Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Philip Namankiewicz

Entwicklung und Simulation einer Wasserversorgungsstation für mobile Pflanzen

Fakultät Technik und Informatik
Department Fahrzeugtechnik und Flugzeugbau

Faculty of Engineering and Computer Science
Department of Automotive and Aeronautical Engineering

Philip Namankiewicz

Entwicklung und Simulation einer Wasserversorgungsstation für mobile Pflanzen

Bachelorarbeit, eingereicht im Rahmen der Bachelorprüfung

im Studiengang Mechatronik
am Department Fahrzeugtechnik und Flugzeugbau
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. rer. nat. Thomas Lehmann
Zweitgutachter: Prof. Dr.-Ing. Jutta Abulawi

Abgegeben am 04. August 2020

Zusammenfassung

Philip Namankiewicz

Thema der Bachelorarbeit

Entwicklung und Simulation einer Wasserversorgungsstation für mobile Pflanzen

Stichworte

Mechatronik, Sensor, Sensorik, Aktoren, Simulation, Schrittmotor, Raspberry Pi, Software Engineering, Robot Operating System, Gazebo, ROS, Robotik, Linux, Programmieren, Mikrocontroller

Kurzzusammenfassung

Die Bachelorarbeit befasst sich mit der Entwicklung einer Wasserversorgungsstation für mobile Pflanzenroboter. Die Wasserversorgungsstation wird anhand verschiedener Anforderung konstruiert und in eine Simulationswelt integriert. Die einzelnen Schritte der Programmierung einer Simulation werden durchlaufen und anhand von Beispielen erklärt. Die Wasserversorgungsstation soll eine Pflanze bewässern, welche sich auf einer mobilen, autonomen Roboterplattform befindet. Die Simulation dient als realer Ersatz und wird für das Testen der Anforderungen genutzt. Studierende und andere Interessengruppen können einen ersten Einblick in den Aufbau einer Simulationswelt in Gazebo bekommen.

Philip Namankiewicz

Title of the paper

Development and simulation of a water supply station for mobile robots.

Keywords

Mechatronics, sensors, sensors, actuators, simulation, stepper motor, Raspberry Pi, software development, robot operating system, pavilion, ROS, robotics, Linux, programming, microcontrollers

Abstract

The bachelor thesis deals with the development of a water supply station for mobile plant robots. The water supply station is designed and based on various requirements and is integrated into a simulation world. The individual steps of programming a simulation are run through and explained using examples. The water supply station is intended to supply a plant with water, which is located on a mobile, autonomous robot platform. The simulation serves as a real replacement and is used for testing the requirements. Students and other interest groups can get an insight into the structure of a simulation world in Gazebo.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Akronyme

Ibd Internal Block Diagramm

Bdd Block Definition Diagramm

SysML Systems Modeling Language

Ogre Object-Oriented graphics

ROS Robot Operating System

Req Requirement

URDF Unified Robotic Description Format

Glossar

Tag	Ein Tag beschreibt Zusatzinformation für ein Element in der XML-Programmiersprache.
Subscriben	Beim Subscriben wird eine Inforation abonniert und periodisch gespeichert.
Publishen	Beim Publishend wird eine Information periodisch weitergeleitet**.**
Joint	Ein Joint beschreibt ein Gelenk in einer URDF oder Xacro Datei.
Link	Ein Link beschreibt eine einzelne Komponenente, welche in der URDF oder Xacro-Datei definiert wird.
Transmission	Als Transmission wird der Weg von Klasse A zu Klasse B beschrieben.
LiDAR	Ein LiDAR beschreibt einen Sensor, der Hindernisse mit Laserstrahlen detektiert.

1 Einleitung

Viele Autofirmen und Kickstarter konzentrieren sich auf das autonome Fahren mit selbstdenkenden Fahrzeugen. Die meist elektrischen Fahrzeuge sollen sich selbst im Raum zurechtfinden und sich im besten Fall selbstständig mit den benötigten Daten versorgen. Der Mensch übernimmt so einen zunehmend passiven Teil in der technischen Zukunft und dient mehr als Prüfer der Technik und weniger als Anwender. Einfache Aufgaben wie Einparken oder das Saugen der Wohnung werden automatisiert und durch Sensoren optimiert. Es können Leistungen durch Aktoren erzielt werden, die von Menschen nicht erreichbar sind.

Besonders in der Wirtschaft ist das Gebiet der Industrie 4.0 ein bedeutsames Thema. Diese technische Revolution soll eine intelligente Verbindung von Maschinen und Arbeitsabläufen mit Hilfe von digitalen Informationsschnittstellen ermöglichen. Einfache Lagerhallen werden so zu Datenriesen, die weltweit Informationen über Lagerungsorte und Bedarf austauschen, um schnell und vorausschauend die besten Lagermöglichkeiten oder Versandorte zu bestimmen. Die Fehlerraten sinken und Probleme werden bereits vorab vermieden. Die Lagerhallen werden somit optimal genutzt und jede Maschine arbeitet intelligent, denn die Datenriesen sind lernfähig und können ihr Wissen weitergeben.

Das Wissen wird im sogenannten Internet of Things weitergegeben, das im industriellen Bereich Maschinen, Abläufe, Waren und Menschen verbindet. Im privaten Gebrauch bietet das Internet Massen an Ideen und Konzepten für smarte Technologien. Das smarte Zuhause ist das aktuell bedeutendste Thema für Hausingenieure. Alle Werte im Eigenheim werden geregelt und auf jede Situation angepasst. Der Morgen beginnt mit Kaffee aus der automatisierten Kaffeemaschine, die jeden Tag um sieben Uhr selbstständig zu arbeiten beginnt. Die Heizung reguliert sich anhand der Temperaturwerte und der Luftfeuchtigkeit selbst, sodass das Raumklima angenehm ist. Diese kleinen, oftmals als lästig empfundenen Aufgaben erledigen sich von selbst und bieten damit erheblichen Komfort.

Eine der oft als lästig empfundenen Aufgaben ist das Pflegen von Pflanzen. Im Haushalt sind diese Lebewesen auf den Menschen angewiesen, wenn es um die Versorgung mit Wasser und Licht geht. Schnell geraten diese Aspekte in Vergessenheit und Pflanzen können wie Möbelstücke wirken, da sie sich im Gegensatz zu anderen Lebewesen nicht bewegen.

1.1 Problemstellung und Zielsetzung

Die Bachelorarbeit beschäftigt sich mit dem Thema der automatisierten Versorgung einer Pflanze. Die Arbeit baut auf einer Studienarbeit auf, die an der HAW Hamburg begonnen wurde. Die Idee bestand darin, eine fahrende Plattform für eine Pflanze zu entwickeln, die selbstständig die benötigten Sonnenstrahlen sucht und sich zum Sonnenplatz bewegt. Das Eingreifen einer Person sollte nur in möglichst geringem Maße nötig sein, sodass der Roboter in Abwesenheit von Menschen selbstständig arbeiten kann. Dieser Roboter soll im Living Place ausgestellt werden, der sich in der Fakultät Technik und Informatik befindet. Der Raum ist wie ein gewöhnliches Wohnzimmer aufgebaut und besitzt viele Hindernisse, die vom Roboter umfahren werden sollen.

Das Ziel dieser Arbeit ist es, diesen Gedanken weiterzuführen. Hierfür ist eine Wasserstation geplant, die eine weitere essenzielle Versorgung für die Pflanze bieten soll. In der Natur bekommt die Pflanze wichtige Nährstoffe durch Regenwasser oder Kondenswasser. Dieses Phänomen ist in einem geschlossenen Raum nicht möglich und muss durch Gießen realisiert werden. Schnell wird diese einfache Arbeit vergessen oder mit der falschen Wassermenge ausgeübt. Die notwendige Menge an Wasser ist abhängig von der Pflanze und ihre Fehldosierung ist die häufigste Ursache für das Absterben. Die Station soll die Pflanze optimal mit Wasser befeuchten, sodass eine Intervention des Pflanzenbesitzers nur minimal stattfindet. Die Wasserversorgungsstation soll konstruiert, simuliert und getestet werden. Sie ist nicht mit der Plattform der Pflanze verbunden, sodass der Pflanzenroboter sich immer noch selbstständig im Raum bewegen kann.

1.2 Systemidee

Die Arbeit umfasst demnach die Planung und das Simulieren einer Wasserversorgungsstation, die mit der ebenfalls simulierten fahrbaren Roboterplattform kommunizieren und interagieren kann. Die Roboterplattform und die Versorgungsstation werden mit Hilfe von Gazebo und dem Robot Operating System (ROS) simuliert und getestet. Gleichermaßen baut die drahtlose Kommunikation zwischen den beiden Robotern auf ROS-programmierten Verknüpfungen auf.

Die bereits entworfene, fahrbare Roboterplattform transportiert die Topfpflanze auf ihrer Oberfläche. Ihren Antrieb erhält sie durch integrierte Motoren. Die Plattform wird auf Kompatibilität mit der Versorgungsstation getestet und modifiziert. Der Fokus liegt hierbei hauptsächlich auf dem Pflanzentopf, der die zentrale Schnittstelle in der Interaktion mit der Versorgungsstation während der Wasserzufuhr darstellt. Die Steuereinheit ist im Roboter verbaut und nach IP65 staubfest und gegen Spritzwasser aus einem beliebigen Winkel geschützt. Ein Kontakt mit größeren Wassermengen, beispielsweise durch eine Überflutung der Plattform, führt zum Kurzschluss der Steuereinheit und somit zum Ausfall des Roboters. Infolgedessen soll diese Gefahr durch die Plattformmodifikationen verhindert werden und mögliche Fehler sollen durch ein Signal angezeigt werden.

1.3 Aufbau der Arbeit

Die vorliegende Arbeit setzt sich mit den meisten Bereichen des Studiengangs Mechatronik auseinander. Es werden Grundlagen der Elektrotechnik, der Informatik und des Maschinenbaus behandelt. Zu Beginn werden die einzelnen Komponenten der Wasserversorgungsstation anhand der Requirements evaluiert und ausgewählt. Es folgt die technische Berechnung von Anforderungen an die Wasserversorgungs-

station und die Roboterplattform. Der Hauptteil der Arbeit beschäftigt sich mit dem Zusammenfügen der einzelnen Komponenten zu einem Ganzen. Außerdem wird die Schnittstelle zwischen der Station und der Roboterplattform näher beschrieben. Diese besteht aus der Wasserregulierung und der Kommunikation.
Es folgt die Einarbeitung in Gazebo, wo die entworfenen Mechanismen zur Interaktion, Regulierung und Kommunikation simuliert und getestet werden. Die auftretenden Fehler werden überarbeitet, verbessert und weitere Testläufe der Simulation werden gestartet. Es entsteht ein Entwicklungsprozess, dessen Hergang in der vorliegenden Arbeit nicht mehr nachvollzogen werden kann, da nur das finale, zufriedenstellende Ergebnis beschrieben wird.

2 Analyse der Randbedingungen

Das folgende Kapitel beschäftigt sich mit der Analyse der Randbedingungen, die zur Vorbereitung der Entwicklung des Produkts dient. Das Konkretisieren des konzeptionellen Rahmens ermöglicht eine Einschränkung von Produkteigenschaften und Produktfunktionen. Die Analyse unterteilt sich in die Auseinandersetzung mit Gegebenheiten im Rahmen des Betriebsumfeldes und der Interaktion mit Menschen und Maschinen.

2.1 Living Place

Die Wasserversorgungsstation wird für die Hochschule für Angewandte Wissenschaften entwickelt. Speziell soll sie im Living Place ausgestellt werden und mit den dort befindlichen Pflanzenrobotern zusammenarbeiten. Der Living Place ist eine Wohnung, die das Wohnkonzept der Zukunft darstellen soll. Ihre Gesamtfläche beträgt etwa 140 m² und ist mit den herkömmlichen Räumen einer Wohnung ausgestattet [Liv19]. Die meisten alltäglichen Gegenstände werden hier erweitert oder

automatisiert, sodass ein Großteil der Geräte miteinander kommuniziert und selbstständig verschiedene Prozesse ausführt. Beispielsweise befindet sich im Badezimmer ein interaktiver Spiegel, der Neuigkeiten aus der Wirtschaft präsentiert oder das Wetter für die kommende Woche, wenn dies vom Benutzer gewünscht wird.

2.2 Stakeholder

Die Stakeholderanalyse beschreibt, welche Maßnahmen der Projektleiter zum erfolgreichen Abschluss der Arbeit ergreifen muss. Dabei können nicht immer alle Interessen der Mitwirkenden berücksichtigt werden. Der Kernpunkt dieser Analyse ist es, zu erkennen, welche Einflüsse und Werte auf die Wasserversorgungsstation einwirken. Es stellen sich die Fragen: *Wie wird der Kunde die Station bedienen?*, *Wo wird die Station stehen?* und *Soll die Wasserversorgungsstation durch andere erweiterbar sein?* Die Analyse der Stakeholder gleicht einem einfachen linearen Prozess. Zuerst müssen die Stakeholder identifiziert und anschließend anhand ihres Einflusses bewertet werden. Die Tabelle 1 ist an [Leh19] angelehnt und stellt die Stakeholderanalyse für die Wasserversorgungsstation dar.

Tabelle 1: Stakeholderanalyse

Stakeholder und Wichtigkeit	Interessen
Mitarbeiter des Living Place (Primär)	Keine Beeinträchtigung durch Wasser- und Elektroschäden, wenig Platzeinnahme, stört nicht durch Aussehen, kein elektrischer Schlag bei Berührung
Professor (Primär)	Erweiterung der Wasserversorgungsstation und der Roboterplattform, Bewertung und Vorführung, kein elektrischer Schlag bei Berührung
Studenten (Primär)	Lernen und Erweitern der Wasserversorgungsstation und der Roboterplattform, kein elektrischer Schlag bei Berührung
Entwickler (Primär)	Konstruieren, Programmieren, Fehleranalyse des Systems
Reinigungspersonal (Sekundär)	Keine Beeinträchtigung durch Platzmangel, keine Wasserüberflutung
Facilitymanagement (Sekundär)	Keine Beeinträchtigung durch Wasser- und Elektroschäden
Andere Roboter/Geräte (Sekundär)	Keine Störung durch Signale und Interaktionen

Die Stakeholder werden in primäre und sekundäre Stakeholder unterteilt. Dabei sind die primären Interessengruppen die wichtigsten und werden am häufigsten mit dem Projekt arbeiten. Hierbei kommen die Anforderungen größtenteils von den Mitarbeitern des Living Place und den Professoren, die die Wasserversorgungsstation in den Living Place integrieren. Sie sind der Kunde, auf dessen Wunsch die Wasser-versorgungsstation gebaut wird. Es folgen die Studierenden und der Entwickler, die ebenfalls wiederholt mit dem Projekt konfrontiert werden. Ihr Interessenschwerpunkt liegt auf der Sicherheit und dem aus dem Produkt gezogenen Nutzen.

