trotzdem muss der Node anschliessend neu gestartet werden, was in einem Industriedemonstrator welcher später an einer Messe laufen soll nicht sehr von Vorteil ist.

7.4.3 Support

Der Support von ROS-Industrial setzt rein auf die Foren und Wikipages. Diese Foren und Wikipages sind Voll mit Informationen und viele Probleme welche während dieser Arbeit entstanden sind konnten über diese relativ schnell gelöst werden. Trotzdem kann es vorkommen, dass ein Problem entsteht bei welchem weder die Wikipages oder die Foren helfen können. In diesem Fall ist

7.4.4 Empfehlung ROS-I

ROS-Industrial bietet viele Funktionen an, welche für die Industrie gebraucht werden, die sehr ausgereift und durchdacht sind. Gleichzeitig scheint es aber bei einigen Teilen noch in den Kinderschuhen zu stecken. Im momentan scheint ROS-Industrial für die Anwendung im Industriedemonstrator nicht geeignet zu sein. Die Implementierung der Montageaufgabe ist zwar nicht ausserordentlich Komplex, jedoch konnten einige Probleme nicht oder nur mit behelfsmässigen Patches gelöst werden. Eine realisierung der Montageaufgabe mithilfe der vom Hersteller zur Verfügung gestellten Tools scheint hier wesentlich schneller und vor allem stabiler zu Funktionieren.

ROS-Industrial ist interessant für Anwendungen in welchen sich die Umgebungen ändern und diese mit Sensoren oder Kameras überwacht werden können. Dort kann ROS durch seine dynamische Bahnplanung und direkte Verarbeitung von Sensor- und Kameradaten punkten. Ein weiterer Vorteil von ROS ist die Freiheit mit einer Applikation unterschiedliche Robotertypen anzusteuern, somit wäre ROS sicherlich in einer Anlage mit mehreren unterschiedlichen Robotern geeignet.

7.5 Probleme und offene Punkte

7.5.1 SMC Aktuatoren

Beim Momentanen Stand der Arbeit konnte keine Kommunikation über EtherCAT mit den Aktuatoren von SMC hergestellt werden. Die Kommunikation über EtherCat mit den Kontrollern Funktioniert zwar, jedoch weigern sich die Kontroller vom EtherCAT Staus SSafe Operationalïn den Status Öperationalßu wechseln. Aus diesem Grund kann von ROS aus im Moment die Aktuatoren nicht angesteuert werden. Aus diesem Grund konnten die Greiferpositionen noch nicht bestimmt werden und es konnten somit auch keine Kugelschreiber in der Anlage zusammengebaut werden.

7.5.2 Crash Nodes

Es kann vorkommen, dass die move_group, RVIZ oder ein anderer Node sich während der Bahnplanung oder auch sonst "aufhängt". Bis zur Abgabe dieser Arbeit konnte die Ursache für dieses Vorkommen nicht eindeutig ermittelt werden.

7.5.3 Fahrgeschwindigkeit und Aborts

ROS berechnet aufgrund der maximalen Gelenkgeschwindigkeiten und des zu verfahrenden Weges die Zeit die es Braucht um die Ziel Position zu erreichen. Diese Zeit wird mit einem Faktor (Standardwert 1.2) multipliziert und ergibt die maximal erlaubte Plannungszeit. Falls diese vom Roboter überschritten wird wird durch die move_group das Ausführen der Trajektorie abgebrochen. Dies kommt beim ABB IRB120 in der Simulation nicht vor, jedoch kann es auf dem echten Roboter dazu kommen. Eine erhöhung des Faktors oder gar eine Deaktivierung dieser Funktion scheinen keine Auswirkungen auf dieses Abbruchkriterium zu haben.

