Kurzfassung

Ein mechatronisches System, wie ein Roboter, ist ein komplexes Zusammenspiel von Teilsystemen, die aus den verschiedenen Disziplinen Mechanik, Elektronik und Informatik stammen.

Dieser Umstand macht es schwierig, die einzelnen Teilsysteme zu entwickeln und testen. Um die Entwicklung zu unterstützen und beschleunigen, werden Simulations- und Visualisierungs-Werkzeuge eingesetzt. In dieser Arbeit sollen die Möglichkeiten vom *ROS*-Ökosystem als unterstützendes Werkzeug für *EEROS*-Applikationen evaluiert werden. Im speziellen sollen Lösungen und Vorlagen für die Entwickler von *EEROS*-Applikationen erarbeitet werden.

Dafür werden zwei Fallbeispiele umgesetzt. Die Umsetzung besteht zum einen aus einer Simulation und zum anderen aus einer Visualisierungs-Lösung. Mit der Visualisierung sollen Daten und Zustände gleichermassen vom realen Roboter oder vom simulierten Roboter dargestellt werden.

Das erste Fallbeispiel ist eine einfache Motor-Apparatur. Mit diesem Beispiel konnte erfolgreich aufgezeigt werden, wie die Entwicklung eines Reglers in *EEROS* unterstützt werden kann. Ebenfalls wurde eine Entwicklungsumgebung mit Simulation und Visualisierung für den *EEDURO-Delta* Roboter erstellt. Diese kann dann eingesetzt werden, bei der Entwicklung der *EEROS*-Applikation für den Delta-Roboter. Auch kann mit der Entwicklungsumgebung vom *EEDURO-Delta* Roboter das Framework *EEROS* kennen gelernt werden, ohne zuerst Hardware anzuschaffen.

Mit den beiden in dieser Arbeit gezeigten Fallbeispielen erhalten *ROS*- und *EEROS*-Entwickler eine Vorlage für das Erstellen von Simulationen und Visualisierungen von Robotern.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Bei der Entwicklung von *EEROS*-Applikationen muss immer eine Hardware vorhanden sein für die die Applikation entwickelt wird. Dies ist vor allem beim Beginn der Entwicklung ein erschwerender Umstand, da bei Fehlern die Hardware beschädigt werden kann. Auch gibt es keine einfache und praktische Möglichkeit die Grössen und Zustände von einem System ansprechend zu visualisieren.

1.2 Aufgabenstellung

Das Ziel dieser Arbeit ist zu evaluieren, wie *ROS* als unterstützendes Werkzeug für *EEROS*-Applikationen eingesetzt werden kann. Dabei soll vor allem ein Werkzeug für Simulationen und eines für Visualisierungen gefunden werden.

Dafür soll zuerst ein Konzept ausgearbeitet werden, in dem unter anderem geklärt wird, wie die Schnittstellen zwischen *ROS* und *EEROS* aussehen. Das Konzept soll anschliessend für zwei Fallbeispiele umgesetzt werden. Eine einfache Motor-Apparatur dient als erstes Fallbeispiel. Anhand dieses Beispiels wird die Korrektheit des Konzeptes überprüft. Denn es soll eine *EEROS*-Applikation mithilfe der im Fallbeispiel erstellten Simulation entwickelt und getestet werden. Das zweite Fallbeispiel ist der EEDURO-Deltaroboter. Mit einem Deltaroboter soll gezeigt werden, dass das Konzept auch mit komplizierteren Robotern funktioniert.

1.3 Begriffe

In diesem Abschnitt werden in Kürze die Programme und Begriffe vorgestellt, die in dieser Arbeit verwendet werden.

1.3.1 ROS

ROS (Robot Operating System) ist ein Software-Framework für die Programmierung von Roboteranwendungen. Mit *ROS* werden vor allem übergeordnete Aufgaben in einer Roboteranwendung umgesetzt. Es besteht aus einem Set von Softwarebibliotheken und Werkzeugen. Die einzelnen Teile von *ROS* sind organisiert als Packages. Das *ROS*-Netzwerk besteht aus Knoten die über Peer-to-Peer miteinander kommunizieren.

Für das Verständnis dieser Arbeit ist ein Grundwissen über *ROS* essentiell. Im Abschnitt 1.3.1.1 werden die Begriffe aus *ROS* nochmals kurz aufgefrischt. Darum wird *ROS*-Neulingen empfohlen, sich einen Überblick über *ROS* zu verschaffen. Gute Quellen für dies sind:

- Core Componets[1]
- ROS Wiki[2]
- Arbeit "*MME Simulationsprojekt: ROS und Gazebo*" [3]

[1] http://www.ros.org/core-components/
[2] http://wiki.ros.org/
[3] siehe Anhang xx

1.3.1.1 ROS-Komponenten

In diesem Kapitel werden die wichtigsten *ROS*-Komponenten aufgelistet. Es werden nur die Begriffe der Komponenten kurz vorgestellt. Die Begriffe werden bewusst nicht ins Deutsche übersetzt damit die Begriffe geläufig sind, wenn man die offizielle *ROS*-Dokumentation konsultiert.

Master
Der Master ist die Kernkomponente vom *ROS*-Netzwerk. Er verwaltet die Kommunikation zwischen den Knoten (Nodes). Das Kommunikationsverfahren das verwendet wird, ist das Publish & Subscribe[4] Prinzip.

Nodes
Knoten sind Prozesse in denen Berechnungen oder Treiber ausgeführt werden. Somit kann ein Knoten eine Datenverarbeitung, ein Aktor oder ein Sensor darstellen.

Messages
Die Messages sind eine Konvention für den Austausch von Daten. Darum gibt es für die unterschiedlichen Anforderungen der Kommunikation, verschiedene Nachrichten-Typen.

Topics
Der Nachrichtenaustausch mit dem Publish & Subscribe Mechanismus ist anonym. Darum werden die Nachrichten unter einem Thema (Topic) ausgetauscht.

1.3.2 EEROS

EEROS[5] ist ein Roboter-Framework mit Fokus auf die Ausführung von Echtzeit-Aufgaben. Somit ist es geeignet für die Umsetzung von untergeordneten Aufgaben in einer Roboteranwendung, wie das Regeln von einem Motor.

1.4 Zusammenarbeit

An der Ausarbeitung des Konzeptes hat Marcel Gehrig aktiv mitgewirkt. Denn die *ROS*-Schnittstelle im *EEROS*, die er im Rahmen seiner Arbeit *"Eine ROS Anbindung für EEROS"*[6] entwickelt hat, ist eine integraler Bestandteil des Konzepts. Auch hat er die *EEROS*-Applikation für das Fallbeispiel Motor-Apparatur entwickelt und somit die Umsetzung des Fallbeispiels validiert.