Die zweite Stakeholdergruppe ist die sekundäre Gruppe. Sie wirkt passiv auf die Entwicklung ein. Das Reinigungspersonal und das Facilitymanagement arbeiten nicht direkt mit der Wasserversorgungsstation, dennoch haben sie Kontakt und stellen ebenfalls Anforderungen an die Konstruktion. Hierzu zählen beispielsweise die Stromversorgung oder die Beeinflussung der Reinigung durch die Größe der Station. Ein weiterer wichtiger Aspekt ist ein möglicher Defekt der Wasserversorgung. Es könnte Wasser in den Raum austreten und den Boden verunreinigen oder sogar andere Geräte bei Kontakt zerstören. Dieser Fall ist ein sogenanntes Worst-Case-Szenario, sollte aber mitbedacht werden, um sich im Voraus dagegen abzusichern.

2.3 Anforderungen der Wasserversorgungsstation

Durch die Stakeholderanalyse kristallisieren sich erste Anforderungen an die Wasserversorgungsstation heraus. Diese Ergebnisse werden weiterverarbeitet und weiterverfolgt.

2.3.1 Use-Case

Um die einfache Interaktion eines Systems mit der Umwelt darzustellen, ist es wichtig, das Verhalten anhand eines Anwendungsfalles aufzuzeigen. So werden Konzeptideen gewonnen, die auf das Design und die Schnittstellen einwirken. Eine Möglichkeit hierfür ist die Nutzung eines Use-Cases. Er beschreibt das System aus Anwendersicht und verdeutlicht dessen Verhalten im Rahmen eines speziellen Anwendungsfalles. Der Aufbau richtet sich nach dem Use-Case von [Leh19].

Titel: Versorgung einer mobilen Zimmerpflanze mit Wasser
Akteur: Wasserversorgungsstation, Roboterplattform mit Zimmerpflanze
Ziel: Gießen einer Zimmerpflanze, die sich auf einer Roboterplattform befindet
Auslöser: Roboterplattform steht vor der Wasserversorgungsstation

Vorbedingungen:

- Wasserversorgungsstation eingeschaltet
- Roboterplattform kann sich bewegen
- Roboterplattform ist eingeschaltet
- Pflanze befindet sich auf der Plattform
- Wassertank enthält Wasser
- alle Sensoren funktionieren ohne Fehler
- Roboterplattform befindet sich vor der Station

Nachbedingungen:

- Roboterplattform bewegt sich von der Wasserversorgungsstation weg
- Wasserversorgungsstation bewegt sich in den Anfangszustand

Erfolgsszenario:

1. Die Roboterplattform mit Zimmerpflanze fährt vor die Wasserversorgungsstation in eine toleranzbehaftete Position und Ausrichtung.
2. Die Roboterplattform sendet Daten an die Wasserversorgungsstation, dass sie versorgt werden möchte.
3. Die Wasserversorgungsstation misst die Position für die Höhe und Entfernung, um die Versorgung der Pflanze zu gewährleisten.
4. Die Wasserversorgungsstation lässt das Wasser oben in den Topf fließen.
5. Die Wasserversorgungsstation erkennt, dass die Wasserversorgung erfolgreich ist.
6. Die Roboterplattform verlässt die vorgesehene Position.
7. Die Wasserversorgungsstation bekommt die Information, dass die Roboterplattform die Position verlassen hat.
8. Die Wasserversorgungsstation geht in Startposition.

Fehler:

3.a. Roboterplattform erhält das Signal, dass sie sich falsch positioniert hat und positioniert sich neu.

4.a Es fließt kein Wasser, da der Tank leer ist. Wasserversorgungsplattform gibt diese Information an die Roboterplattform.

4.b Wasserversorgungsstation signalisiert dem Nutzer, dass kein Wasser im Tank ist.

2.3.2 Informationsfluss

Der Informationsfluss beschreibt die Beziehungen zwischen der Wasserversorgungsstation und anderen Systemen. Mit Hilfe der Systems Modeling Language (SysML) kann dies anhand eines Block Definition Diagram (bdd) vereinfacht dargestellt werden. In Abbildung 1 wird der Systemkontext der Wasserversorgungsstation veranschaulicht, der dem Aufbau aus [Leh19] [Dai17] entnommen wurde.

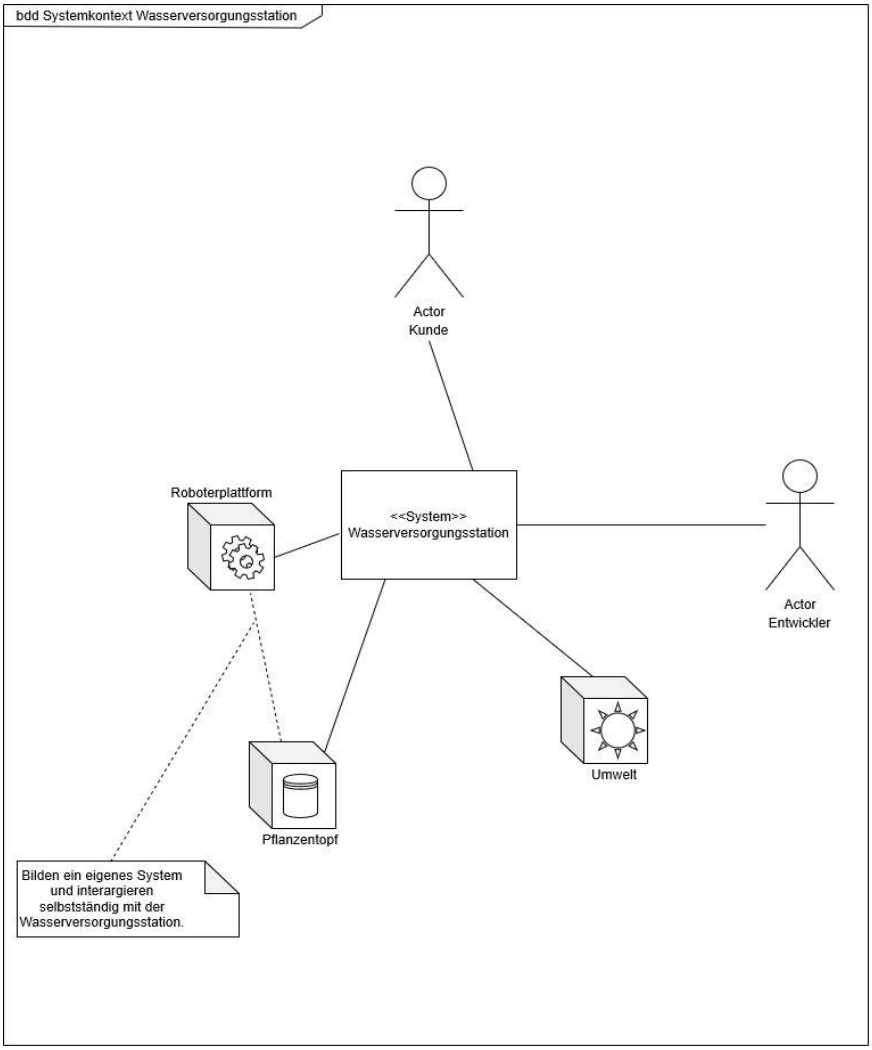

Abbildung 1: bdd Systemkontext Wasserversorgungsstation

Die Systemgrenzen werden statisch bestimmt, sodass die Wasserversorgungsstation unabhängig von der Roboterplattform betrachtet und weiterentwickelt werden kann. Trotzdem besitzt das System eine Beziehung und somit auch eine Schnittstelle zu den anderen Systemen. Die Schnittstellen müssen nicht unbedingt ein Datenaustausch, sondern können auch eine Kraftauswirkung oder Interaktion auf Bedienungsbasis sein. Somit weist das Projekt ebenfalls Schnittstellen mit dem Pflanzentopf, dem Kunden, dem Entwickler und der Umwelt auf.

2.4 Anforderungen der Roboterplattform

Bei dem mobilen Roboter handelt es sich um einen Pioneer 3-DX (siehe Abbildung 2). Er besitzt zwei motorisierte Räder, die sich ideal für das Fahren auf glatten Oberflächen eignen.

Abbildung 2: Pioneer 3-DX mit freier Plattform

Die wichtigsten Daten [Zpk3] des Pioneer 3-DX sind:

- Gewicht: 9 kg
- maximales Beladungsgewicht: 17 kg
- Batteriespannung: 12 V
- Batteriekapazität: 3 x 7.2 Ah
- Spannungsversorgung: 5 V bei 1.5 A und 12 V bei 2.5 A
- maximale Geschwindigkeit: 1.2 m/s
- Rotationsgeschwindigkeit: 300 °/s

Des Weiteren ist der Pioneer mit mehreren Sensoren ausgestattet. Dabei handelt es sich um einen nicht serienmäßig mitgelieferten LiDAR-Scanner, der für die Orientierung und das selbstständige Fahren im Raum zuständig ist. Außerdem besitzt der Roboter serienmäßig Ultraschallsensoren, die am vorderen Ende des Roboters montiert sind, und Kodierer, die für die Reproduktion der Bewegung zuständig sind. Um mit der Wasserversorgungsstation arbeiten zu können, benötigt der Pioneer weitere Anforderungen. Dazu gehört die Kommunikation mit der Wasserversorgungsstation, eine Vorrichtung für den Transport einer Vase und Angaben zur Feuchtigkeit der Pflanzenerde in der Vase.

2.4.1 Informationsfluss

Der Informationsfluss des Pioneer 3-DX ähnelt der Wasserversorgungs-station. Dieser wird ebenfalls mit Hilfe eines bdd vereinfacht dargestellt [Leh19] [Dai17].

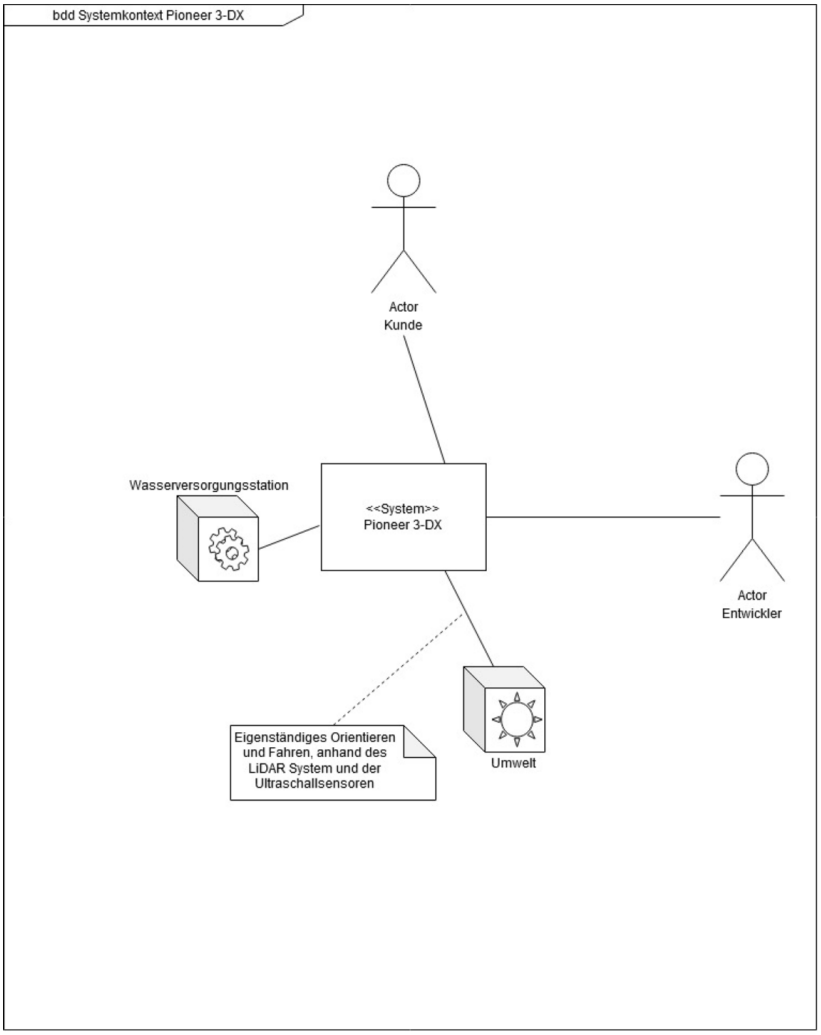

Abbildung 3: bdd Pioneer 3-DX

Ebenso besitzt der Pioneer 3-DX je eine Schnittstelle mit dem Kunden, dem Entwickler und der Umwelt (siehe Abbildung 3). Anders als bei der Wasserversorgungsstation interagiert der Roboter aktiv mit der Umwelt. Durch das Navigieren werden mehr Informationen über die Umwelt ausgewertet, sodass beispielsweise keine Kollision stattfinden kann. Die Navigation des Roboters und seine Position in Relation zur Wasserversorgungsstation sind durch die Verarbeitung der Umweltdaten

gewährleistet. Mit Hilfe eines integrierten WLAN-Adapters ist der Pioneer mit dem örtlichen Netzwerk verbunden, wodurch auch eine Kommunikation mit einem anderen System realisiert werden könnte.

2.5 Zusammenfassung der Requirements

Einer der wichtigsten Schritte ist das Aufstellen der Requirements. Dies ist ein rechtlich verbindlicher Schritt in der Zusammenarbeit mit dem Kunden, der eine Art Vertrag darstellt. Die aufgelisteten Anforderungen sind der Soll-Zustand des Endprodukts für den Kunden. Die Requirements können durch mehrere Methoden entwickelt werden. Die wichtigste Methode ist das Kundengespräch. Die Requirements bilden sich mit jedem Gespräch weiter und werden spezifischer formuliert, sodass die Kundenzufriedenheit letztendlich gewährleistet ist. Im Verlauf des Entwicklungsprozesses können sich Anforderungen verändern, da zum Beispiel der Kunde neue Erkenntnisse gewonnen hat oder sich Probleme im Design entwickeln. Aus diesem Grund sollte die Liste kontinuierlich gepflegt und aktualisiert werden.