8 Diskussion und Ausblick

8.1 Erreichung der Ziele

Nicht alle gesteckten Ziele für diese Arbeit konnten erreicht werden. Es konnte eine Arbeitsumgebung für ROS und ROS-Industrial eingerichtet werden, mit welcher die Implementierung der Montageaufgabe in den Industrie 4.0 Demonstrator durchgeführt werden konnte. Zur Implementierung der Montageaufgabe musste ein sehr breites Basiswissen von ROS und ROS-Industrial angelegt werden. Dies ermöglicht nun fundierte Aussagen über die Möglichkeiten und Einschränkungen von ROS-Industrial machen zu können.

Aufgrund von falsch gezeichneten und falsch bestellten Teilen durch die beiden Bachelorstudenten, welche das mechanische Konzept der Anlage entwickelten, konnte erst kurz vor Abgabe erste Tests mit dem Greifer am Roboter gemacht werden. Dies nach hinten verschoben Testphase verunmöglichte es grosse Änderungen an der Implementation zu machen. Da zum Zeitpunkt der Abgabe der Arbeit die Aktuatoren von SMC noch nicht über EtherCAT angesteuert werden konnten, konnte noch kein Kugelschreiber montiert werden.

8.2 Ausblick

Sobald der Stäubli Roboter geliefert wird und dieser in der Anlage verbaut ist, kann die Implementierung von ROS-Industrial auf dessen Steuerung beginnen. Die Weiterführung von ROS-Industrial als Steuerung des Knickarmroboters ist meines erachtens nur sinvoll, falls in weiterer Zukunft an der ZHAW oder am IMS im Sinne der Forschung weiter mit ROS gearbeitet werden will. Oder falls die Komplexität der Montageaufgabe sich erhöht, weil zum Beispiel über Vision Sensoren Objekte im Arbeitsraum erkennt werden können und dynamisch umfahren werden sollen.
Ansonsten ist die Implementierung des momentanen Montageprozesses schneller und effizienter mit den bisherigen Robotertools.

Literaturverzeichnis

[1] "Ros/introduction - ros wiki." http://wiki.ros.org/ROS/Introduction. (Accessed on 24.05.2017).

[2] "Ros.org | is ros for me?." http://www.ros.org/is-ros-for-me/. (Accessed on 24.05.2017).

[3] L. Joseph, *Mastering Ros for Robotics Programming*. PACKT PUB, 2015.

[4] I. Saito, "Distributions - ros wiki." http://wiki.ros.org/Distributions, 05 2017. (Accessed on 26.07.2017).

[5] K. Conley, "Nodes - ros wiki." http://wiki.ros.org/Nodes, 02 2013. (Accessed on 26.07.2017).

[6] K. Conley, "Master - ros wiki." http://wiki.ros.org/Master, 02 2012. (Accessed on 26.07.2017).

[7] D. Thomas, "Parameter server - ros wiki." http://wiki.ros.org/Parameter%20Server. (Accessed on 26.07.2017).

[8] "actionlib - ros wiki." http://wiki.ros.org/actionlib. (Accessed on 31.07.2017).

[9] "Rep 103 – standard units of measure and coordinate conventions (ros.org)." http://www.ros.org/reps/rep-0103.html. (Accessed on 31.07.2017).

[10] H. L. Allen, *Threaded Applications with the roscpp API*, pp. 51–69. Cham: Springer International Publishing, 2016.

[11] S. Chitta, *MoveIt!: An Introduction*, pp. 3–27. Cham: Springer International Publishing, 2016.

[12] "Concepts | moveit!." http://moveit.ros.org/documentation/concepts/. (Accessed on 25.07.2017).

[13] V. Lamoine, "Industrial/tutorials/create_a_moveit_pkg_for_an_industrial_robot - ros wiki." http://wiki.ros.org/Industrial/Tutorials/Create_a_MoveIt_Pkg_for_an_Industrial_Robot, 10 2015. (Accessed on 26.07.2017).

[14] D. M. Jia Pan, Sachin Chitta, "Fcl: A general purpose library for collision and proximity queries." in IEEE International Conference on Robotics and Automation, 2012.

[15] "Kinematics configuration tutorial — moveit_tutorials indigo documentation." http://docs.ros.org/kinetic/api/moveit_tutorials/html/doc/pr2_tutorials/kinematics/src/doc/kinematics_configuration.html?highlight=kdl. (Accessed on 30.07.2017).