[4] https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern
[5] http://eeros.org
[6] siehe Anhang xx

2 Konzept

Eine Skizze des Konzeptes ist in der Abbildung 2.1 zu sehen. Die zwei Hauptkomponenten sind *EEROS* und *ROS*. *EEROS* übernimmt in diesem Szenario die Aufgabe des Reglers. Und *ROS* die Aufgaben der Simulation und der Visualisierung.

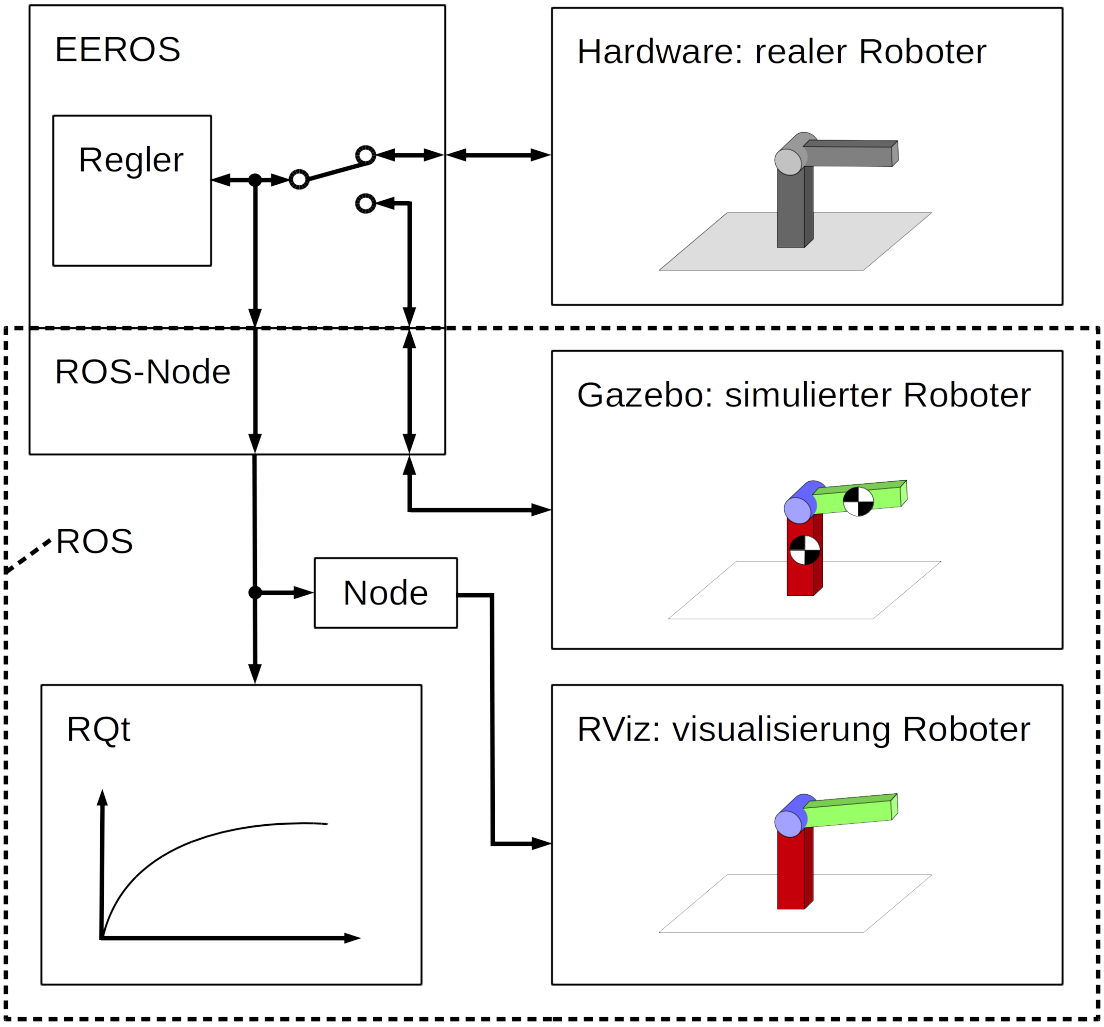

Abbildung 2.1: Konzept

Damit *EEROS* mit *ROS*-Komponenten kommunizieren kann, wurde ein *ROS*-Node ins *EEROS* integriert. Diese Integration wird in der Arbeit xx erklärt. Somit hat *EEROS* die Fähigkeit über das *ROS*-Kommunikationsprinzip *Publish & Subscribe* Daten auszutauschen. Das Konzept sieht vor, das *EEROS* wahlweise einen echten oder simulierten Roboter steuert und regelt. Der Roboter in der Skizze ist aus zwei Gliedern und einem Gelenk aufgebaut. Die Stellgrösse die von *EEROS* vorgegeben wird ist das Drehmoment für das Gelenk. Und die Regelgrösse ist die Winkelposition des Gelenks.

Die Visualisierung mit *RQt* und *RViz* erfolgt gleichermassen, ob ein echter oder simulierter Roboter betrieben wird. Mit *RQt* können die Grössen in Plots dargestellt werden, und der Zustand des Roboters kann mit *RViz* visualisiert werden. Da *RViz* nicht direkt die Gelenkwinkel interpretieren kann, müssen diese von einem *ROS*-Knoten[1] in Koordinatensystem-Transformationen[2] umgerechnet werden.

[1] siehe Kapitel 2.5.1
[2] siehe Kapitel 2.5

2.1 Gazebo

Gazebo ist eine Starrkörper-Simulationsumgebung für Mehrkörper-Systeme. Vom System das man simulieren möchte, muss eine System-Beschreibung in Form von einer *SDF*-Datei erstellt werden. *Gazebo* baut dann die Simulation anhand dieser Datei auf. In unserem Fall wird im *SDF*[3] das Roboter-Modell beschrieben.

Mit Plugins kann *Gazebo* mit Funktionen erweitert werden. Dabei kann man schon fertige Plugins verwenden, oder selber eines programmieren. Im Zusammenhang mit diesem Konzept, werden die Plugins für die Kommunikation mit dem *EEROS* eingesetzt. Ein Plugin wird für das Applizieren der Stellgrösse (Drehmoment) auf das Gelenk eingesetzt und eines für die Messung und Rückführung (publish) des Gelenkwinkels.

2.2 RViz

Mit dem Visualisierungs-Tool *RViz* kann der Zustand vom Roboter visualisiert werden. Der Zustand ist die momentane Stellung von den Gelenken des Roboters, oder einfacher ausgedrückt die Pose des Roboters. Auch wird *RViz* eingesetzt für die Visualisierung von anderen Daten wie z.B. die Sensor-Daten einer 3D-Kamera.