Die Tabelle 2 ist an [Dai17] angelehnt und stellt die finale Version der Requirements für das Produkt der Wasserversorgungsstation dar. Jede Anforderung enthält eine Kennzahl, die mit weiteren Gliederungsnummern detaillierter beschrieben wird. Die Gliederungsnummern wurden bei der Konzeptentwicklung nachgetragen. Zudem lässt sich jedes Requirement in einen Themenbereich einordnen. Es beschreibt eindeutig, ob die Anforderung auf einer rechtlich verbindlichen Vorschrift, einem Gesetz oder Sonstigem beruht. Hierfür werden die englischen Verben *must*, *shall* und *should* verwendet. Für eine gesetzliche Forderung wird *must* verwendet und für sonstige Forderungen *shall*. Bei einem Wunsch und weniger relevanten Anforderungen wird *should* verwendet. Neben dem Gespräch mit dem Kunden müssen auch Annahmen durch den Entwickler getroffen werden.

Diese sind speziell auf den Living Place abgestimmt und lauten:

- Im Living Place gibt es eine konstante WLAN-Verbindung.
- Im Living Place herrschen Raumtemperaturen von 18 bis 30 °C.
- Die Wasserversorgungsstation wird auf ebenem Boden platziert.
- Der Raum besitzt einen Stromzugang von 230 Volt.

Tabelle 2: Requirements der Stakeholder Teil 1

Nummer	Kategorie	Beschreibung	Klassifikation	Stakeholder
1	Konstruktion	Die Wasserversorgungsstation soll Wasser speichern können.	Must	Kunde
1.1	Konstruktion	Die Wasserversorgungsstation wird mindestens zwei Liter Wasser speichern können.	Shall	Kunde
1.1.1	Konstruktion	Die Wasserversorgungsstation wird vom Kunden befüllt.	Shall	Kunde
1.2	Konstruktion	An der Wasserversorgungsstation soll eine Schutzmaßnahme für ungewollten Wasseraustritt vorgenommen werden.	Must	Kunde, Entwickler, Mitarbeiter des Living Place
1.3	Konstruktion	Die Wasserversorgungsstation soll bei leerem Wassertank ein Signal für den Kunden anzeigen.	Should	Kunde
2	Konstruktion	Die Wasserversorgungsstation wird im Living Place eingesetzt.	Shall	Kunde
2.1	Konstruktion	Die Wasserversorgungsstation wird nicht mobil sein.	Must	Kunde
2.2	Konstruktion	Die Wasserversorgungsstation wird nicht an einer Wand oder am Boden befestigt.	Must	Kunde
2.2.1	Konstruktion	Die Wasserversorgungsstation soll sicher auf dem Boden stehen.	Must	Kunde
3	Konstruktion	Die Wasserversorgungsstation soll eine handelsübliche Topfpflanze mit Wasser versorgen können.	Should	Kunde
3.1	Konstruktion	Der Topf soll von oben mit Wasser befüllt werden.	Shall	Entwickler
3.1.1	Konstruktion	Der Topf soll durch abnehmbare Hilfsmittel für die Wasserversorgung erweitert werden können.	Shall	Entwickler
3.1.2	Konstruktion	Der Topfrand muss eine Höhe zwischen 15 und 45 Zentimeter über dem Boden haben.	Must	Entwickler

3.1.3	Konstruktion	Der innere Topf muss durch den Manipulator in 5 bis 25 cm Entfernung erreichbar sein.	Must	Entwickler
3.2	Konstruktion	Die Wasserversorgungsstation soll keine starre Verbindung mit dem Topf herstellen.	Shall	Kunde
3.3	Mechanik, Sensorik	Der Versorgungsprozess soll nicht länger als 120 Sekunden dauern.	Shall	Kunde
3.4	Sensorik	Die Wasserversorgungsstation soll erkennen, wenn ein Topf vor ihr zur Versorgung steht.	Shall	Kunde
3.5	Sensorik	Die Wasserversorgungsstation wird 100 ml Wasser abgemessen fördern.	Should	Kunde
4	Konstruktion	Der Topf wird auf einer mobilen Roboterplattform verbaut.	Shall	Kunde
4.1	Konstruktion	Der Topf muss auf der Roboterplattform stabilisiert sein.	Should	Kunde
4.2	Mechanik, Sensorik	Die Roboterplattform muss den Topf vor die Wasserversorgungs-station an eine mit Toleranz behaftete Position bringen, die vom Manipulator angefahren werden kann.	Must	Kunde
5	Mechanik	Die Wasserversorgungsstation soll den Topf mit einem Mani-pulator anfahren.	Must	Kunde
5.1	Mechanik	Der Manipulator soll sich nicht drehen können.	Shall	Kunde
5.2	Mechanik	Der Manipulator soll sich 15 bis 45 Zentimeter vom Boden heben können.	Shall	Kunde
5.3	Mechanik	Der Manipulator soll zwischen 5 und 25 Zentimeter von der Wand der Wasserversorgungsstation ausfahrbar sein.	Shall	Kunde
5.4	Mechanik, Sensorik	Der Manipulator soll die Position für die Befüllung messen können.	Must	Kunde
5.4.1	Sensorik, Sensorik	Der Manipulator soll sich selbstständig für die Befüllung ausrichten können.	Must	Kunde
6	Sensorik, Informatik	Die Roboterplattform soll ein eigenständiges System bilden.	Must	Kunde
6.1	Sensorik, Informatik	Die Roboterplattform soll mit der Wasserversorgungsstation Daten austauschen.	Shall	Kunde

6.2	Sensorik, Informatik	Die Roboterplattform wird selbstständig die Werte der Pflanze messen und verarbeiten können.	Must	Kunde
7	Konstruktion	Die Wasserversorgungsstation soll bei Kollision mit der Umwelt funktionsfähig bleiben.	Should	Kunde, andere Roboter
8	Elektrotechnik	Die Wasserversorgungsstation besitzt keine eigene Energie-versorgung.	Must	Kunde, Entwickler

3 Konzeptentwicklung

Das folgende Kapitel beschreibt den Entwicklungsprozess der Wasser-versorgungsstation. Die ersten Abschnitte behandeln das Verständnis der Problematik und bauen auf der Anforderung der Wasserversorgungsstation auf. Sie gehen zudem auf den Datenaustausch und die benötigten Funktionen für das erfolgreiche Ausführen der Bewässerung ein. Zum Schluss des Kapitels wird ein fertiges Konzept mit ausgewählten Komponenten präsentiert.

3.1 Objektflüsse

Die SysML enthält als weitere Darstellung das Internal Block Diagram (ibd). Das ibd ähnelt dem bdd und bietet einen Überblick über die Objektflüsse [Dai17]. Dabei wird nicht nur der Datenaustausch berücksichtigt, sondern auch die Auswirkung von Kraft auf das System der Wasserversorgungsstation. Es ist zu beachten, dass immer eine Gegenwirkung in Form von Actio und Reactio stattfindet.

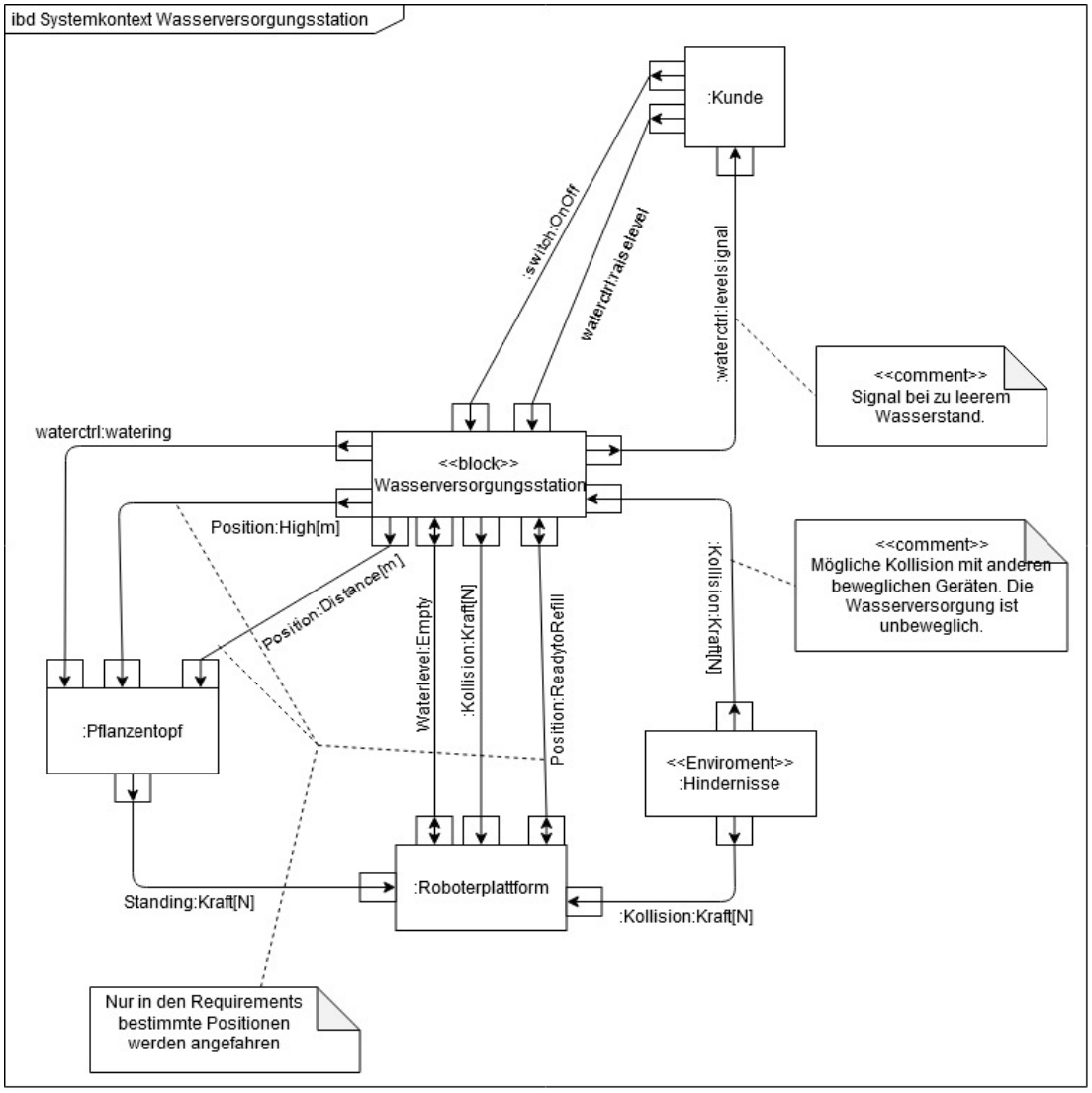

Abbildung 4: ibd Wasserversorgungsstation

Diese Gegenwirkung ist in Abbildung 4 bei der Interaktion der Roboterplattform und der Wasserversorgungsstation zu beobachten. Die Station ist ein eigenständiges, nichtmobiles System, das durch die Roboterplattform angefahren werden kann. Die Kollision wird mit der Kraft in Newton symbolisiert. Eine Kollision kann auch mit dem Pflanzentopf stattfinden, jedoch ist der Pflanzentopf ein Teil der Roboterplattform und bildet ein eigenes System. Trotzdem gibt es spezielle Interaktionen, die unabhängig von der Roboterplattform durchgeführt werden. Dazu gehören die Positionsfindung und die Bewässerung. Deshalb wird für einen besseren Überblick der Pflanzentopf ebenfalls als eigener Block dargestellt.

Das System sollte möglichst autonom arbeiten, wodurch keine Interaktion mit dem Kunden während der Bewässerung nötig ist. Dennoch ist der Kunde für die Bedienung und Befüllung der Wasserversorgungsstation zuständig.

3.2 Sicherheitsanalyse

Die Sicherheitsanalyse ist wichtig für die Erkennung der möglichen Fehlfunktion und deren Auswirkung. Bei diesem Projekt wird primär auf die Sicherheit des Menschen geachtet und nur sekundär auf die Sicherheit der Umwelt. Dabei wird nicht auf die Wasserversorgungsstation in der Simulation eingegangen, sondern auf die Risiken einer realen Station. Für die Analyse wird eine FMEA-Tabelle (Failure Modes and Effects Analysis) nach [Ihi19] [GdR15] genutzt, die auf für das Projekt relevante Punkte eingeschränkt worden ist (siehe Tabelle 3).

Tabelle 3: FMEA-Tabelle der Sicherheitsanalyse

Process step	Failure	Failure cause	Failure effect	Actions to reduce the occurence
Anfahren mit der Roboterplattform	Roboterplattplattform fährt gegen die Wasserversorgungsstation.	Kollisionsrückmeldung nicht vorhanden	Verschieben des Ursprungspunkt der Wasserversorgungsstation und Beschädigung der Station.	Hinzufügen eines Kontaktsensors an die mobilen Roboter; Hinzufügen von Leitkomponente in Form einer Planke oder Bodenlinie für den mobilen Roboter
Bewegung der Achsen bei der Wasserversorgung	Einklemmung von Gliedmaßen oder Kleinteilen	Berührung beim Bewegen der Achsen	Verletzung von Personen und Zerstörung der Motoren	Anbau einer Verkleidung, die das Berühren der Achsen erschwert; Notausschalter
Bewässerung	Wasseraustritt	Wasserleitung beschädigt	Wassereintritt in die Station und möglicher Kurzschluss	Isolierung der Komponenten, die mit Wasser arbeiten in einen eigenen versiegelten Raumabschnitt; Fördern von kleineren Wassermengen; Einbau einer Sicherung gegen Kurzschluss; Einbau eines Wassersensors fürs schnellere Detektieren
Station im Betrieb	Stromschlag bei Berührung von Komponenten und Stromleitung	Schlecht isolierte Stromleitung	Kurzschluss oder Gefährdung von Personen	Einbau einer Sicherung; Zugriff auf die Komponenten bei anliegender Spannung verwehren; Notausschalter

Die genannten Lösungsvorschläge für die Reduktion von Fehlern werden in der Simulation nicht beachtet, da sie nur schwer getestet werden können. Die einzige Möglichkeit wäre das Testen mit mehreren Kontaktsensoren. Aufgrund des hohen Aufwandes und der niedrigen Relevanz für die Simulation wurde die Sicherheitsanalyse nur theoretisch aufgelistet und nicht weitergeführt.