[16] "traclabs / trac_ik — bitbucket." https://bitbucket.org/traclabs/trac_ik. (Accessed on 30.07.2017).

[17] "Available planners." http://ompl.kavrakilab.org/planners.html. (Accessed on 26.07.2017).

[18] S. M. LaValle, *Planning Algorithms*. Cambridge, U.K.: Cambridge University Press, 2006. Available at http://planning.cs.uiuc.edu/.

[19] "Current members — ros-industrial." http://rosindustrial.org/ric/current-members/. (Accessed on 22.07.2017).

[20] "Description — ros-industrial." `http://rosindustrial.org/about/description/`. (Accessed on 31.07.2017).

[21] "Industrial/supported_hardware - ros wiki."

[22] A. M. Christian Hartmann, "Industrie 4.0 Demonstrator," Bachelorarbeit, ZHAW School of Engineering, 2017.

Anhang

Im Anhang dieses Dokumentes befinden sich nur die in der Arbeit referenzierten Codeausschnitte sowie die Aufgabenstellung. Alle in dieser Arbeit generierten Files sind auf dem ZHAW pool unter: `//shared.zhaw.ch/pools/t/T-IMS-SmartPro/Projekt/ROS-Industrial` zu finden.

1 Offizielle Aufgabenstellung

Zürcher Hochschule
für Angewandte Wissenschaften
School of
Engineering
IMS Institut für
Mechatronische Systeme

Einsatz von ROS Industrial in einem Industrie 4.0 Demonstrator

Allgemeine Angaben zur Vertiefungsarbeit

Startdatum:	1. März 2017
Enddatum:	30. Juni 2017
Studierender:	Luis Meier, meierlui
Betreuer:	Marcel Honegger, honr
Credits:	12

Kurzbeschreibung

Industrieroboter, wie sie von diversen Herstellern in allen möglichen Grössen und Konfigurationen erhältlich sind, haben typischerweise eine herstellerspezifische Steuerung. Die Funktionalitäten dieser Steuerungen sind alle ähnlich, aber die Programmiersprachen sind alle unterschiedlich, wie z.B. RAPID bei ABB oder KRL bei KUKA Robotern. Bestrebungen, eine gemeinsame Roboter-Programmiersprache zu entwickeln sind bei den Roboterherstellern nicht zu erkennen.
Mit dem Robot Operating System (ROS) ist aber ein Framework für die Entwicklung von Software für Roboter entstanden, welches insbesondere für Anwendungen in der Mobilrobotik in den letzten Jahren sehr populär geworden ist. Dieses Framework bietet unter Anderem eine einfache Integration von externen Sensoren (z.B. Laserscanner), Algorithmen für die automatische Bahnplanung und Simulationen verschiedener Roboter.
Ein Ableger dieses Frameworks, ROS Industrial, konzentriert sich auf die Entwicklung von Anwendungen mit Industrierobotern. Es erlaubt, eine Anwendung weitestgehend unabhängig von einem spezifischen Robotermodell zu entwickeln, zu simulieren und schliesslich mit einem echten Roboter laufen zu lassen. Es hat den Anspruch, die Abhängigkeit von herstellerspezifischen Programmiersprachen aufzubrechen, und spezifische Roboter als abstrahierte Komponenten zu modellieren.
Im Rahmen dieser Vertiefungsarbeit soll eine konkrete Montageaufgabe für einen Industrie 4.0 Demonstrator mit einem Stäubli Industrieroboter realisiert werden, wobei als Programmiersprache C++ und das ROS Industrial Framework eingesetzt werden. Nach Möglichkeit soll aber keine Stäubli spezifische Software entwickelt werden.
Die Portierbarkeit dieser Anwendung soll schliesslich mit einem Wechsel des gewählten Roboters zu einem Universal Robots UR3 gezeigt werden.