Damit RViz den Zustand vom Roboter darstellen kann, braucht es folgende Informationen:

- Aussehen jedes Glieds des Roboters, wie Form und Farbe
- laufend für jedes Glied: Position und Orientierung

Mit einer *URDF*-Datei[4] müssen dem *RViz* die Information über das Aussehen zur Verfügung gestellt werden. *URDF* ist ein Format für die Modell-Beschreibung eines Roboters. In *RViz* muss immer ein Koordinatensystem als fixe Referenz angegeben werden. Normalerweise wird das Welt-Koordinatensystem gewählt. Um ein Körper im Raum darzustellen, braucht *RViz* eine Angabe zur Position und Orientierung, relative zum Referenz-Koordinatensystem. Diese Angabe wird als Transformation[5] bezeichnet. *RViz* erhält laufend diese Transformations-Daten von einem *ROS*-Knoten und kann anhand von diesen, den aktuellen Zustand des Roboters visualisieren.

2.3 RQt

RQt ist Framework für die GUI Entwicklung in *ROS*. Diese GUI's werden als Plugins implementiert. Somit können mehre GUI's in einem *RQt*-Fenster verwendet werden. Für fast jedes gängige *ROS*-Command-line-Tool gibt es schon Plugins[6]. Es können aber auch selbst programmierte Plugins verwendet werden. Für die Visualisierung von zeitabhängigen Grössen wird das *RQt*-Plugin *multiplot*[7] eingesetzt.

[3] siehe Kapitel 2.4.2
[4] siehe Kapitel 2.4.1
[5] siehe Kapitel 2.5
[6] http://wiki.ros.org/rqt/Plugins
[7] http://wiki.ros.org/rqt_multiplot

2.4 Roboter-Modell

Die Programme *Gazebo* und *RViz* verwenden zwei unterschiedliche Datei-Formate für die Modell-Beschreibung. Beide sind im XML-Stil gehalten und repräsentieren kinematische Strukturen. Diese Strukturen sind aus Gliedern (Links) und Gelenken (Joints) aufgebaut.

2.4.1 URDF-Unified Robot Description Format

Das *URDF*-Format[8] wird standardmässig in *ROS* verwendet für die Beschreibung von Robotern. Es kann nur kinematische Strukturen abbilden die eine Baum-Form haben. Somit können keine geschlossenen kinematischen Ketten beschrieben werden. Im Kapitel 2.4.1 wird beschrieben, wie ein *URDF*-Model aufgebaut werden kann.

2.4.2 SDF-Simulation Description Format

Das *SDF*-Format wird bis jetzt ausschliesslich im *Gazebo* verwendet. Es kann kinematische Graph-Strukturen abbilden. Deshalb können auch geschlossene kinematische Ketten beschrieben werden.

2.4.3 Konvertierung URDF zu SDF

Damit für ein Roboter nicht zwei Dateien erstellt und instand gehalten werden müssen gibt es eine Lösung. Die Lösung besteht darin die *URDF*-Datei in eine *SDF*-Datei zu konvertieren. Die Konvertierung erfolgt im *ROS*-Node *"urdf_spawner"* [9] zur Laufzeit, wenn das Roboter-Modell in die Simulations-Umgebung von *Gazebo* geladen wird. Das *URDF*-Format ist jedoch nicht so mächtig wie das *SDF*-Format. Um jedoch den ganzen Umfang der Möglichkeiten vom *SDF*-Format zu nutzen kann die *URDF*-Datei mit speziellen XML-Elementen erweitert werden. In welchen Fällen es solche Spezial-Elemente braucht und wie sie eingesetzt werden wird im Kapitel 3.2.1 gezeigt.

2.4.4 URDF-Modell

Ein Modell besteht aus den zwei Teilen: Glied (Link) und Gelenk (Joint). Für jedes gib es ein entsprechendes XML-Element. Die Beziehung zwischen den beiden Komponenten ist in der Abbildung 2.2 zu sehen. Wichtig zu beachten ist, dass der Ursprung vom Link (z.B Link 2) zusammenfällt mit dem Ursprung vom vorgelagerten Joint (z.B. Joint 1). Die Abstände zwischen den Gelenken werden nicht vom Link-Element definiert sondern vom Joint-Element (siehe Gelenk-Ursprung in Kapitel 2.4.4.2). Somit wird die Kinematische Struktur des Roboters nur von den Joint-Elementen vorgegeben.

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

[9] https://github.com/ros-simulation/gazebo_ros_demos/blob/kinetic-devel/rrbot_gazebo/launch/rrbot_world.launch

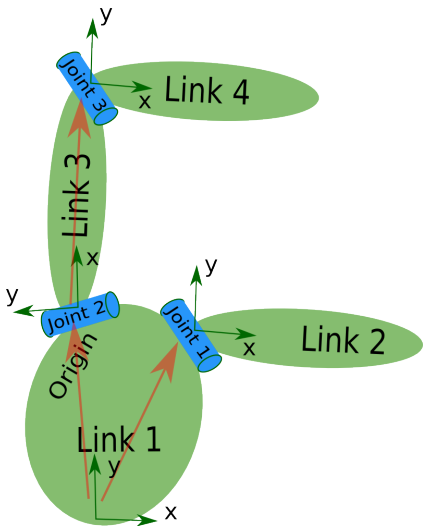

Abbildung 2.2: Aufbau URDF [1]

2.4.4.1 Link

Das Link-Element beschreibt ein einzelnes Glied vom Roboter. Der Ursprung vom Link ist der gleiche, wie der Ursprung vom Joint. Folgende Informationen sind als Sub-Elemente enthalten:

- Massenträgheit
- Geometrie und Material für Aussehen
- Geometrie für Kollisionsberechnung

Für jedes dieser aufgelisteten Sub-Element muss dessen Ursprung bezüglich des Link-Ursprungs angegeben werden (dargestellt durch drei schwarze und gekrümmte Pfeile in Abbildung 2.3).

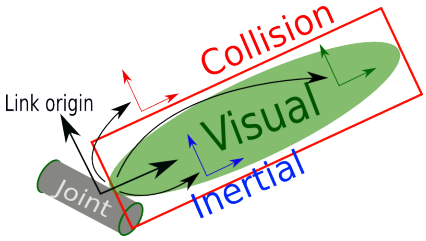

Abbildung 2.3: Schematische Darstellung Link-Element [1]

2.4.4.2 Joint

Mit dem Joint-Element können Gelenke unterschiedlichster Art beschrieben werden. Folgende Informationen braucht es zur Beschreibung eines Gelenks:

- Gelenk-Type
- Gelenk-Ursprung (als Transformation gegenüber Ursprung vom Eltern Link)
- Eltern Link-Element
- Kind Link-Element

Die zwei häufigsten Gelenk-Typ die verwendet werden sind:

- "kontinuierliche" Typ, freie Rotation um eine Achse (continuous)
- "fixe" Type, beide Links fest verbunden (fixed)

Die Beziehung dieser Angaben zueinander ist in Abbildung 2.4 dargestellt.