3.3 Funktionsanalyse

Die Funktionsanalyse nach [Abu19] beschreibt abstrakt den Soll-Zustand des Endprodukts. Die Stakeholder und die Requirements beschreiben die Hauptfunktionen, die in Unter- und Teilfunktionen gegliedert werden. Die Teilfunktion bildet in Tabelle 4 die unterste Ebene und muss keine weitere Ebene besitzen, sofern die Funktion vereinfacht ist.

Tabelle 4:Funktionsanalyse der Wasserversorgungsstation

Funktionsanalyse					
Nr.	**Hauptfunktion**	**Nr.**	**Unterfunktion**	**Nr.**	**Teilfunktion**
F1	Wasserübertragung in den Pflanzentopf	F1.1	Position Roboterplattform ermitteln	F1.1.1	Bestimmung der Erreichbarkeit der Roboterplattform
		F1.2	Manipulator zum Topfrand positionieren	F1.2.1	Topfrandhöhe bestimmen
				F1.2.2	Entfernungsmessung vom Topf
		F1.3	Topf mit dem Manipulator anfahren	F1.3.1	Lösbare Verbindung zum Topf herstellen
				F1.3.2	Manipulator heben und senken
				F1.3.3	Manipulator ausfahren
		F1.4	Wasser zum Topf befördern	F1.4.1	Wasserdurchfluss prüfen
				F1.4.2	Wasser aus dem Wassertank pumpen
F2	Wasser bevorraten	F2.1	Ausgabe eines Signals beim leeren Tank		
		F2.2	Mindestens 2 Liter Wasser bevorraten		
F3	Daten austauschen	F3.1	Statusinformation innerhalb der WVS senden und empfangen		
		F3.2	Statusinformationen mit der Roboterplattform austauschen		

Auf den ersten Blick erscheint diese Analyse überflüssig, jedoch ist das Risiko, eine Funktion zu übersehen, sehr groß, wenn die Entwicklung zu schnell vorangeht. Zudem bietet die Analyse ein einheitliches Verständnis, sollten mehrere Personen an dem Projekt beteiligt sein. Erst wenn alle Personen die gleichen Funktionen erkennen und verstehen, können sie erfolgreich zusammenarbeiten.

3.4 Morphologischer Kasten

Mit Hilfe der Funktionsanalyse kann ein morphologischer Kasten erstellt werden. Dabei bilden die untersten Ebenen der Funktionsanalyse die Problemfunktion, für die eine Lösungsvariante gefunden werden soll. Dieser kreative Schaffensprozess bezieht sich nicht auf die Findung einer optimalen Lösung, sondern auf das Ausleben der

Kreativität. Es sollen zahlreiche Möglichkeiten gefunden werden, sodass diese diskutiert und vergleichend bewertet werden können. Nicht jede Lösung für eine Funktion ist mit anderen Lösungsansätzen kompatibel. Diese Inkompatibilität bei der Systemintegration muss erkannt und durch Kombinationsmöglichkeiten ausgearbeitet werden.

Tabelle 5: Morphologischer Kasten

Funktion	Variante 1	Variante 2	Variante 3	Variante 4
Positionsbestimmung der Roboterplattform für die WVS	RFID	Indoor Positioning System	433 Mhz Signal	Ultraschall
Topfrandhöhe bestimmen	Ultraschall	Reflexlichtschranke	Infrarotentfernungsmesser	Kameraauswertung
Entfernungsmessung vom Topf	Ultraschall	Reflexlichtschranke	Infrarotentfernungsmesser	Kameraauswertung
Lösbare Verbindung zum Topf herstellen	Saugnapf	Drehmechanismus	Keine Berührung	Greifer
Manipulator heben und senken	Schrittmotor Riemenantrieb	Hydraulikzylinder	Servomotor mit ausfahrbarer Zahnstange	Spindelantrieb
Manipulator ausfahren	Schrittmotor Riemenantrieb	Hydraulikzylinder	Servomotor mit ausfahrbarer Zahnstange	Spindelantrieb
Wasserdurchfluss prüfen	Füllstandsensor	Ultraschalldurchflusssensor	Wasserdurchflusssensor mit Flügelrad	magnetisch-induktiver Durchflusssensor
Mindestens 2 Liter Wasser bevorraten	Fester Wassertank	Einsetzbarer Wassertank		
Wasser aus dem Wassertank fördern	Ansaugpumpe	Tauchpumpe	Kolbenpumpe	Linearmembranpumpe
Ausgabe eines Signals beim leeren Tank	Led	7-Segmentanzeigen	Display	Buzzer
Statusinformation innerhalb der WVS senden und empfangen	Arduino	Raspberry Pi	Beaglebone	PICmicro
Statusinformationen mit der Roboterplattform senden und empfangen	Bluetooth	Wlan	433 Mhz Signal	Infrarot

In Tabelle 5 nach [Abu19] sind die Problemfunktionen der Wasserversorgungsstation aufgelistet und anhand von maximal vier Varianten beschrieben. Die Varianten sind einfache Ideen, die als Beispiellösungen für das Problem dienen.

3.5 Konzeptauswahl

Die Varianten des morphologischen Kastens werden miteinander verglichen und durch die Kombinationsmöglichkeiten verbunden und bewertet [Zmk19]. Hierfür wird in jeder Zeile eine Variante ausgewählt und mit einer weiteren Variante in der nachfolgenden Funktion verbunden. Dadurch entsteht ein Weg, der ein fertiges Konzept beschreibt. Nicht jeder Weg ist jedoch als gut zu bewerten, da beispielsweise das Zusammenspiel mancher Sensoren nicht möglich ist. Die Wegfindung kann unter dem Pfad *CD:/Entwicklung und Simulation einer Wasserversorgungsstation für mobile Pflanzen\Bachelorthesis\Tabellen* nachvollzogen werden.

Es wurden drei Wege zur Auswahl von Konzepten genutzt. Diese werden in der folgenden Tabelle 6 anhand der getroffenen Auswahl dargestellt.

Tabelle 6: Vorstellung der ausgewählten Konzepte

Konzept 1	Konzept 2	Konzept 3
Ultraschall	WLAN-Datenaustausch	Indoor Positioning System
Ultraschall	Reflexlichtschranke	Kameraauswertung
Ultraschall	Infrarotentfernungsmessung	Kameraauswertung
Greifer	Keine Berührung	Greifer
Hydraulikzylinder	Spindelantrieb	Servomotor mit ausfahrbarer Zahnstange
Spindelantrieb	Servomotor mit ausfahrbarer Zahnstange	Spindelantrieb
magnetisch-induktiver Durchflusssensor	Wasserdurchflusssensor mit Flügelrad	Ultraschalldurchflusssensor
Kolbenpumpe	Ansaugpumpe	Tauchpumpe
Buzzer	Led	Display
Arduino	Raspberry Pi	Raspberry Pi
Fester Wassertank	Einsetzbarer Wassertank	Einsetzbarer Wassertank
Bluetooth	WLAN-Datenaustausch	433-MHz-Signal

Die beste Lösung wird in Tabelle 7 anhand mehrerer Kriterien ausgewählt, die eine wichtige Rolle für das Simulieren und mögliche Implementieren spielen. In Anlehnung an [Zmk19] wird anhand der Direct-Ranking-Methode eine Konzeptbewertungstabelle erstellt.

Tabelle 7: Konzeptbewertung anhand der Direct-Ranking-Methode

			Solution Sets					
			Konzept 1		Konzept 2		Konzept 3	
Nr.	Criteria	Weights	achieved	total points	achieved	total points	achieved	total points
1	Konstruktionsaufwand	0,2	8	1,6	7	1,4	3	0,6
2	Kosten	0,1	7	0,7	7	0,7	5	0,5
3	Störanfälligkeit	0,1	2	0,2	7	0,7	4	0,4
4	Gewicht und Größe	0,2	6	1,2	7	1,4	6	1,2
5	Programmierungsaufwand	0,15	9	1,35	8	1,2	2	0,3
6	Fertigungsaufwand	0,15	5	0,75	5	0,75	4	0,6
7	Wartung	0,1	4	0,4	7	0,7	5	0,5
	Total	1		0,62		0,685		0,41
	Ranking			2		1		3

Die Auswahl der Kriterien basiert auf der Erfahrung des Entwicklers mit ähnlichen Produkten. Die anschließende Gewichtung sollte mit dem Kunden besprochen werden, da Prioritäten oft unterschiedlich gesetzt werden.
In diesem Projekt sind die wichtigsten Kriterien der Konstruktionsaufwand, das Gewicht und die Größe des Projektes, der Fertigungsaufwand und der Programmierungsaufwand. Dabei erhalten die Kriterien die höchsten Gewichtungen in der Spalte *Weights*. Jedes Konzept wird anhand einer Skala von 0 bis 10 bewertet und mit der Gewichtung verrechnet. Die 0 symbolisiert den schlechtesten und die 10 den besten Wert. Die erreichten Werte sind in *achieved* gelistet und werden mit den *Weights* multipliziert. Der Produktwert wird in den *total points* aufgeschrieben. Alle Werte der *total points* werden anschließend summiert, sodass eine maximale Summe von 1 erreicht werden kann. Diese Ergebnisse werden in *Total* angezeigt. Anhand dieser Werte kann ein Ranking erstellt werden, wodurch das beste Konzept herausgefiltert werden kann. Das Konzept 2 erhält die Nummer eins im Ranking, da die Totalwertung 0.685 beträgt und somit der maximalen Wertung am nächsten ist.

3.6 Wasserversorgung

Am Anfang des Projekts stellt sich die einfache Frage: Wie nimmt die Pflanze das Wasser auf? – Diese Frage muss beantwortet werden und anhand der Antwort sollte die Pflanzenversorgung konstruiert werden.
Bei den meisten Zimmerpflanzen ist der Wassertransport ein Prozess, der hauptsächlich über die Wurzel vollzogen wird. Dabei gibt es Ausnahmen, zum Beispiel Pflanzen, die über die Blattoberfläche Kondenswasser aufnehmen. Dieses Phänomen kann im geschlossenen Raum nicht realisiert werden, wodurch einzig die Bewässerung über die Wurzel bleibt.

3.6.1 Möglichkeiten des Transports

Das Resultat der Anfangsfrage ist, dass das Wasser zur Wurzel und somit in die Erde des Topfes gelangen muss. Dies kann auf mehreren Wegen realisiert werden. Die erste Idee war das direkte Einspritzen in die Erde, indem die Topfwand einen Zugang für die Wasserversorgungsstation besitzt. Das Problem wäre die Veränderung des Topfes, die aufwändig wäre und den Topf unästhetisch aussehen ließe. Aus diesem Grund ist die Entscheidung auf die Bewässerung von oben gefallen. So müssen wenig bis keine Veränderungen am Topf vorgenommen werden und Töpfe oder Vasen sind für den Kunden frei wählbar.

3.6.2 Veränderung am Pflanzentopf

Der Pflanzentopf muss für die Wasserversorgungsstation und insbesondere für die Reflexlichtschranke kenntlich gemacht werden. Dabei sollte ein Reflektor benutzt werden [Ref20], wie in Abbildung 5 dargestellt. Dieser wird an den oberen Topfrand montiert und gilt als Marker für die Reflexlichtschranke. Ein falsches Anbringen bedeutet auch eine falsche Justierung der Achsen durch die Wasserversorgungsstation, wodurch die Pflanze im schlimmsten Fall nicht bewässert wird und das Wasser auf die Roboterplattform gelangen könnte. Dies könnte zu Schäden der elektrischen Komponenten in der Roboterplattform führen. Der Reflektor sollte, wenn möglich, rechteckig sein und in der Breite mit dem Pflanzentopf übereinstimmen. Hierdurch wird die Wahrscheinlichkeit einer Erkennung durch die Reflexlichtschranke erhöht.

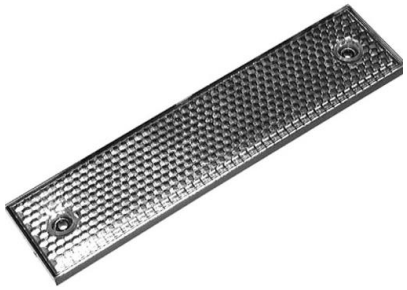

Abbildung 5: Rechteckiger Reflektor

Der größte Vorteil dieser Veränderung ist, dass sie den Pflanzentopf nicht irreparabel verändert und durch den Kunden entfernt werden kann. Dadurch ist es möglich, jeden Behälter mit einem Reflektor auszustatten und für die Versorgung mit Wasser zu modifizieren. Die Anzahl der Roboterplattformen mit Pflanzen, die mit Wasser versorgt werden müssen, kann so problemlos erhöht werden.

3.7 Grundkonstruktion

Die Grundkonstruktion wird in Abbildung 6 und Abbildung 7 anhand des ausgewählten Konzepts zwei dargestellt. Die Basis für die Wasserversorgungsstation bildet ein Standfuß aus Holz, der für das sichere Aufstellen im Living-Place zuständig ist. Dieser wird nicht am Boden oder an der Wand befestigt, wodurch eine freie Aufstellung im Raum möglich ist. Die Konstruktion des Standfußes beinhaltet im Inneren drei Kammern.

Die erste Kammer dient zur Platzierung des Wassertanks. Sie ist nicht mit den anderen Kammern verbunden und soll als Auffangbecken für Wasser dienen, sofern der Wassertank beschädigt werden sollte. Somit ist gewährleistet, dass kein Wasser zu den elektrischen Komponenten in die weitere Kammer gelangen kann. Der Wassertank besteht aus einer 2-Liter-PET-Flasche, welche einen Adapter als Flaschendeckel besitzt. Dieser Adapter ermöglicht das Einsetzen eines Schlauches in die Flasche, welcher mit der Ansaugpumpe verbunden ist.

In der zweiten Kammer befinden sich die Ansaugpumpe und der Wasserdurchflusssensor. Beide Komponenten sind ebenfalls durch einen Schlauch miteinander verbunden und für den Transport von Wasser zuständig.

Die dritte Kammer beinhaltet den Raspberry Pi sowie ein 12-Volt-Netzteil, das alle elektrischen Komponenten mit Spannung versorgt. Diese Kammer ist mit der zweiten über ein kleines Kabelloch verbunden, das vor Inbetriebnahme versiegelt worden ist.