Arbeitspakete

- Aufsetzen einer Entwicklungsumgebung mit ROS Industrial.
- Simulation einer einfachen Montageaufgabe des Industrie 4.0 Demonstrators, wobei als Roboter sowohl ein Stäubli Industrieroboter, wie auch ein Universal Robots UR3 und ein ABB IRB120 verwendet werden sollen.
- Entwicklung und Umsetzung der Montageaufgabe mit dem Stäubli Industrieroboter.
- Portierung dieser Montageaufgabe auf einen Universal Robots UR3, inkl. Vergleichstests mit dem Stäubli Roboter.
- Evaluation von ROS Industrial, mit einer Beurteilung dessen Möglichkeiten und Einschränkungen für industrielle Anwendungen.

2 Sourcecodefiles

2.1 Vollständige CMakeLists

```
cmake_minimum_required(VERSION 2.8.3)
project(cell_core)

## Add support for C++11
add_definitions(-std=c++11)

## Find catkin macros and libraries
## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS xyz)
## is used, also find other catkin packages
find_package(catkin REQUIRED COMPONENTS
  roscpp
  std_msgs
  message_generation
  geometry_msgs
  moveit_core
  moveit_ros_planning
  moveit_ros_planning_interface
  pluginlib
  cmake_modules
  genmsg

)

find_package(Boost REQUIRED
  system
  filesystem
  date_time
  thread
)

find_package(
  Eigen3 REQUIRED
)

################################################
## Declare ROS messages, services and actions ##
################################################

# Generate messages in the 'msg' folder
add_message_files(
  FILES
    status_msg.msg
)

## Generate services in the 'srv' folder
add_service_files(
  FILES
    montage_service.srv
)

## Generate actions in the 'action' folder
# add_action_files(
#   FILES
#   Action1.action
#   Action2.action
# )

## Generate added messages and services with any dependencies listed here
generate_messages(
```

```
  DEPENDENCIES
   std_msgs
)

###################################
## catkin specific configuration ##
###################################
## The catkin_package macro generates cmake config files for your package
## Declare things to be passed to dependent projects
## INCLUDE_DIRS: uncomment this if you package contains header files
## LIBRARIES: libraries you create in this project that dependent projects also need
## CATKIN_DEPENDS: catkin_packages dependent projects also need
## DEPENDS: system dependencies of this project that dependent projects also need
catkin_package(
  INCLUDE_DIRS include
# LIBRARIES
  CATKIN_DEPENDS
    moveit_core
    moveit_ros_planning_interface
    interactive_markers
    message_runtime
    geometry_msgs
    roscpp
)

## Enable compiler warnings
set( CMAKE_CXX_FLAGS "-Wall -Wextra" )
set( CMAKE_CXX_FLAGS_DEBUG "-g -O0" )
set( CMAKE_CXX_FLAGS_RELEASE "-O3" )

###########
## Build ##
###########

## Specify additional locations of header files
## Your package locations should be listed before other locations
# include_directories(include)
include_directories(
  SYSTEM
    ${Boost_INCLUDE_DIR}
    ${EIGEN_INCLUDE_DIRS}
  include
    ${catkin_INCLUDE_DIRS}
)

link_directories(
  ${catkin_LIBRARY_DIRS}
)

install(DIRECTORY include/${PROJECT_NAME}/
  DESTINATION ${CATKIN_PACKAGE_INCLUDE_DESTINATION}
  PATTERN ".svn" EXCLUDE
)

## Declare a C++ library
add_library(collisionObject
  src/collisionObject.cpp
)

## Declare a C++ executable
add_executable(penAssembly
  src/penAssembly.cpp
)
```

```
## Add cmake target dependencies of the executable
## same as for the library above
add_dependencies(collisionObject ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})
add_dependencies(penAssembly ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})

## Specify libraries to link a library or executable target against
target_link_libraries(penAssembly
  ${catkin_LIBRARIES}
  collisionObject
)
```