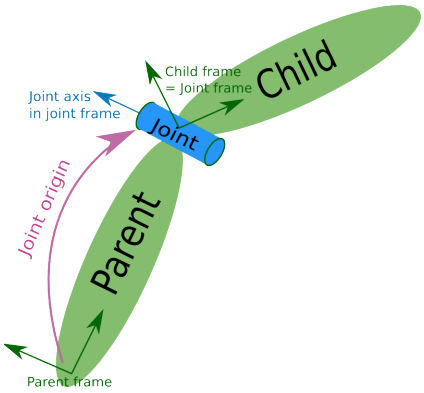

Abbildung 2.4: Schematische Darstellung Joint-Element [1]

2.5 Transformationen

Eine Transformation beschreibt, wie ein Koordinatensystem relativ zu einem anderen steht. Koordinatensysteme beschreiben eine Position und Orientierung im Raum, für zum Beispiel ein Glied. Aufbau einer Transformation:

- Bezeichnung von Eltern-Koordinatensystem bzw. Eltern-Glied
- Bezeichnung von Kind-Koordinatensystem bzw. Kind-Glied
- Translations-Komponente
- Rotations-Komponente

Durch die Eltern/Kind Relation bilden die Transformationen eine Baumstruktur. Das Wurzel-Element dieses Baumes ist meistens das Welt-Koordinatensystem.

Zur Veranschaulichung dieser Sachverhalte wird ein Beispiel eines einfachen Roboters mit zwei Gliedern und einem Gelenk gezeigt (zusehen in Abbildung 2.5). Die Transformations-Baumstruktur (TF-Tree) zu diesem einfachen Roboter ist in Abbildung 2.6 dargestellt.

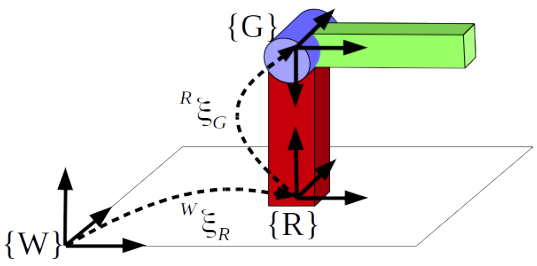

Abbildung 2.5: Beispiel: Transformation

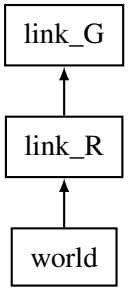

Abbildung 2.6: Beispiel: TF-Tree

Man berechnet das Koordinatensystem des grünen Gliedes in Bezug zu der Welt wie folgt:

$$^{W}\xi_{G} = ^{W}\xi_{R} * ^{R}\xi_{G} \tag{2.1}$$

Die Angabe für das Koordinatensystem von Objekt {B}, in Bezug zu einem Koordinatensystem {A}, ist gleichzeitig auch die Transformation (${}^{A}\xi_{B}$) von Koordinatensystem {A} zu {B}.

2.5.1 Umrechnung Gelenkwinkel zu Transformationen

Für die Umrechnung von den Winkelpositionen der Gelenke zu den Koordinaten-Transformationen gibt es in *ROS* den *robot_state_publisher*. Dieser *ROS*-Node benötigt die Beschreibung des Roboter-Modells als *URDF*-Datei. Er setzt dann die Gelenkwinkel im Modell ein und rechnet anschliessend die einzelnen Transformationen zwischen den Gelenken aus. Mit diesen Transformationen kann dann *RViz* die Glieder im Raum so positionieren, dass es den Roboter korrekt darstellen kann.

3 Motor-Apparatur

In diesem Kapitel wird anhand eines einfachen Fallbeispiels gezeigt, wie mit *ROS* die Entwicklung einer *EEROS* Applikation unterstützt werden kann. Das Fallbeispiel ist eine Motor-Apparatur (Abbildung 3.1).

Für dieses System muss im *EEROS* ein Regler entwickelt werden. Um die Entwicklung zu unterstützen und den Regler zu testen, wird eine Simulation erstellt. Mit Plots der Stell- und Regel-Grössen kann die Leistung des Reglers besser beurteilt werden.

Durch das einfache Beispiel können alle Grundlagen vorgestellt werden, die es braucht um eine Simulation und Visualisierung erstellen zu können.

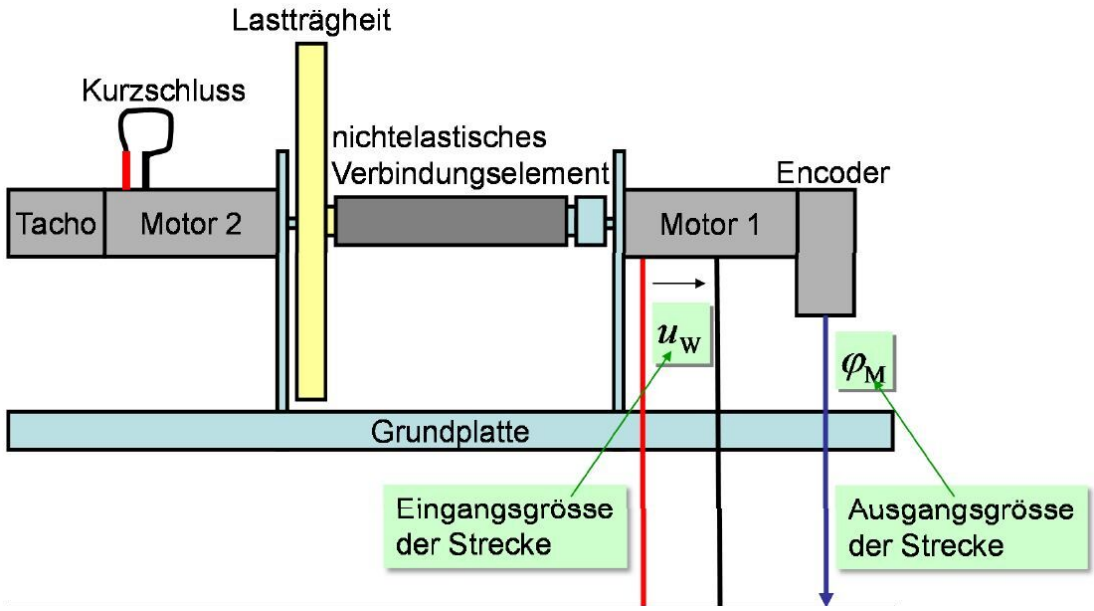

Abbildung 3.1: schematische Darstellung von Motor-Apparatur

3.1 Motor-Apparatur

Die Baugruppe besteht aus einem Motor, Schwungrad und linearen Dämpfer. Diese drei Komponenten sind miteinander gekoppelt. Der lineare Dämpfer ist realisiert, durch einen zweiten kurzgeschlossenen Motor.