Der Standfuß besitzt einen Holzdeckel, der die Spindelführung und den Servomotor mit der ausfahrbaren Zahnstange [Sec19] trägt. Außerdem sind die Schalter und die LEDs auf dem Deckel angebracht. Die erste Achse liegt in Form eines Spindelantriebs vor, der für das Anpassen der zweiten Achsenhöhe zuständig ist. Sie bewegt die

ausfahrbare Zahnstange in die gewünschte Höhe. Anhand einer Reflexlichtschranke und eines Infrarotentfernungsmessers wird die Topfpflanze erkannt und mit dem Spindelantrieb auf passender Höhe angefahren. Durch die ausfahrbare Zahnstange wird die Entfernung zum Topf überwunden. Am Ende der Zahnstange ist der Schlauch fixiert, der an der Ansaugpumpe befestigt ist. So kann das Wasser aus dem Schlauch in den darunter befindlichen Topf fließen.

Der Servomotor mit der ausfahrbaren Zahnstange befindet sich auf einer Plattform, die am Spindelantrieb angebracht ist. Die Plattform ist nicht käuflich erhältlich, sondern muss mit einem 3D-Drucker hergestellt werden. Unterhalb der Plattform wird die Reflexlichtschranke fixiert und an der Vorderseite der Plattform wird der Infrarotentfernungsmesser angebracht. Beide Sensoren müssen übereinander liegen, sodass die gleichen Punkte des Pflanzentopfs gemessen werden können.

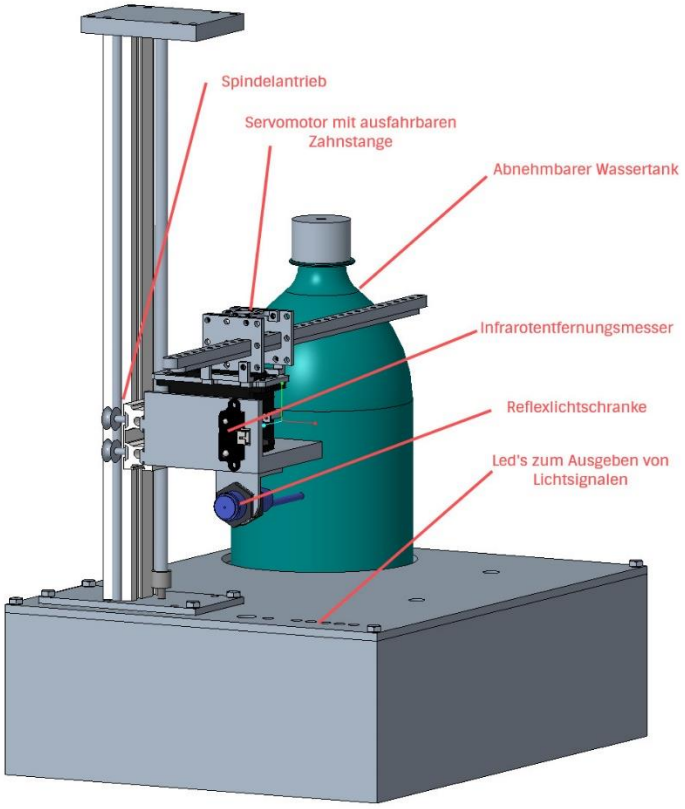

Abbildung 6: Äußere Ansicht Wasserversorgungsstation

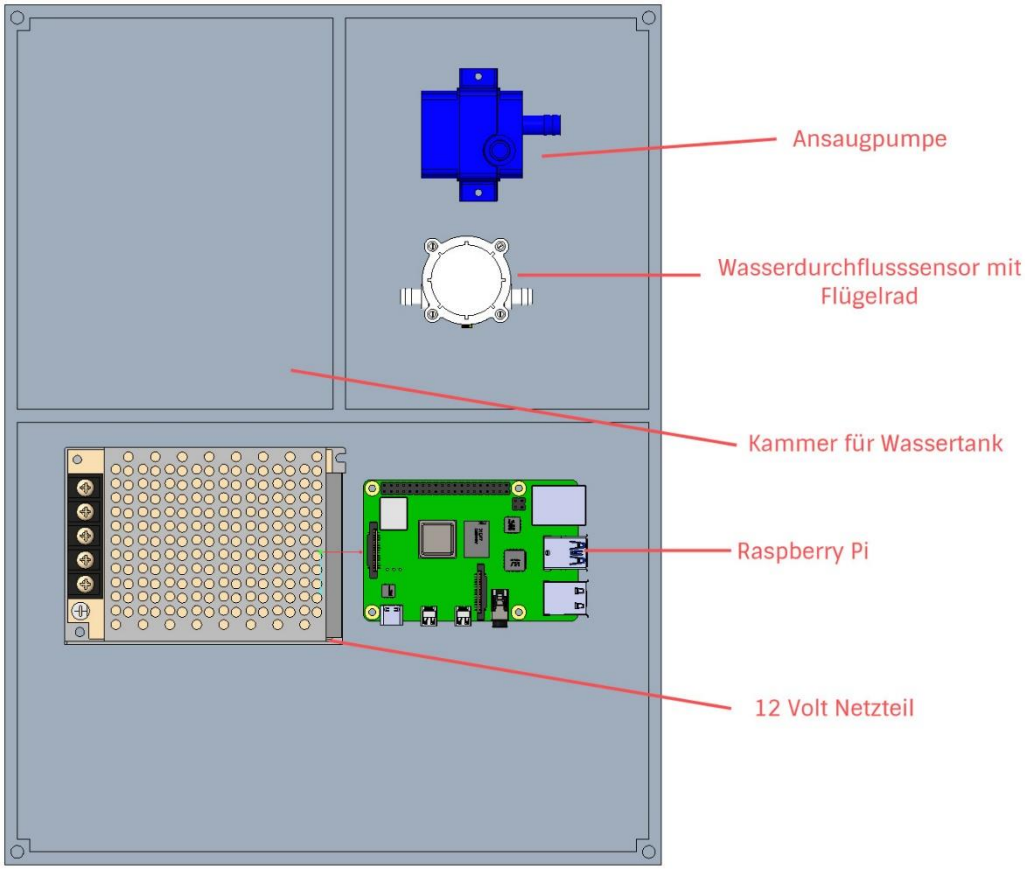

Abbildung 7: Innere Grundkonstruktion der Wasserversorgungsstation

3.8 Auswahl der elektrotechnischen Elemente

Das Zusammenspiel der Komponenten wird in Abbildung 8 dargestellt. Alle Elemente werden vom Raspberry Pi ausgewertet und gesteuert. Die Spannungsversorgung aller elektronischen Komponenten findet über ein 12-Volt-Netzteil statt, das mit der Spannungsversorgung des Zimmers verbunden ist. Über einen Spannungsverteiler wird jeder Sensor mit der benötigten Spannung versorgt. Wenn ein Sensor eine maximale Spannung von 5 Volt nutzen kann, muss die Spannung von 12 Volt auf die gewünschten 5 Volt gebracht werden. Dazu kann ein Spannungsregulator in Form eines L7805 verwendet werden, der dazu 5 A liefert. Diese Werte sind für alle 5 V Geräte ausreichend.

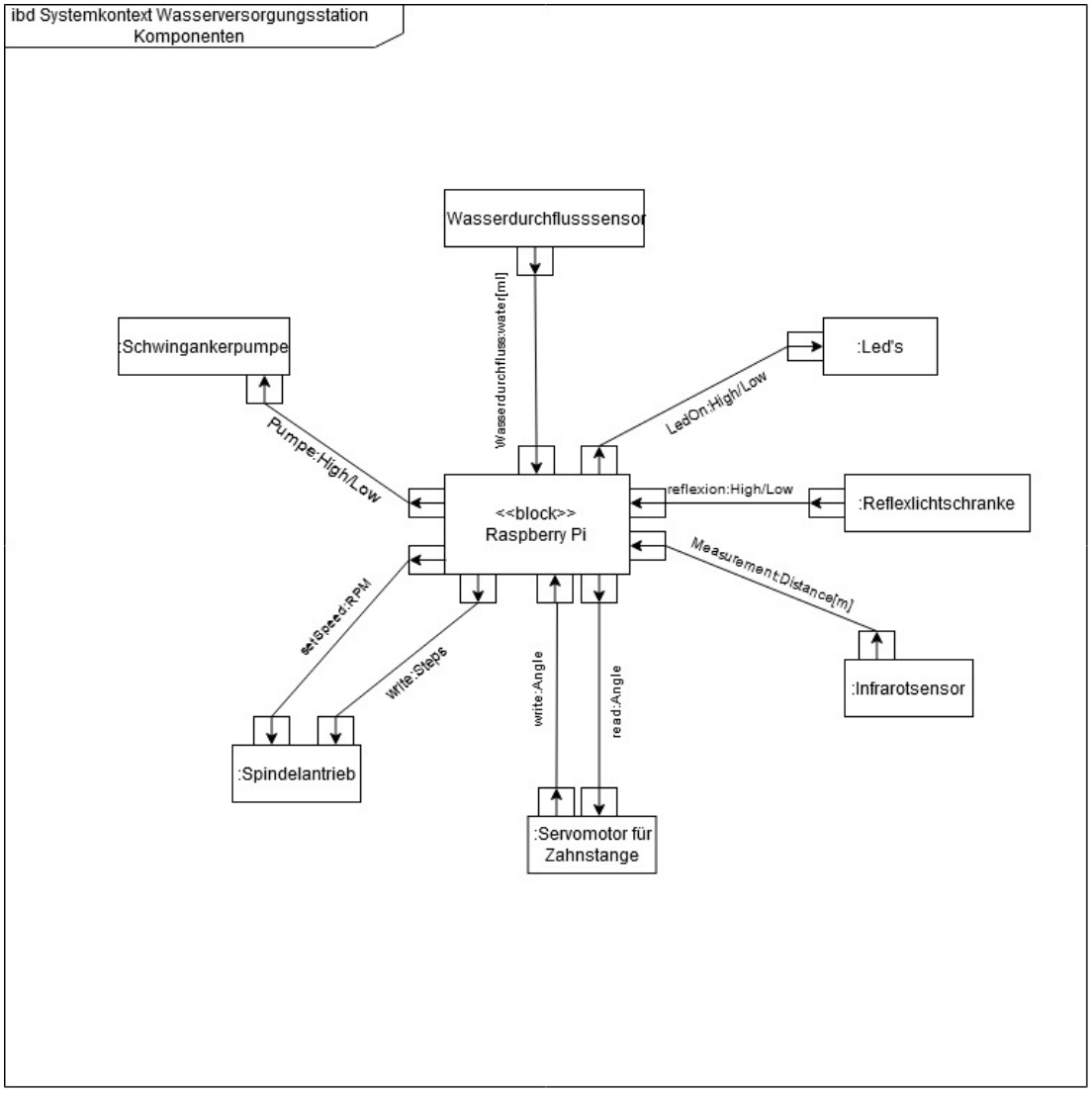

Abbildung 8: ibd Systemkontext der einzelnen Komponenten

3.8.1 Sensoren

Es werden mehrere Sensoren in der Wasserversorgungsstation verbaut, die für den Wassertransport und die Positionierung der Achsen zuständig sind. Es werden die wichtigsten Daten der Sensoren vorgestellt.

Infrarotsensor

Für die Messung der Distanz zum Pflanzentopf wird ein Infrarotsensor der Marke Sharp, Typ GP2Y0A41SK0F, verwendet, der in Abbildung 9 illustriert wird.
Die wichtigste Anforderung an den Sensor ist, dass der Infrarotsensor den kleinsten Abstand von 5 cm und den maximalen Abstand von 25 cm erkennen kann. Diese Anforderungen sind in den Requirements enthalten und müssen eingehalten werden. Die Größe hat keine Bedeutung, da die meisten Sensoren sehr klein gehalten sind.

Die wichtigsten Kenndaten aus dem Datenblatt [Ir18] sind:

- Typ: GP2Y0A41SK0F
- Betriebsspannung: 5 V/DC
- Ausgänge: Analogspannung
- minimale Reichweite: 4 cm
- maximale Reichweite: 30 cm
- min. Temperatur: -10 °C
- max. Temperatur: +60 °C

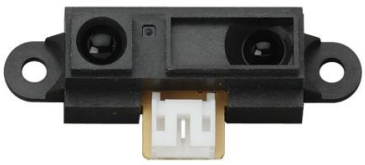

Abbildung 9: Infrarotsensor GP2Y0A41SK0F

Reflexlichtschranke

Die Reflexlichtschranke ist für die Erkennung des Pflanzentopfes zuständig. Sie soll anhand eines Reflexionsmarkers am oberen Teil des Pflanzentopfes erkennen, ob sich ein markierter Topf vor der Wasserversorgungsstation befindet.
Als Sensor wird eine Panasonic Reflexions-Lichtschranke CY-192B-P-Y-C verwendet, wie sie in Abbildung 10 dargestellt ist.

Abbildung 10: Panasonic Reflex-Lichtschranke CY-192B-P-Y-C

Die wichtigsten Kenndaten nach [Rfx20] sind:

- Typ: CY-192B-P-Y-C
- Betriebsspannung: 12 V
- Ausgänge: PNP
- max. Reichweite: 4 m
- Länge: 46 mm
- Durchmesser 18 mm

Mit Hilfe eines Montagewinkels wird der Sensor unter der beweglichen Plattform montiert, sodass der Infrarotsensor darüber angebracht werden kann. Damit ist gewährleistet, dass beide Sensoren die gleiche Fläche zum Messen anvisieren, wenn sie den Pflanzentopf anstrahlen.
Erkennt der Sensor eine Reflexion, wird der Ausgang zum Raspberry Pi geschlossen, sodass die 12 Volt anliegen. Es ist darauf zu achten, dass keine Spannung höher als 5 Volt an den Raspberry-Pi-Pin gelangt, da dieser beschädigt werden können. Aus diesem Grund muss eine Spannungsstabilisierung durchgeführt werden. Hierfür sollte eine Z-Diode mit einer Zenerspannung von $< 5\ V$ parallel integriert werden, sodass diese Spannung an der Z-Diode abfallen kann. An dem Raspberry-Pi-Pin liegt ebenfalls eine Spannung von < 5V an.