2.2 moveit_planning_execution.launch

```
<launch>
  <!-- The planning and execution components of MoveIt! configured to run -->
  <!-- using the ROS-Industrial interface. -->

  <!-- Non-standard joint names:
       - Create a file [robot_moveit_config]/config/joint_names.yaml
           controller_joint_names: [joint_1, joint_2, ... joint_N]
       - Update with joint names for your robot (in order expected by rbt controller)
       - and uncomment the following line: -->
  <!-- <rosparam command="load" file="$(find [robot_moveit_config])/config/joint_names.yaml"/> -->

  <!-- the "sim" argument controls whether we connect to a Simulated or Real robot -->
  <!--  - if sim=false, a robot_ip argument is required -->
  <arg name="sim" default="true" />
  <arg name="robot_ip" unless="$(arg sim)" />

  <!-- load the robot_description parameter before launching ROS-I nodes -->
  <include file="$(find [robot_moveit_config])/launch/planning_context.launch" >
    <arg name="load_robot_description" value="true" />
  </include>

  <!-- run the robot simulator and action interface nodes -->
  <group if="$(arg sim)">
    <include file="$(find industrial_robot_simulator)/launch/robot_interface_simulator.launch" />
  </group>

  <!-- run the "real robot" interface nodes -->
  <!--   - this typically includes: robot_state, motion_interface, and joint_trajectory_action nodes
  ↪  -->
  <!--   - replace these calls with appropriate robot-specific calls or launch files -->
  <group unless="$(arg sim)">
    <include file="$(find [robot_interface_pkg])/launch/robot_interface.launch" >
      <arg name="robot_ip" value="$(arg robot_ip)"/>
    </include>
  </group>

  <!-- publish the robot state (tf transforms) -->
  <node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher" />

  <include file="$(find [robot_moveit_config])/launch/move_group.launch">
    <arg name="publish_monitored_planning_scene" value="true" />
  </include>

  <include file="$(find [robot_moveit_config])/launch/moveit_rviz.launch">
    <arg name="config" value="true"/>
  </include>

  <include file="$(find [robot_moveit_config])/launch/default_warehouse_db.launch" />

</launch>
```

2.3 tx260l_macro.xacro

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://ros.org/wiki/xacro">
   <xacro:include filename="$(find staubli_resources)/urdf/common_materials.xacro" />

   <xacro:macro name="staubli_tx260l" params="prefix">
      <!-- links -->
      <link name="${prefix}base_link">
         <visual>
            <origin xyz="0 0 0" rpy="0 0 0" />
            <geometry>
               <mesh filename="package://staubli_tx260l_support/meshes/tx260l/visual/base_link.stl"
               ↪  />
            </geometry>
            <xacro:material_staubli_ral_melon_yellow />
         </visual>
         <collision>
            <origin xyz="0 0 0" rpy="0 0 0" />
            <geometry>
               <mesh
               ↪  filename="package://staubli_tx260l_support/meshes/tx260l/collision/base_link.stl"
               ↪  />
            </geometry>
         </collision>
         <inertial>
            <mass value="5.76415" />
            <origin xyz="-0.010284 -0.000676 0.087340" rpy="0.0 0.0 0.0" />
            <inertia ixx="0.000025" ixy="-0.000001" ixz="-0.000002" iyy="0.000034" iyz="-0.0000001"
            ↪  izz="0.000029" />
         </inertial>
      </link>
      <link name="${prefix}link_1">
         <visual>
            <origin xyz="0 0 0" rpy="0 0 0" />
            <geometry>
               <mesh filename="package://staubli_tx260l_support/meshes/tx260l/visual/link_1.stl" />
            </geometry>
            <xacro:material_staubli_ral_melon_yellow />
         </visual>
         <collision>
            <origin xyz="0 0 0" rpy="0 0 0" />
            <geometry>
               <mesh filename="package://staubli_tx260l_support/meshes/tx260l/collision/link_1.stl"
               ↪  />
            </geometry>
         </collision>
         <inertial>
            <mass value="7.769025" />
            <origin xyz="-0.000234 0.023187 0.322578" rpy="0.0 0.0 0.0" />
            <inertia ixx="0.000059" ixy="-0.0000001" ixz="-0.0000005839" iyy="0.0001436040"
            ↪  iyz="0.0000040196" izz="0.0000118231" />
         </inertial>
      </link>
      <link name="${prefix}link_2">
         <visual>
            <origin xyz="0 0 0" rpy="0 0 0" />
            <geometry>
               <mesh filename="package://staubli_tx260l_support/meshes/tx260l/visual/link_2.stl" />
            </geometry>
            <xacro:material_staubli_ral_melon_yellow />
         </visual>
         <collision>
```