Die Parameter für das Simulations-Model konnten grösstenteils aus den Datenblättern der jeweiligen Komponenten entnommen werden. Die Dämpfungskonstante des Dämpfers und die Rotationsträgheit der Schwungscheibe mussten berechnet werden. Die Datenblätter und Berechnungen sind im Repository "*motor_sim*" [1] zu finden.

Für Parameter die nicht im Datenblatt stehen, oder nicht genügend Informationen für eine Berechnung vorhanden waren, wurden geschätzt. Jedoch haben die geschätzten Parameter in diesem Fallbeispiel keinen nennenswerten Einfluss auf die Simulation.

[1] https://github.com/manuelilg/motor_sim/tree/master/motor_description/datasheets

3.2 Modell

Für die Apparatur wurde eine Modell-Beschreibung im *URDF*-Format erstellt. Die Kinematische Struktur des Modells ist in der Abbildung 3.2 dargestellt. Auffallend ist, dass der Dämpfer noch nicht mit dem Schwungrad gekoppelt ist. Warum dies so ist, wird im Kapitel 3.2.1 erläutert.

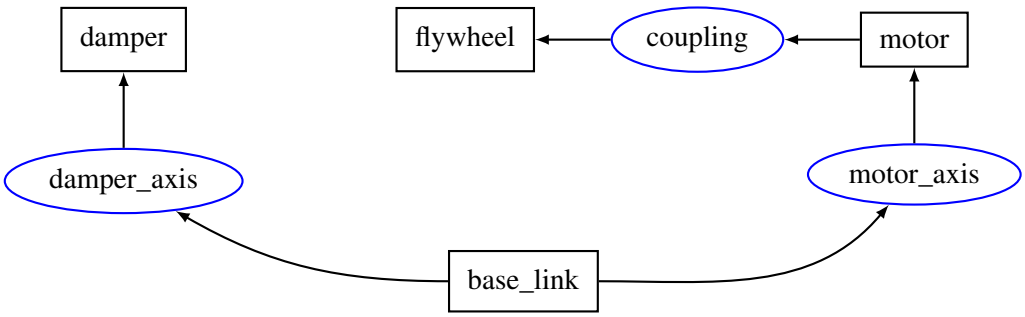

Abbildung 3.2: kinematische Struktur Motor-Apparatur

Die komplette Datei ist im Repositiory "*motor_sim*" [2] abgelegt. Ein Ausschnitt aus dieser Datei ist in Auflistung 3.1 gezeigt.

```
1  <?xml version="1.0"?>
2  <robot name="motor">
3
4    <link name="world"/>
5
6    <joint name="fixe_base" type="fixed">
7      <parent link="world"/>
8      <child link="base_link"/>
9      <origin xyz="0 0 0.005" rpy="0 0 0"/>
10   </joint>
11
12   <link name="base_link">
13     <visual>
14       <geometry>
15         <box size="0.2 0.1 0.01"/>
16         <origin xyz="0 0 0.01" rpy="0 0 0"/>
17       </geometry>
18     </visual>
19     <collision>
20       <geometry>
21         <box size="0.1 0.1 0.01"/>
22         <origin xyz="0 0 0.01" rpy="0 0 0"/>
23       </geometry>
24     </collision>
25     <inertial>
26       <mass value="1"/>
27       <inertia ixx="1" ixy="0.0" ixz="0.0" iyy="1" iyz="0.0" izz="1"/>
28     </inertial>
29   </link>
30 ...
```

Auflistung 3.1: Ausschnitt aus motor.urdf

[2] https://github.com/manuelilg/motor_sim/tree/master/motor_description/urdf

3.2.1 Kinematische Kette schliessen

Das *URDF*-Format kann keine geschlossenen kinematischen Strukturen abbilden. Dieser Mangel kann aber behoben werden mit: einem *SDF*-Joint Eintrag im *URDF* und mit dem *URDF*-Spezial-Element "Mimic".

3.2.1.1 SDF-Joint

Die in diesem Abschnitt erklärte Anpassung wird benötigt, damit in der Simulation das Schwungrad und der Dämpfer miteinander gekoppelt sind.

Im *URDF* können Informationen hinterlegt werden, die nur *Gazebo* interpretiert. Dafür müssen die Informationen mit dem XML-Element "gazebo" umschlossen werden.

Wenn man jetzt ein Joint-Element im *SDF*-Format einsetzt, kann man die kinematische Struktur schliessen. Denn die URDF-Datei wird, bevor es ins *Gazebo* geladen wird, ins *SDF*-Format konvertiert. Während der Konvertierung werden die Zeilen die sich im XML-Element "gazebo" befinden einfach ins *SDF* übernommen.

Wie der Eintrag für die Motor-Baugruppe lauten muss, ist in Auflistung 3.2 gezeigt.

```
1   <gazebo>
2     <joint name="coupling_2" type="fixed">
3       <parent>damper</parent>
4       <child>flywheel</child>
5     </joint>
6   </gazebo>
```

Auflistung 3.2: SDF-Joint in URDF

3.2.1.2 Mimic

Der *ROS*-Knoten *robot_state_publisher* berechnet die Transformations-Daten für die Darstellung der Körper im *RViz*. Dafür braucht er von allen Gelenken im Modell die Gelenkwinkel. Das Modell vom *URDF* hat zwei Gelenke, aber das in der Simulation nur eines. Da in der Simulation die beiden Gelenke fix verbunden sind (siehe Kapitel 3.2.1.1).

Deshalb muss ein Mimic-Element in einem der beiden Gelenk-Definitionen eingefügt werden (siehe Auflistung 3.3).

```
1  <joint name="damper_axis" type="continuous">
2    <parent link="base_link"/>
3    <child link="damper"/>
4    <axis xyz="1 0 0"/>
5    <origin xyz="-0.07 0 0.1" rpy="0 0 0"/>
6    <dynamics damping="2.1087e-4"/>
7    <mimic joint="motor_axis" multiplier="1" offset="0"/>
8  </joint>
```

Auflistung 3.3: Mimic in URDF-Joint

Dieser Eintrag wird vom *ROS*-Knoten *joint_state_publisher*[3] interpretiert. Er bewirkt, dass das Gelenk *"damper_axis"* den gleichen Winkel hat, wie das Gelenk *"motor_axis"*. Somit erhalten wir vom *joint_state_publisher* für die Eingabe von einem Gelenkwinkel *"motor_axis"*, die Ausgabe für die Gelenkwinkel *"motor_axis"* und *"damper_axis"*.