Wasserdurchflusssensor

Bei diesem Projekt wird Wasser genutzt, das ein im Vergleich zu anderen Flüssigkeiten einfach zu messendes Medium ist. Die Wassermenge muss nicht auf die Nachkommastelle genau gemessen werden, da eine Pflanze auch 10 ml mehr oder weniger vertragen kann. Somit ist die Auswahl auf einen Wasserflusssensor mit Flügelrad gefallen, wie er in Abbildung 11 zu sehen ist.

Die wichtigsten Kenndaten [Wfs20] lauten:

- Betriebsspannung: 5–12 V
- Typ: YF-S401
- Genauigkeit: ±5 %
- Maße: 58 x 35 x 26 mm
- maximaler Wasserdruck: 0.8 MPa
- Wasserdurchflussbereich: 0.3–6 L/Min

Abbildung 11: Wasserdurchflusssensor YF-S401

3.8.2 Aktoren

Die Wasserversorgungsstation besitzt vier Aktoren, die bei der Steuerung und Signalrückmeldung über LEDs zum Einsatz kommen. Im Folgenden werden die einzelnen Aufgaben und relevanten Kenndaten genannt.

LEDs

Die Wasserversorgungsstation soll fünf LEDs beinhalten, die dem Kunden den Wasserstand sowie den Zustand im Betrieb anzeigen soll. Sie werden auf dem Deckel montiert, sodass die Leuchtsignale leicht zu erkennen sind. Als Leuchtfarbe soll Rot verwendet werden, da es eine Signalfarbe ist, die vom Menschen schnell wahrgenommen wird. Für die Montage wird ein Stahlgehäuse mit Innenreflektor [Zle20] benutzt (siehe Abbildung 12), das in den Deckel der Wasserstation eingesetzt wird.

Abbildung 12: LED-Stahlgehäuse

Wasserpumpe

Die Wasserpumpe ist für den Transport des Wassers zuständig. Es ist wichtig, dass die Pumpe das Wasser ansaugen kann, da sie nicht im Wasservorrat platziert werden soll. Zum Einsatz kommt die Pumpe aus Abbildung 13.

Abbildung 13: Wasserpumpe

Ihre Kenndaten nach [Wap29] lauten:

- Betriebsspannung: 6–12 V
- Stromstärke: 0.5–0.7A
- maximale Durchflussmenge: 1–3 L/Min
- maximale Förderhöhe: 3 m
- maximale Ansaughöhe: 2 m
- Maße: 86 x 43 mm

Ein weiteres Kriterium für die Auswahl der Pumpe ist, dass sie mit den anderen Sensoren und Aktoren kompatibel sein muss. Speziell wird auf die Kompatibilität mit dem Wasserdurchflusssensor geachtet. Dieser sollte die Durchflussmenge der Pumpe erkennen können. Das ist mit 1 bis 3 L/min bei der Pumpe und mit 0.3 bis 6 L/min beim Sensor gewährleistet.

Servomotor für die Bewegung der Zahnstange

Der Servomotor für die ausfahrbare Achse ist von der Firma Tetrex produziert und in Abbildung 14 dargestellt.

Abbildung 14: Servomotor für die Zahnstangenbewegung

Die Kenndaten [Sec19] lauten:

- Typ: HS-785HB
- Betriebsspannung: 4.8–6 V
- Drehmoment: mit 4.8 V bei 108 Ncm
- Lagerart: Doppelt kugelgelagert
- Stellzeit bei 4.8 V: 1.68 s (60°)
- Servomotor-Technologie: analog

Der Motor ist für das Ausfahren der Zahnstange zuständig und besitzt als Aufsatz ein Zahnrad. Durch die maximale Anzahl der Umdrehungen des Servomotors kann die Zahnstange etwa 25 cm ausgefahren werden, was den Requirements entspricht und ein Ausfallen der Zahnstange verhindert. Der Motor, inklusive ausfahrbarer Zahnstange, ist als fertiges Set erhältlich. Er muss auf der Plattform fixiert werden, sodass keine unerwünschte Verschiebung stattfindet, wenn die Zahnstange ausgefahren wird.

Schrittmotor

Zum Bewegen der ersten Achse wird ein Schrittmotor benötigt, der über eine Wellenkupplung an die Trapezspindel montiert wird. Die Spindel ist für die Bewegung der Höhenachse zuständig und somit für das Bewegen der Plattform. Zum Einsatz kommt der Schrittmotor aus Abbildung 15. Dieser besitzt ein hohes Haltemoment, das für das Gegenmoment sorgt, sollte die Spindel durch äußere Einflüsse, wie das Gewicht der Plattform gedreht werden.

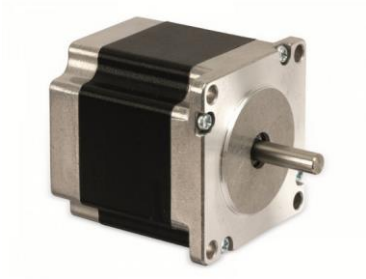

Abbildung 15: Schrittmotor ACT 23HM6620

Die Kenndaten aus dem Datenblatt [Sch19] lauten:

- Typ: ACT 23HM6620
- Phasen: 2 Phasen
- Strangspannung: 5,4 V
- Strangstrom 1,4 A/Phase
- Haltemoment: 1,1 Nm
- Strangwiderstand: 3,6 Ω/Phase
- Motormaße ohne Welle (B x H x T): 56,4 x 56,4x 56 mm

3.8.3 Leistungselektronik

Die Wasserversorgungsstation soll keine Batterie nutzen. Sie soll über das Hausstromnetz versorgt werden.

Schaltnetzteil

Die Hausspannung liefert einen maximalen Wert von 230 V. Die verwendeten Sensoren benötigen eine Spannung von fünf bis zwölf Volt. Aus diesem Grund wird ein Schaltnetzteil (siehe Abbildung 16) eingebaut, das die Spannung von 230 V auf zwölf Volt umwandelt. Die Spannung wird anschließend über einen Spannungsverteiler auf den genutzten Aktor oder Sensor geschaltet.

Die Kenndaten aus [Zzc27] lauten:

- Typ: LRS-150-12
- Eingangsspannung (max.): 264 V/AC
- Ausgangsspannung (max.): 12 V/DC
- Ausgangsstrom (max.): 12.5 A

Abbildung 16: Schaltnetzteil LRS-150-12

Steuergerät

Als Steuergerät wird ein Raspberry-Pi der vierten Generation verwendet. Dieser kann über Pins angeschlossene Sensoren und Aktoren schalten und die Daten verarbeiten. Außerdem bietet er eine Programmierungsoberfläche, auf die der Nutzer über einen Bildschirm zugreifen kann, um beispielsweise Veränderungen in der Programmierung vorzunehmen.

4 Simulation

Zum Lösen praktischer Probleme werden immer öfter Simulationen genutzt, die in 2D oder 3D betrachtet werden. Dabei handelt es sich um eine effiziente Methode, Systeme auf ihre Grenzen zu testen. Die digitale Umgebung bietet viele Möglichkeiten, Experimente durchzuführen, ohne ein reales Risiko einzugehen. Zudem werden Kosten sowohl auf materieller als auch personeller Ebene gespart und die zuständigen Ingenieure können, um Zeit zu sparen, ein vereinfachtes Systemmodell programmieren. Denn das Wichtigste sind die physikalischen Eigenschaften der Simulation. Sie bieten großen Spielraum beispielsweise für Gravitationswerte oder Kollisionseigenschaften von Modellen, wodurch das präzise Einstellen der physikalischen Werte für Anfänger häufig schwierig ist. Diese Genauigkeit der verschiedenen Werte bietet aber auch einen höheren Detailgrad, der zu präziseren Vorhersagen und Lösungen führt. Anders als die Analytik, die mit Solvern und generellen Lösungen für numerische Probleme arbeitet, kann die Simulation schnell und einfach ein Zustandsbild für die Beobachtung von Systemverhalten darstellen. Diese Visualisierung von Konzepten erleichtert das Überprüfen und Durchblicken des Systems, wodurch Ideen leichter mit anderen kommuniziert werden können. Dadurch bietet die Visualisierung ein Ergebnis, das dem Management und den Kunden anschaulich vorgestellt werden kann.

4.1 Simulationsumgebung

Für die Simulation der Wasserversorgungsstation wird das Simulationsprogramm Gazebo verwendet, das unter dem Betriebssystem Ubuntu 18.04 arbeitet. Gazebo ist für das Simulieren von Robotern entwickelt worden, die in geschlossenen, aber auch offenen Räumen arbeiten sollen. Durch mehrere Physik-Engines bietet die Software Einstellmöglichkeiten für die physikalischen Eigenschaften, wodurch eine Vielzahl von Experimenten möglich ist.

Zu den Physik-Engines gehören mitunter [Zga20]:

- Bullet, das für das Simulieren bei Kollision von starren oder auch deformierbaren Objekten genutzt wird,
- Open Dynamics Engine, das für das Simulieren der Körperdynamik verwendet wird,
- Simbody, das auf das Simulieren von menschlichen und Robotermechanismen spezialisiert ist.

Für das realistische Rendern der Robotermodelle und der Umgebung ist die Object-oriented Graphics Rendering Engine (OGRE) zuständig, die auch als Spiele-Engine für namenhafte Computerspiele genutzt wird.

Für das Programmieren der Roboter wird als Framework das Robot Operating System (ROS) verwendet. Das ROS-Projekt wurde an der Stanford University im Jahr 2007 entwickelt und gilt als eines der bekanntesten Roboter-Frameworks [Rog19]. Es ist eine Open-Source-Software, die die meisten Programmiersprachen unterstützt, die für das Implementieren von Robotern genutzt werden. Darunter fallen C++, Python und Lisp, die vollständig unterstützt werden, und C#, Java und Lua, die nur durch experimentelle Bibliotheken unterstützt werden. Des Weiteren bietet die Software ein Interface für die Kommunikation zwischen Programmen an. Dies ist besonders bedeutsam für den Austausch von Sensordaten, sodass eine realitätsnahe Hardware simuliert werden kann. Ebenfalls können durch die Kommunikation von Sensoren, Roboter in der Realität arbeiten und anhand ihrer Daten kann eine Simulation generiert werden. Unterstützt wird ROS durch zahlreiche Drittanbieter, die ihre Bibliotheken zur freien Nutzung zur Verfügung stellen. Als Beispiel kann die Bibliothek OpenCV genannt werden, die des Öfteren zum Auswerten und Manipulieren von Bilddaten genutzt wird. Ein weiteres interessantes Feature von ROS sind die integrierten Algorithmen. Dazu zählen Roboter-Algorithmen wie:

- Nutzung von Kontrollern mit PID-Eigenschaften,
- simulierte Lokalisation und Kartierung,
- Planen von Fahrrouten.

Zusammenfassend ist das ROS ein Kollektiv von Bibliotheken, Physik-Engines und anderen Werkzeugen, die das Programmieren und Simulieren für den Benutzer vereinfachen.

4.2 Robot-Operating-System-Pakete

Für die Installation des ROS ist die offizielle Hauptseite [Ros20] empfehlenswert. Hier werden verschiedene Versionen für unterschiedliche Linux-Distributionen angeboten. Für dieses Projekt wurde die Version *Kinetic* verwendet, die zu den meistgenutzten Versionen von ROS gehört und somit die ausgiebigsten Werkzeuge enthält.

4.2.1 Aufsetzen des Workspace

Für das Benutzen von ROS-Plugins und weiteren Erweiterungen wird ein sogenannter Workspace benötigt. Dieser wird als Ordner angelegt und beinhaltet alle neuen Projekte, die mit ROS arbeiten sollen. Für das Erstellen des Ordners kann die Kommandozeile verwendet werden. Mit dem folgenden Befehl wird ein Workspace mit dem Namen *catkin_ws* erstellt, der sich im Pfad der Kommandozeile befindet.

```
$ mkdir -p ~/catkin_ws/src
```

In diesem Ordner befindet sich nach dem Erstellen nur der weitere Ordner *src*, der als Hauptordner für das Programmieren der Simulation dient. Um den erstellten Ordner mit ROS zu verwenden, muss er mit ROS verbunden werden. Hierfür wird im Ordner folgender Befehl über die Kommandozeile ausgeführt.

```
$ catkin_make
```

Nach dem Ausführen sollte eine *CMakeLists.txt*-Datei in dem *src*-Ordner erstellt worden sein und außerdem die Ordner *build* und *devel*, die sich in *catkin_ws* befinden. Nachdem dieser Schritt ausgeführt worden ist, kann die *setup.bash*-Datei ausgeführt werden. Sie befindet sich im Unterordner *devel* und sorgt für das Hinzufügen verschiedener Umgebungsvariablen für ROS. Sie wird mit folgendem Befehl ausgeführt.

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

Die zuletzt genannten Befehle sind die wichtigsten für das Arbeiten mit ROS. Sie werden mit jeder Veränderung im Workspace angewandt, sodass die Umgebungsvariablen upgedatet bleiben. Oft ist dies eine Fehlerquelle für Error-Nachrichten beim Ausführen verschiedener Dateien.

Nachdem die vorherigen Schritte durchgeführt worden sind, kann die eigentliche Installation mit einem Überprüfen der ROS-Umgebungsvariablen durchgeführt werden.

```
$ echo $ROS_PACKAGE_PATH
```

Es sollte ein Pfad angegeben werden, der vergleichbar mit dem folgenden ist.

```
/home/username/catkin_ws/src:/opt/ros/kinetic/share
```

4.2.2 Aufbau des Workspace

Schnell kann die Ordnerstruktur beim Arbeiten mit dem Projekt unübersichtlich werden. Aus diesem Grund wird die Beispielstruktur für die Wasserversorgungsstation in Abbildung 17 dargestellt.