```
      <origin xyz="0 0 0" rpy="0 0 0" />
      <geometry>
        <mesh filename="package://staubli_tx260l_support/meshes/tx260l/collision/link_2.stl"
        ↪  />
      </geometry>
    </collision>
    <inertial>
      <mass value="14.350897" />
      <origin xyz="0.002379421 0.041907421 0.170583237" rpy="0.0 0.0 0.0" />
      <inertia ixx="0.000137" ixy="0.000000" ixz="-0.000001" iyy="0.000144" iyz="0.000004"
      ↪  izz="0.000012" />
    </inertial>
  </link>
  <link name="${prefix}link_3">
    <visual>
      <origin xyz="0 0 0" rpy="0 0 0" />
      <geometry>
        <mesh filename="package://staubli_tx260l_support/meshes/tx260l/visual/link_3.stl" />
      </geometry>
      <xacro:material_staubli_ral_melon_yellow />
    </visual>
    <collision>
      <origin xyz="0 0 0" rpy="0 0 0" />
      <geometry>
        <mesh filename="package://staubli_tx260l_support/meshes/tx260l/collision/link_3.stl"
        ↪  />
      </geometry>
    </collision>
    <inertial>
      <mass value="5.16" />
      <origin xyz="-0.003889 0.019212 -0.007944" rpy="0.0 0.0 0.0" />
      <inertia ixx="0.000030" ixy="-0.000001" ixz="-0.000000" iyy="0.000022" iyz="0.000001"
      ↪  izz="0.000020" />
    </inertial>
  </link>
  <link name="${prefix}link_4">
    <visual>
      <origin xyz="0 0 0" rpy="0 0 0" />
      <geometry>
        <mesh filename="package://staubli_tx260l_support/meshes/tx260l/visual/link_4.stl" />
      </geometry>
      <xacro:material_staubli_ral_melon_yellow />
    </visual>
    <collision>
      <origin xyz="0 0 0" rpy="0 0 0" />
      <geometry>
        <mesh filename="package://staubli_tx260l_support/meshes/tx260l/collision/link_4.stl"
        ↪  />
      </geometry>
    </collision>
    <inertial>
      <mass value="4.0287377" />
      <origin xyz="-0.001208 -0.000000 0.287605" rpy="0.0 0.0 0.0" />
      <inertia ixx="0.000042" ixy="-0.000000" ixz="0.000001" iyy="0.000041" iyz="-0.000000"
      ↪  izz="0.000007" />
    </inertial>
  </link>
  <link name="${prefix}link_5">
    <visual>
      <origin xyz="0 0 0" rpy="0 0 0" />
      <geometry>
        <mesh filename="package://staubli_tx260l_support/meshes/tx260l/visual/link_5.stl" />
      </geometry>
```