[3] http://wiki.ros.org/joint_state_publisher

Diese beiden werden dann vom *robot_state_publisher* als Eingabe benötigt. In Abbildung 4.3 wird gezeigt wie der *robot_state_publisher* bei der Visualisierung von einem Delta-Roboter eingesetzt wird.

3.2.2 Gazebo Plugins

Damit das Modell in der Simulation mit dem *ROS*-Netzwerk interagieren kann, müssen Plugins eingesetzt werden.

Mit den in Auflistung 3.4 gezeigten Zeilen werden zwei *Gazebo*-Plugins im Motor-Modell eingebettet.

```
1  <gazebo>
2    <plugin name="gazebo_ros_joint_force" filename="
         libgazebo_ros_joint_force.so">
3      <robotNamespace>/motor_sim</robotNamespace>
4      <topicName>effort</topicName>
5      <jointName>motor_axis</jointName>
6    </plugin>
7  </gazebo>
8
9  <gazebo>
10   <plugin name="gazebo_ros_joint_state_publisher" filename="
         libgazebo_ros_joint_state_publisher.so">
11     <robotNamespace>/motor_sim</robotNamespace>
12     <jointName>motor_axis</jointName>
13     <updateRate>10000.0</updateRate> <!-- to get sure that one every
           sim step called, up to 10k -->
14   </plugin>
15 </gazebo>
```

Auflistung 3.4: Plugin in URDF

Das Plugin *"gazebo_ros_joint_state_publisher"* misst den Winkel von Gelenk *"motor_axis"* und stellt den Messwert zu Verfügung (publish).
Es ist im Standard *ROS-Package "gazebo_ros_pkgs"* [4] enthalten. Achtung das Gazebo-Plugin *"gazebo_ros_joint_state_publisher"* ist nicht mit dem ROS-Knoten *"joint_state_publisher"* zu verwechseln.

Das zweite Plugin *"gazebo_ros_joint_force"* wird im Abschnitt 3.2.2.1 genauer erläutert.

3.2.2.1 Joint Force Plugin

Dieses Plugin wurde im Rahmen dieser Arbeit erstellt und ist im gleichnamigen Repository *"gazebo_ros_joint_force"* [5] zu finden. Teile von dem Plugin werden vor jedem Simulations-Schritt aufgerufen und erfüllen somit folgende Aufgaben.

- Stellgrösse (Drehmoment) von *EEROS* empfangen
- Drehmoment auf definiertes Gelenk applizieren
- Simulation mit *EEROS* synchronisieren

[4] http://wiki.ros.org/gazebo_ros_pkgs
[5] https://github.com/manuelilg/gazebo_ros_joint_force

3.3 Installation

In diesem Abschnitt wird die Installation vom *ROS*-Bestandteil des Fallbeispiel Mortor-Apparatur erklärt. Die Installation vom *EEROS*-Bestandteil wird in der Arbeit *"Eine ROS Anbindung für EEROS"* von Marcel Gehrig gezeigt.

Alle erforderliche Repositorys in *Workspace* klonen:

```
$ cd ~/catkin_ws/src
$ git clone https://github.com/manuelilg/motor_sim.git
$ git clone https://github.com/manuelilg/gazebo_ros_joint_force.git
$ git clone https://github.com/manuelilg/gazebo_ros_pkgs.git
```

Alle geklonten Packages kompilieren:

- mit *catkin-tools* [6] (empfohlen)

  ```
  $ catkin build
  ```

- mit *catkin_make* [7]

  ```
  $ catkin_make
  ```

Umgebungsvariablen laden für Ausführung:

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

3.4 Ausführen

In diesem Abschnitt wird gezeigt wie die verschiedenen Bestandteile gestartet werden können.

3.4.1 Gazebo

Simulation von Motor-Apparatur starten (Simulation ist pausiert)

```
$ roslaunch motor_gazebo motor_gazebo.launch
```

Anschliessend um die Simulation zu beeinflusse *EEROS* starten.

3.4.2 RViz

RViz für Visualisierung von Modell starten

```
$ roslaunch motor_description motor_rviz.launch
```

vorgängig oder anschliessend *EEROS* starten (zusammen mit Simulation oder Hardware)

[6] http://catkin-tools.readthedocs.io/en/latest/index.html
[7] http://wiki.ros.org/catkin/commands/catkin_make

3.5 RQt

RQt für das Ploten der System-Grössen starten

```
$ roslaunch motor_rqt motor_rqt.launch
```

vorgängig oder anschliessend *EEROS* starten (zusammen mit Simulation oder Hardware)

4 EEDURO-Delta-Roboter

In diesem Kapitel wird das zweite Fallbeispiel mit dem EEDURO-Delta-Roboter (Abbildung 4.1) vorgestellt. Der EEDURO-Delta ist ein kleiner Delta-Roboter, der für Schulungszwecke gedacht ist. Die Modell-Beschreibung des Roboters ist im *URDF*-Format gehalten. Dabei wurde gleich vorgegangen wie für die Motor-Apparatur[1], deshalb werden in diesem Kapitel nur noch neue Aspekte erläutert.

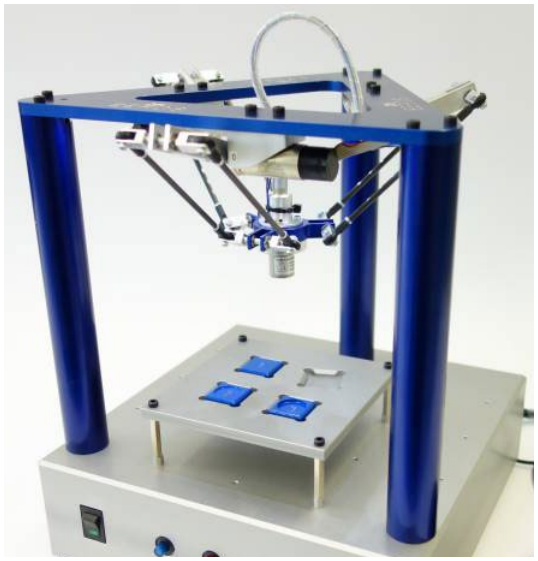

Abbildung 4.1: EEDURO-Delta [2]

4.1 Modell

Die Modell-Beschreibung des Deltaroboters wurde im *URDF*-Format erstellt. Die ganze Datei ist im Repository *"eeduro_delta"* [2] abgelegt. Speziell ist aber das zusätzlich noch die Skriptsprache *XACRO*[3] beim erstellen der *URDF*-Datei eingesetzt wurde.