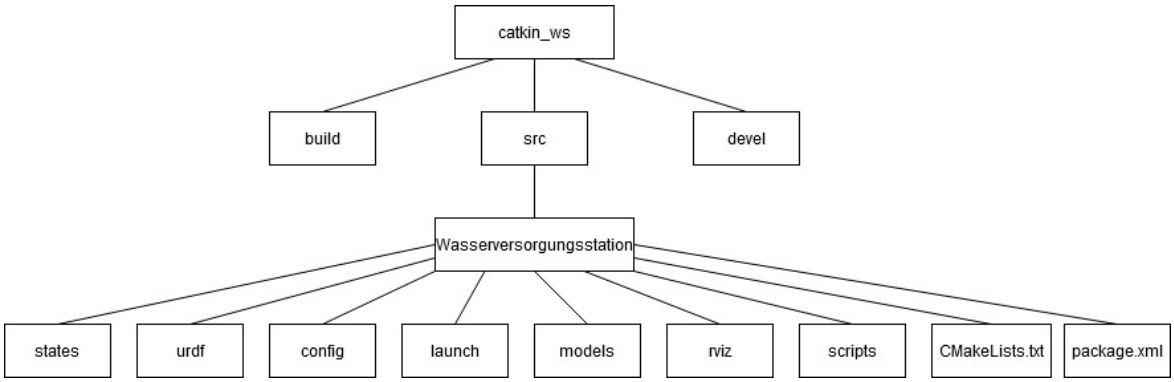

Abbildung 17: Ordnerstruktur des Wasserversorgungsprojekts

Nach dem Erstellen des *catkin_ws*-Ordners muss ein Paket in *src* erstellt werden. Dieses enthält alle Dateien, die für die Simulation und das Debuggen der Wasserversorgungsstation nötig sind. Das Paket dient als vollständiges Projekt der Wasserversorgungsstation. Mit dem nachfolgenden Befehl wird ein ROS-Paket erstellt.

```
$ catkin_create_pkg <Paketname> [Abhängigkeit 1] [Abhängigkeit 2]
```

Als Paketname muss ein noch nicht benutzter Ordnername verwendet werden, da sonst Fehler beim Ausführen des Pakets entstehen können. Nach der Auswahl eines Namens können Abhängigkeiten für das Paket hinzugefügt werden, wie das Einbinden von Python mit der Abhängigkeit *rospy*. Diese können auch nach dem Erstellen nochmals hinzugefügt werden. Nach dem Erstellen befinden sich im Paketordner eine weitere *CMakeLists.txt*- und eine *package.xml*-Datei. Weitere Unterordner können einfach erstellt oder eingefügt werden, um Dateien besser organisieren zu können. Beispielsweise befinden sich im selbst erstellten Ordner *models* alle verfügbaren CAD-Daten, die für die Simulation verwendet werden.

4.2.3 package.xml

Die *package.xml*-Datei enthält alle Abhängigkeiten des Pakets [Rob18]. Sie besitzt ein vordefiniertes Muster wie im Quellcode 1. Dieser kann erweitert werden, wodurch weitere Abhängigkeiten hinzugefügt werden.

Quellcode 1: Beispielcode der package.xml

```
<?xml version="1.0"?>
<package format="2">
   <name>wasserversorgungsstation_description</name>
   <version>0.0.0</version>
   <description>The wasserversorgungsstation_description
package</description>

   <maintainer email="user@todo.todo">user</maintainer>
   <license>TODO</license>

   <buildtool_depend>catkin</buildtool_depend>
   <build_depend>rospy</build_depend>
   <build_export_depend>rospy</build_export_depend>
   <exec_depend>rospy</exec_depend>
</package>
```

Für weitere Abhängigkeiten muss die Datei mit den Tags *build_depend*, *build_export_depend* und *exec_depend* erweitert werden, die den Abhängigkeitsnamen im Tag haben. Außerdem kann der Programmierer seine persönlichen Informationen für Fragen freigeben.

4.2.4 CMakeLists.txt

Die *CMakeLists.txt*-Datei ist für das Einbinden der Pakete in CMake zuständig [Rob18] [Ros20]. Ein Beispielmuster mit verschiedenen Abhängigkeiten kann im Quellcode 2 betrachtet werden.

Quellcode 2: Beispielcode einer CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0.2)
project(wasserversorgungsstation)

find_package(catkin REQUIRED COMPONENTS
  controller_manager
  joint_state_publisher
  robot_state_publisher
  rospy
  rviz
)

catkin_package(
)

include_directories(
  ${catkin_INCLUDE_DIRS}
)
```

Die Datei kann auf unterschiedliche Weise modifiziert werden, jedoch wird für dieses Projekt nur das Hinzufügen einer Abhängigkeit vorgestellt.
Dabei wird unter der Funktion *find_package()* der Abhängigkeitsname hinzugefügt, sodass dieser der Simulationsumwelt bekannt und für Suchende auffindbar ist.

4.3 ROS-Nodes und -Topics

Für die Datenübertragung beispielsweise von Sensorwerten arbeitet ROS mit sogenannten Nodes und Topics [Rob18] [Ros20]. Ein Node dient als Sender oder Empfänger von Nachrichten. Im Programm können mehrere dieser Nodes erstellt werden und sind aktiv, solange das Programm einen Prozess durchläuft. Die Nodes können nach Belieben verknüpft werden und über mehrere Programme miteinander kommunizieren. Dabei können verschiedene Datentypen übertragen werden. Die Wasserversorgungsstation arbeitet beispielsweise mit Distanzdaten, die einen

einzelnen Float-Wert weitergeben – anders als die Kamera, die auch ganze Kamerabilder in Arrays im RGB-Format versendet. Die Brücke für den Tausch bilden die Topics. Sie stellen die Verknüpfung zwischen den Nodes dar. Jedes Topic erhält einen Namen und kann von einem Node gezielt benutzt werden. Die Abbildung 18 zeigt die Möglichkeit für eine Übertragung. Node 1 publiziert (*publish*) ein Wert auf das Topic namens */topic* und Node 2 greift auf die Daten zu, indem er das Topic abonniert. Jede Veränderung durch Node 1 wird auf das Topic geschrieben und jede Veränderung wird von Node 2 bearbeitet. Dabei sollte eine möglichst gleiche Frequenz zwischen den Nodes herrschen, sodass die Werte sich gleichzeitig aktualisieren.

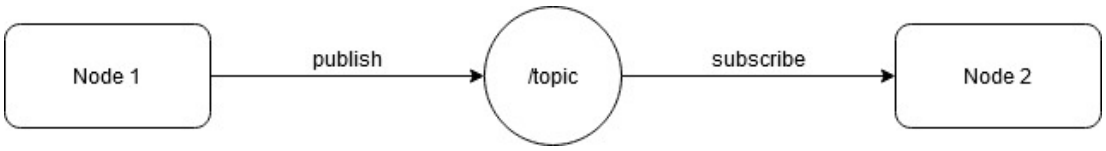

Abbildung 18: Zusammenspiel der Nodes

Die laufenden Topics und Nodes können mitverfolgt werden und über die Kommandozeile hat der Nutzer Zugriff auf die verschiedenen Werte. Die folgenden Befehle listen alle verfügbaren Topics und Nodes auf.

```
$ rostopic list
$ rosnode list
```

Mit dem Zusatz *info* können weitere Informationen zu speziellen Nodes oder Topics aufgerufen werden.

```
$ rostopic info <topicname>
$ rosnode info <nodeinfo>
```

Mit Hilfe der Info wird beispielsweise der Type des Topics gelistet. Dieser ist besonders wichtig, um auf das Topic zuzugreifen. Empfänger und Sender müssen dem Datentyp des Topics entsprechen. Mit der Info über einen Node werden alle Verbindungen zu Topics aufgelistet und es wird unter Publishing- und Subscribing-Topics unterschieden.

Die Wasserversorgungsstation besitzt mehrere Publisher und Subscriber. In Abbildung 19 können alle Nodes und Topics der Wasserversorgungsstation betrachtet werden. Die Ellipsen zeigen einen Node und dessen Namen an. Bei dem eckigen Rahmen handelt es sich um ein Topic. Jeder Node sollte mit mindestens einem Topic verbunden sein. Dabei zeigen die Pfeile die Richtung der Datenübertragung an. Beispielsweise ist */Publishertest.py* ein Publisher, der seine Werte auf das Topic */ZustandJetzt* schreibt. Anders ist es beim Node */Wasserdurchflusssensor.py*, der das Topic */ZustandJetzt* subskribiert (*subscribe*) und ebenfalls einen Publisher enthält, der seine Daten auf ein anderes Topic übermittelt. Ein Programm kann über mehrere Publisher und Subscriber verfügen. Für eine bessere Visualisierung der Zugehörigkeit von Topics können diese zu einer übergeordneten Gruppe hinzugefügt werden. Ein Beispiel dafür sind die Topics, die mit der Steuerung der Gelenke verbunden sind. Diese gehören zu der Überschrift *WVS*. Auffallend ist das Topic */wvs/controller_spawner*, da dieses mit keinem Node verbunden ist. Der Grund hierfür ist, dass manche Nodes automatisch mit dem Starten der Gelenkkontroller aktiviert werden und ein Signal für das erfolgreiche Ausführen ausgeben. Dieses Signal wird bei der Verarbeitung von Daten nicht benötigt, der Nutzer kann aber beim Aufrufen der Node-Liste erkennen, dass das Programm ausgeführt wird.

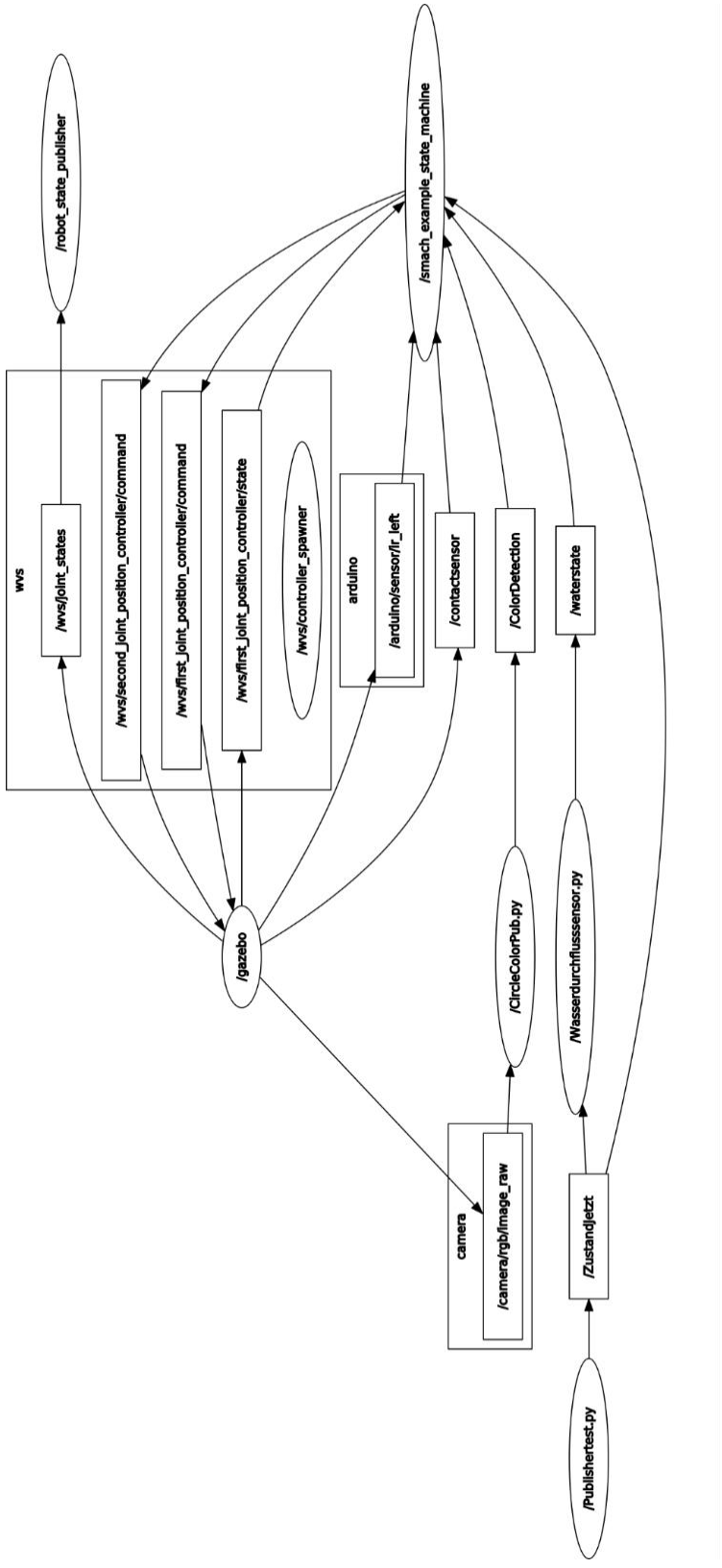

Abbildung 19: Alle Nodes und Topics der Wasserversorgungsstation

4.3.1 Publisher

Der nachfolgende Quellcode 3 zeigt ein Beispiel für die Programmierung eines Publishers, der einen Sender-Node enthält. Er dient als Musterbeispiel auf der offiziellen ROS-Seite [Ros20] und enthält alle nötigen Elemente für das Publizieren eines Strings auf ein Topic.

Quellcode 3: Beispielmuster eines Publishers

```
#!/usr/bin/env python

import rospy
import time
from std_msgs.msg import String

def Publizieren():
    pub = rospy.Publisher('topicname', String, queue_size=10)
    rospy.init_node('node1', anonymous=True)
    rate = rospy.Rate(10)
    while not rospy.is_shutdown():
        global start_str
        start_str = "start"
        pub.publish(start_str)
        rate.sleep()

if __name__ == '__main__':
    try:
        Publizieren()
    except rospy.ROSInterruptException:
        pass
```

Es folgt eine Aufbaubeschreibung, um die Programmierung verständlich zu machen und mit dem gesammelten Wissen zu erweitern.

```
#!/usr/bin/env python
```

Eine Python-Datei beginnt mit der soeben gelisteten Zeile, die gewährleistet, dass das Programm unter Python ausgeführt wird.

```
import rospy
import time
from std_msgs.msg import String
```

Die gängigsten Programmiersprachen für Nodes sind C++ und Python, die mit einer Bibliothek importiert werden müssen. In diesem Beispiel wird Python verwendet, wodurch die *rospy*-Bibliothek hinzugefügt werden muss. Der Publisher soll dem Topic einen String überliefern, sodass dieser Datentyp ebenfalls hinzugefügt werden muss. Hierbei wird die Bibliothek *std_msgs* benutzt, welche eine der wichtigsten Bibliotheken ist.