```
      <xacro:material_staubli_ral_grey_aluminium />
   </visual>
   <collision>
      <origin xyz="0 0 0" rpy="0 0 0" />
      <geometry>
         <mesh filename="package://staubli_tx260l_support/meshes/tx260l/collision/link_5.stl"
         ↪  />
      </geometry>
   </collision>
   <inertial>
      <mass value="0.123625" />
      <origin xyz="0.000190 0.000001 0.015244" rpy="0.0 0.0 0.0" />
      <inertia ixx="0.000000" ixy="0.000000" ixz="0.000000" iyy="0.000000" iyz="0.000000"
      ↪  izz="0.000000" />
   </inertial>
</link>
<link name="${prefix}link_6">
   <visual>
      <origin xyz="0 0 0" rpy="0 0 0" />
      <geometry>
         <mesh filename="package://staubli_tx260l_support/meshes/tx260l/visual/link_6.stl" />
      </geometry>
      <xacro:material_staubli_ral_grey_aluminium />
   </visual>
   <collision>
      <origin xyz="0 0 0" rpy="0 0 0" />
      <geometry>
         <mesh filename="package://staubli_tx260l_support/meshes/tx260l/collision/link_6.stl"
         ↪  />
      </geometry>
   </collision>
   <inertial>
      <mass value="0.123625" />
      <origin xyz="-0.000195 -0.000000 -0.006636" rpy="0.0 0.0 0.0" />
      <inertia ixx="0.000000" ixy="-0.000000" ixz="0.000000" iyy="0.000000" iyz="0.000000"
      ↪  izz="0.000000" />
   </inertial>
</link>
<link name="${prefix}tool0" />
<!-- joints -->
<joint name="${prefix}joint_1" type="revolute">
   <origin xyz="0 0 0" rpy="0 0 0" />
   <parent link="${prefix}base_link" />
   <child link="${prefix}link_1" />
   <axis xyz="0 0 1" />
   <!-- from Staubli instruction manual:
(1) effort limit = 318 Nm if floor mounted
(2) effort limit = 111 Nm if wall mounted -->
   <limit lower="-3.14159265" upper="3.14159265" effort="318.0" velocity="7.592" />
   <dynamics damping="0.0" friction="0.0" />
</joint>
<joint name="${prefix}joint_2" type="revolute">
   <origin xyz="0.0 0.130 0.375" rpy="0 0 0" />
   <parent link="${prefix}link_1" />
   <child link="${prefix}link_2" />
   <axis xyz="0 1 0" />
   <limit lower="-2.22529" upper="2.22529" effort="130.0" velocity="6.719" />
   <dynamics damping="0.0" friction="0.0" />
</joint>
<joint name="${prefix}joint_3" type="revolute">
   <origin xyz="0 -0.110 0.400" rpy="0 0 0" />
   <parent link="${prefix}link_2" />
   <child link="${prefix}link_3" />
```

```
      <axis xyz="0 1 0" />
      <limit lower="-2.48709" upper="2.48709" effort="65.0" velocity="8.726" />
      <dynamics damping="0.0" friction="0.0" />
    </joint>
    <joint name="${prefix}joint_4" type="revolute">
      <origin xyz="0 0 0" rpy="0 0 0" />
      <parent link="${prefix}link_3" />
      <child link="${prefix}link_4" />
      <axis xyz="0 0 1" />
      <limit lower="-4.712389" upper="4.712389" effort="34.0" velocity="17.366" />
      <dynamics damping="0.0" friction="0.0" />
    </joint>
    <joint name="${prefix}joint_5" type="revolute">
      <origin xyz="0 0 0.450" rpy="0 0 0" />
      <parent link="${prefix}link_4" />
      <child link="${prefix}link_5" />
      <axis xyz="0 1 0" />
      <limit lower="-2.138028" upper="2.3125612" effort="29.0" velocity="18.587" />
      <dynamics damping="0.0" friction="0.0" />
    </joint>
    <joint name="${prefix}joint_6" type="revolute">
      <origin xyz="0 0 0.070" rpy="0 0 0" />
      <parent link="${prefix}link_5" />
      <child link="${prefix}link_6" />
      <axis xyz="0 0 1" />
      <limit lower="-4.712389" upper="4.712389" effort="11.0" velocity="25.220" />
      <dynamics damping="0.0" friction="0.0" />
    </joint>
    <joint name="${prefix}joint_6-tool0" type="fixed">
      <parent link="${prefix}link_6" />
      <child link="${prefix}tool0" />
      <origin xyz="0 0 0" rpy="0 0 0" />
    </joint>
  </xacro:macro>
</robot>
```