Das Modell des Delta-Roboter kann grob in drei gleiche Teile aufgeteilt werden:

- Rahmen (base)
- Arm (arm 1-3)
- Werkzeug (tool)

Der Arm selber besteht aus einem Oberarm auch Link 1 genannt und aus einem Unterarm. Der Unterarm selber besteht selber wieder aus:

- Doppelgabel (Link 2)
- zwei Stangen (Link 3.1 und Link 3.2)
- 2.Doppelgabel (Link 4)

Wie beim Motor muss die kinematisch Kette geschlossen werden mit dem *SDF*-Joint Element (dargestellt in Abbildung 4.2 mit grünen Ellipsen).

[1] siehe Kapitel 3
[2] https://github.com/manuelilg/eeduro_delta/tree/master/eeduro_delta_description/urdf
[3] siehe Kapitel 4.1.1.2

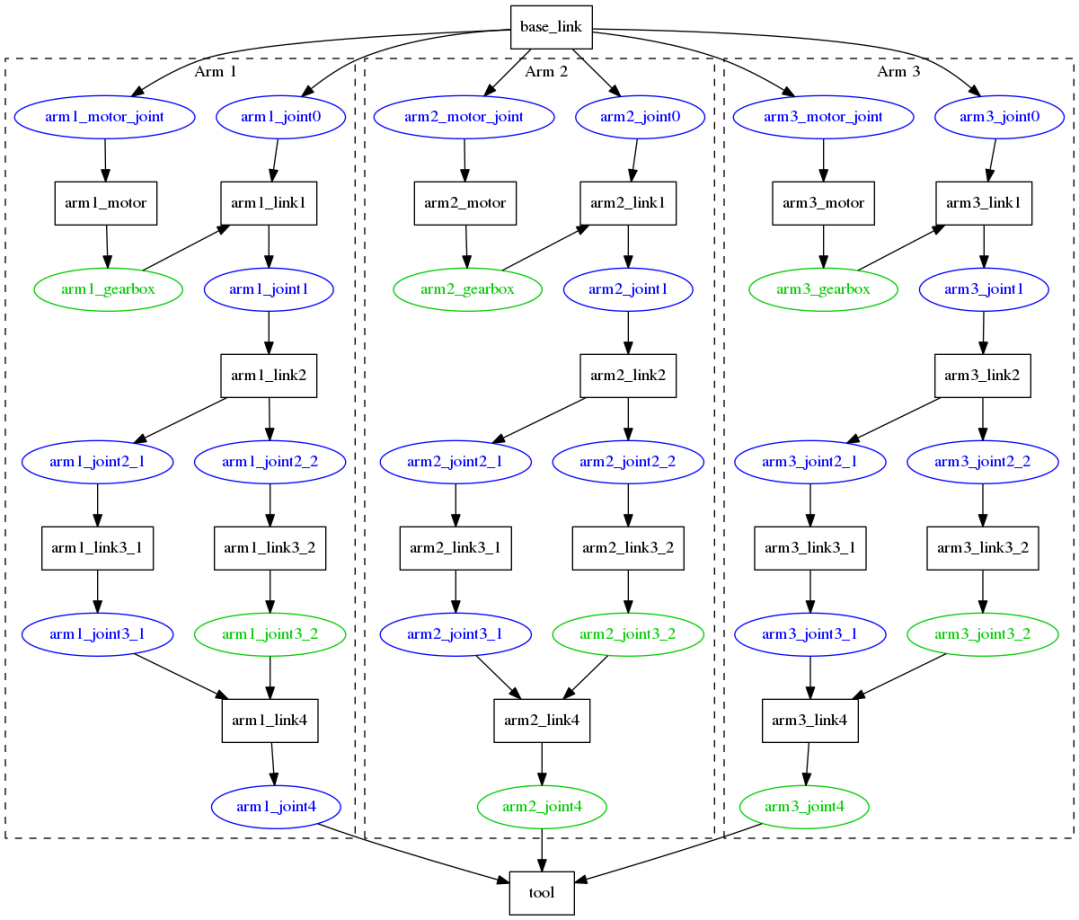

Abbildung 4.2: kinematische Struktur EEDURO-Delta

4.1.1 Parameter für Modell

In diesem Abschnitt wird darauf eingegangen, welche Parameter es braucht für die Erstellung vom Modell.

- Masse und Trägheitstensor von allen Links
- Längenmasse wie Abstand zwischen zwei Gelenken
- Geometrie von allen Links

Ein Teil dieser Parameter konnten aus Datenblättern gewonnen werden. Diese Datenblätter sind im Repository xx zu finden.

4.1.1.1 Berechnung Masse und Trägheitstensor

Für jedes Glied musste die Masse und Rotationsträgheit berechnet werden, damit das Modell des Roboters sich in der Simulation wie der reale Roboter verhält. Die Glieder bestehen jedoch aus mehreren Bauteilen. Dieser Umstand verkompliziert die Berechnung sehr. Auch erschwerend kam hinzu, dass die CAD-Daten nur im *STEP*-Austauschformat vorliegend waren.

Das führt zu folgendem Arbeitsablauf:

- für jedes Glied/Baugruppe eine CAD-Datei erstellen
- für jede CAD-Datei:

  - für jedes Bauteil folgende Daten exportieren
    * Volumen von Körper
    * Schwerpunkt von Körper
    * Rotationsträgheit-Tensor von Volumen
  - exportierte Daten in Matlab eintragen
  - für jedes Bauteil Dichte in Matlab hinterlegen
  - für jedes Bauteil mit Dichte: Masse und Rotationsträgheit ausrechnen
- Schwerpunkt und Masse von Baugruppe bestimmen
- Rotationsträgheit von Baugruppe ausrechnen (Steinerscher Satz für jedes Bauteil anwenden)

Die CAD-Daten und Matlab-Dateien sind im Repository xx zufinden.

4.1.1.2 XACRO

XACRO [4] ist eine Makro Sprache für XML-Dateien. Mit *XACRO* können kürzere und einfacher zu unterhaltende XML-Dateien erstellt werden. Denn mit Hilfe der Makros können Wiederholungen vermieden werden. Auch kann Dank *XACRO* eine XML-Datei in mehrere Unterdateien aufgeteilt werden. Somit kann ein komplexes System logisch in Untersysteme aufgeteilt werden, die dann jeweils in einer eigenen XML-Datei beschrieben werden.

4.2 EEDURO-Delta Joint State Publisher

Der *ROS*-Knoten *"eeduro_delta_joint_state_publisher"* erfüllt folgende Aufgaben:

- Direkt-Kinematik berechnen
- alle Gelenkwinkel ausrechnen

Die Direkt-Kinematik berechnet die Werkzeug-Position aus den drei Gelenkwinkeln von den Oberarmen des Delta-Roboters. Mit Werkzeug-Position können anschliessend die Winkel aller Gelenke des Roboters berechnet werden. Somit müssen für die visualisiert des EEDURO-Delta-Roboters im *RViz* nur die drei Gelenkwinkel von den Oberarmen bekannt sein. Diese Winkel können wahlweise von der Hardware oder von der Simulation kommen.

Eine Ausführliche Dokumentation dieses *ROS*-Knotens kann im Anhang xx gefunden werden.

In Abbildung 4.3 ist der ganze Prozess-Graph mit allen beteiligten Komponenten dargestellt.

[4] http://wiki.ros.org/xacro

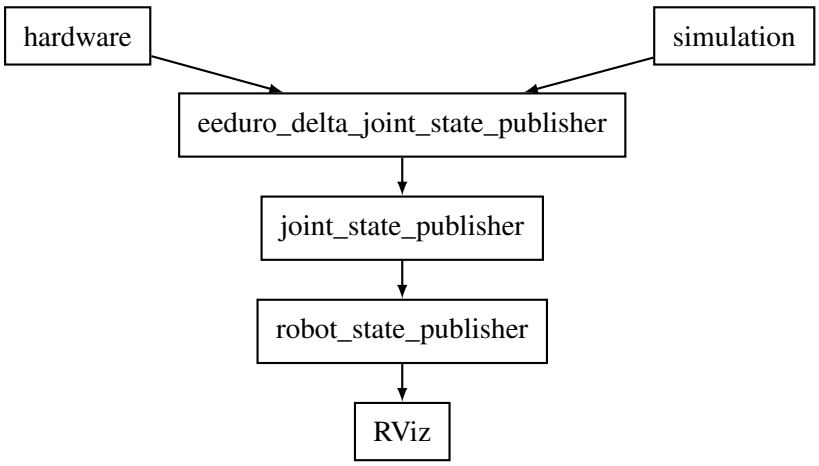

Abbildung 4.3: Prozess-Graph Visualisierung Deltaroboter

4.3 Installation

In diesem Abschnitt wird die Installation des Repository *"eeduro_delta•* gezeigt.

Alle erforderliche Repositorys in *Workspace* klonen:

```
$ cd ~/catkin_ws/src
$ git clone https://github.com/manuelilg/eeduro_delta.git
$ git clone https://github.com/manuelilg/gazebo_ros_joint_force.git
$ git clone https://github.com/manuelilg/gazebo_ros_pkgs.git
```

Alle geklonten Packages kompilieren:

- mit *catkin-tools* [5] (empfohlen)

```
$ catkin build
```

- mit *catkin_make* [6]

```
$ catkin_make
```

Umgebungsvariablen laden für Ausführung:

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

4.4 Ausführen

In diesem Abschnitt wird gezeigt wie die verschiedenen Bestandteile gestartet werden können.

4.4.1 Gazebo

Gazebo-Simulation von EEDURO-Delta starten (Simulation ist pausiert)

```
$ roslaunch eeduro_delta_gazebo eeduro_delta_world.launch
```

[5] http://catkin-tools.readthedocs.io/en/latest/index.html
[6] http://wiki.ros.org/catkin/commands/catkin_make

4.4.2 Gazebo mit RViz

Gazebo zusammen mit *RViz* starten (vgl. Abbildung 4.3)

```
$ roslaunch eeduro_delta_gazebo eeduro_delta_rviz.launch
```

4.5 RQt

RQt für das Ploten der System-Grössen starten

```
$ roslaunch motor_rqt motor_rqt.launch
```

vorgängig oder anschliessend *EEROS* starten (zusammen mit Simulation oder Hardware)

5 Ergebnisse, Fazit und Ausblick

5.1 Ergebnisse

Mit *Gazebo* können Simulationen für *EEROS*-Applikationen erstellt werden. Durch die Synchronisation der Simulation mit dem *EEROS* können Echtzeit-Systeme simuliert werden. Mit dem *RViz* und *RQt* können Daten und Zustände von Robotern visualisiert werden.

Mit dem Fallbeispiel Motor-Apparatur wird gezeigt, dass ein vollständiges Zusammenspiel zwischen *EEROS* und *ROS* umsetzbar ist.

Die Umsetzung des EEDURO-Deltaroboters zeigt die Machbarkeit für komplexere Systeme.

Die Modell-Beschreibungen und Softwarekomponenten die im Rahmen der beiden Fallbeispiel erstellt wurden, können als Vorlage für andere Projekte dienen. Einzelne Softwarekomponenten, wie z.B. das *"gazebo_ros_joint_force"* Plugin, können direkt wiederverwendet werden.

5.2 Fazit

Im Rahmen dieser Arbeit wurde viel Zeit in die Recherche investiert. Besonders die Recherchen für wie Modell-Beschreibung erstellt werden sollen und das wie man *Gazebo* Plugins erstellt.

Die Simulation von einem Delta-Roboter ist die einzig öffentliche.

Auch ist das das ganze Modell des Roboters in einer einzigen Datei beschrieben ist eine

textitEEROS kann von Interessierten dank der Simulation, von der Motor+=Apparatur und dem EEDURO-Delta, ohne Hardwareanforderungen ausprobiert werden. *EEROS*-Applikationen können durch die Synchronisation zwischen *Gazebo* und *EEROS*, auch ohne Realtime-Kernel, entwickelt werden.

5.3 Ausblick

Weitere Arbeiten die im Rahmen dieser Arbeit gemacht werden könnten sind:

Es könnten neue *EEROS*-Applikationen auf Basis der Vorlagen entwickelt werden. Dabei könnten einzelne Software-Komponenten wieder verwendet werden.
Als Beispiel kann der EEDURO-7-Achsen-Roboter umgesetzt werden.

Die Software-Komponenten und Bugfixes könnten als Pull-Requests in die *ROS*-Repositorys gemerged werden.

Eine Möglichkeit das Simulations-Modell des EEDURO-Delta zu erweitern. Zum eine könnte Funktion für das rotieren des Werkzeugs hinzugefügt werden. Und zum andern könnte ein Plugin für das *Gazebo* geschrieben werden, so dass auch die Funktion des Magnets, der das Werkzeug vom Roboter ist, simuliert werden kann.

Quellenverzeichnis

[1] *Web: ROS, https://ros.org/*
Stand vom 27.08.2017

[2] *Web: EEDURO, http://hw.eeros.org/eeduro/*
Stand vom 27.08.2017

A Anhang

- Datenblätter Motor-Baugruppe
- Datenblätter Bauteile von EEDUO-Delta
- Source code