```
pub = rospy.Publisher('topicname', String, queue_size=10)
rospy.init_node('node1', anonymous=True)
rate = rospy.Rate(10)
```

Die Variable *pub* bildet das Topic, das den Namen *topicname* und den Type String besitzt. Die *queue_size* definiert die Buffer-Größe für Nachrichten, die noch nicht empfangen worden sind. Sie werden gespeichert und weitergegeben, wenn der Subscriber diese entgegennehmen kann. Mit *rospy.init_node* wird der Node mit dem Namen *node1* initialisiert. Jedes Topic benötigt einen Node, der in derselben Datei definiert werden muss. Die *rospy.rate* bestimmt die Übertragungsrate des Topics. Diese ist beim vorliegenden Beispiel auf 10 Hz gesetzt, was vollkommen ausreichend für eine Sensordatenübertragung ist.

```
while not rospy.is_shutdown():
  global start_str
  start_str = "start"
  pub.publish(start_str)
  rate.sleep()
```

Die Implementierung des Node und des Topics findet in einer Funktion namens *Publizieren()* statt. Diese wird so lange ausgeführt, bis der Prozess durch den Nutzer gestoppt wird. Dies kann einfach über das Drücken von Ctrl+C erfolgen, wodurch die *is_shutdowm()*-Funktion eingeleitet wird. Erst mit dem Ausführen des *.publish*-Befehls auf die Variable *pub* kann der *start_str* auf das Topic publiziert werden. Mit der *rate.sleep()*-Funktion wird das Publizieren eingeschränkt, sodass die gewünschte Rate von 10 Hz ermöglicht wird.

```
if __name__ == '__main__':
    try:
        Publizieren()
    except rospy.ROSInterruptException:
        pass
```

Die fertige Funktion muss im Hauptprogramm, der sogenannten Main, aufgerufen werden, damit diese beim Starten der Datei ausgeführt wird. Dieser Prozess kann nur durch Beenden der Datei gestoppt werden.

4.3.2 Subscriber

Zum Empfangen der Werte des Publishers wird ein Subscriber benötigt, die großen Ähnlichkeiten mit dem Publisher hat. Der Quellcode 4 wird auch als Musterbeispiel auf der offiziellen ROS-Seite [Ros20] gezeigt.

Quellcode 4: Mustercode für einen Subscriber

```
#!/usr/bin/env python
import rospy
import time
from std_msgs.msg import String
def listener():
    rospy.init_node('Node 2', anonymous=True)
    rospy.Subscriber("topicname", String, callback)
    rospy.spin()
def callback(data):
    rospy.loginfo(recieved)
if __name__ == '__main__':
    listener()
```

Zuerst werden die benötigten Bibliotheken hinzugefügt und besonders muss auf die Kompatibilität mit der Bibliothek der *std_msgs* geachtet werden. Diese muss dieselbe sein wie in dem Publisher, der mit diesem Subscriber verbunden wird.

```
def listener():
    rospy.init_node('Node 2', anonymous=True)
    rospy.Subscriber("topicname", String, callback)
    rospy.spin()

def callback(data):
    rospy.loginfo(recieved)
```

Wie beim Publisher wird das Initialisieren des Node und des Topics in einer Funktion realisiert, die in diesem Beispiel *listener* genannt wird. Beim Starten der Pythondatei wird der Node ebenso lange ausgeführt. Damit der Node durch das Warten auf die Topic-Werte nicht beendet wird, muss die *rospy.spin()*-Funktion benutzt werden. Sie verhindert das Beenden des Node. Anders als beim Publisher verarbeitet ein Subscriber den Topic-Wert. Anhand einer *Callback*-Funktion kann dieser manipuliert oder in derselben Datei weitergegeben werden. Die *Callback*-Funktion wird mit jeder Veränderung des Topics erneut ausgeführt.

4.4 URDF und Xacro

Für die visuelle Darstellung von Modellen in der Gazebo-Simulation müssen diese anhand einer URDF- (Unified Robotic Description Format) oder einer Xacro-Datei vorbereitet werden. Beide Dateitypen sind sehr ähnlich in der Programmierung, da sie mit der Extensible Markup Language (XML) geschrieben werden. Der Begriff Xacro leitet sich aus XMLMacro ab und stellt eine vereinfachte Version von URDF dar. Das Programmieren von Xacro-Dateien bietet eine Modularität für alle Komponenten des Roboters. Aus diesem Grund werden die folgenden Kapitel für die Programmierung mit Xacro beschrieben.

4.4.1 Links

Ein Xacro-Programm besteht hauptsächlich aus Links, die jeweils eine Komponente des Roboters darstellen. Beispielsweise könnte ein mechanischer Arm aus den Links Oberarm, Unterarm und Hand dargestellt werden. Ein Link besitzt mehrere Eigenschaften, die vor dem Ausführen bestimmt werden müssen.
Im Quellcode 5 wird ein Beispiel für einen Link-Tag gezeigt. Hierbei umfasst der Tag mehrere Werte, die die Trägheit, die Kollision und die Visualisierung in der Simulationswelt näher beschreiben [Lix20].

Quellcode 5: Linkbeschreibung in einer Xacro-Datei

```
<link name="GazeboBaseLink">

  <inertial>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <mass value="50" />
    <inertia ixx="0.146" ixy="0.0" ixz="0.0" iyy="0.121"          iyz="0.0" izz="0.110"/>
  </inertial>

  <collision>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <mesh
filename="package://wasserversorgungsstation_description/models/meshes/GazeboBaseLin
k.dae"/>
    </geometry>
  </collision>

  <visual>
    <geometry>
      <mesh
filename="package://wasserversorgungsstation_description/models/meshes/GazeboBaseLin
k.dae"/>
    </geometry>
    <material name="blue">
    </material>
  </visual>
  </link>
```

Trägheit (*inertial*)

Unter der Trägheit muss das Gewicht der Komponente in Kilogramm und das Trägheitsmoment für jede Ausrichtung der Achsen angeben werden. Dies kann besonders schwierig werden bei Komponenten, die aus mehreren Formen bestehen. Die meisten CAD-Programme enthalten Erweiterungen, die diese Eigenschaften selbstständig berechnen und in einer Trägheitsmatrix ausgeben. Diese Werte können für die Beschreibung des Links in Gazebo übernommen werden.

Das Gewicht muss ebenfalls eingegeben werden. Im hier benutzten CAD-Programm Creo Parametric kann diese Eigenschaft mit Hilfe der Auswahl eines Materials bestimmt werden. Des Weiteren kann die Trägheit ihren Ursprungort im *origin*-Tag verschieben und drehen, sollte dies erwünscht sein.

Kollision (*collision*)

In der Simulationswelt kann die Komponente mit der Umwelt kollidieren. Der *mesh*-Tag bestimmt die Form des Bereichs in Gazebo, der kollidiert. Dieser ist bei der Simulation nicht zu sehen und sollte mit der *mesh*-Angabe in der Visualisierung übereinstimmen. Somit ist gewährleistet, dass beispielsweise eine Kugel keinen Kollisionsbereich wie eine Box hat. Alle Angaben können vom Ursprungsort aus verschoben und gedreht werden.

Visualisierung (*visual*)

Die Angaben für die Visualisierung bestimmen die Form, die in der Simulation zu sehen ist. Unter *mesh* kann eine CAD-Datei oder eine Grundform angegeben werden. Sie sollte so gewählt werden, wie in der Kollisionsangabe. Des Weiteren kann ein Material angegeben werden. Das Material bestimmt nicht die Dichte oder das Gewicht, sondern die Farbe in der Simulation. Ohne eine Angabe wird das Aussehen aus der CAD-Datei übernommen.

Quellcode 6: Einstellung der Koeffizienten für die Links

```
<gazebo reference="GazeboBaseLink">
   <!--stiffnes and damping for body contacts-->
   <kp>1000000.0</kp>
   <kd>1000000.0</kd>
   <!--Friction coefficients -->
   <mu1>100</mu1>
   <mu2>100</mu2>
</gazebo>
```

Die weiteren Einstellungen aus Quellcode 6 werden nicht in der Link-Beschreibung vorgenommen. Die Beschreibung findet in einem Gazebo-Tag statt, der auf den Namen des Links referenziert. Es werden der kp- und der kd-Wert angegeben, die den Steifheitsfaktor und den Dämpfungsfaktor für das Material beschreiben, sollte eine Kollision entstehen. Außerdem werden die Reibungskoeffizienten mu1 und mu2 angegeben. Der Wert mu1 beschreibt die Coulombsche Reibung, die auch als Festkörperreibung beschrieben wird. Sollten zwei Flächen aufeinandertreffen, wird dieser Wert als Reibungskoeffizient gewählt. Gazebo wählt dabei den kleinsten Wert der beiden Flächen aus. Für die zweite Reibungsrichtung wird mu2 bestimmt, die senkrecht zur ersten Reibungsrichtung wirkt.

4.4.2 Joints

Für das Bewegen von Links werden Gelenke (*Joints*) benötigt. Gazebo besitzt eine integrierte Joint-Bibliothek, die viele Bewegungsmöglichkeiten anbietet. Jedes Gelenk wird als eigenes Tag programmiert (vgl. Quellcode 7).

Quellcode 7: Aufbau eines Joint-Tags

```
<joint name="first_axis_joint" type="prismatic">
   <parent link="GazeboBaseLink"/>
   <child link="GazeboAchse1Plattform"/>
   <origin xyz="-0.005 0.0845 0.188" rpy="0 0 0"/>
   <limit lower="0" upper="0.21" effort="0.3" velocity="0.5"/>
   <axis xyz="0 0 1"/>
</joint>
```

Jedes Tag besitzt einen eigenen Namen, der einmalig ist. Dieser ist für die nachfolgende Steuerung und für die Durchsichtigkeit bei mehreren Gelenken wichtig. Jeder Joint besitzt einen Typ, der bestimmt, um welches Gelenk es sich handelt.

Die verwendeten Gelenke nach [Joi20] sind:

Prismatic Joint

Der Prismatic Joint vollzieht keine Drehung, sondern verschiebt ein verbundenes Element in eine gewünschte Achsenrichtung. Es bewegt sich ähnlich wie ein Hydraulikzylinder und besitzt einen Anfangspunkt und eine maximale Ausrichtung. Jeder Punkt dazwischen kann angefahren werden.

Fixed Joint

Ein Fixed Joint erlaubt das Fixieren einer Komponente an einem anderen Element. Beim Roboterzusammenbau wird das Gelenk hauptsächlich zum Anbringen nicht beweglicher Teile benutzt. Im Projekt wurde beispielsweise die Kamera an die Stelle der Reflexlichtschranke angebracht.

Ein Gelenk hält immer zwei Komponenten zusammen, wie nach [Joi20] und in Abbildung 20 zu sehen ist. Diese Verknüpfung wird anhand eines Parent und eines Child realisiert. Der Parent gilt als Ausgangskomponente und das Child als das zu bewegende Teil. Die Platzierung des Joints findet im *origin*-Tag statt, wo die Lage auf der x-, y- und z-Achse und eine mögliche Drehung um eine Achse angegeben wird. Das Koordinatensystem für die Bewegung ist das Koordinatensystem des Parent.

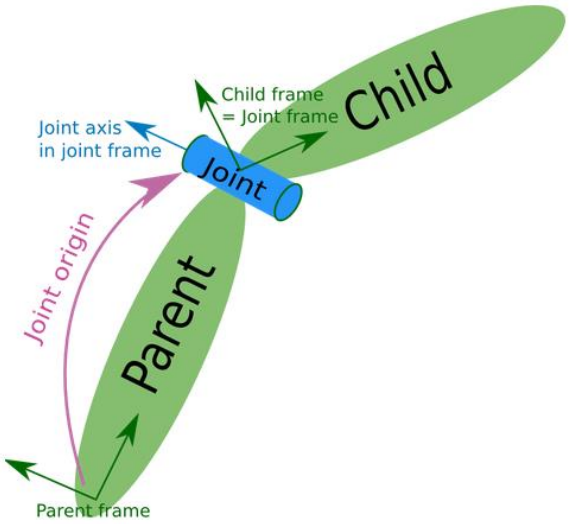

Abbildung 20: Zusammenbau einer Jointverbindung

Im *limit*-Tag wird die Bewegung eingeschränkt. Anhand der *lower*- und *upper*-Werte kann eine maximale und minimale Auslenkung in Metern eingestellt werden. Ebenfalls bestimmen diese Werte auch die maximalen Auslenkungen bei der Steuerung. Mit Hilfe des *velocity*- und der *effort*-Tags werden Schnelligkeit und Kraft des Gelenks ausgewählt. Die Kraft wird in Newton angegeben und die Geschwindigkeit in Metern pro Sekunde. Jedes Gelenk arbeitet auf einer Achse. Dies wird im *axis*-Tag bestimmt.

Die Wasserversorgungsstation besteht hautsächlich aus Fixed und Prismatic Joints. Die Komponenten der Sensoren werden an der Wasserversorgungsstation fixiert und die Achsen werden über die Prismatic Joints in der z- und x-Achse bewegt.

4.4.3 Plugins

Gazebo erlaubt das Integrieren verschiedener Sensoren und Aktoren mit Hilfe von Plugins, die in die URDF- oder Xacro-Datei eingefügt werden. Die meisten werden von anderen Nutzern entworfen, sodass die verfügbare Anzahl im World Wide Web sehr groß ist. Es werden nur die Plugins vorgestellt, die in der Wasserversorgungsstation integriert sind. Sie befinden sich in einer eigenen Xacro-Datei, auf die über die Hauptdatei zugegriffen werden kann.

Controller-Plugin

Das Controller-Plugin dient zur Manipulation der Joints (siehe Quellcode 8). Es wird mit Hilfe eines *Plugin*-Tags sowie mit einem *Transmission*-Tag integriert. Die Internetseite [Rco16] beinhaltet weitere relevante Informationen.

Quellcode 8: Controller-Plugin

```
<gazebo>
  <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
    <robotNamespace>/wvs</robotNamespace>
  <robotSimType>gazebo_ros_control/DefaultRobotHWSim</robotSimType>
  </plugin>
</gazebo>
```

Für das erfolgreiche Hinzufügen dieses Plugins müssen der Name *gazebo_ros_controll* und die Speicherdatei *libgazebo_ros_control.so* im *Plugin*-Tag angegeben werden. Die Bibliothek verfügt über mehrere Funktionen. Um die Gelenke anzusteuern, muss im *robotSimtype*-Tag die Klasse *DefaultRobotHWSim* ausgewählt werden.

Jedes Plugin arbeitet mit Nodes und Topics, wodurch die Manipulation während der Simulation ermöglicht wird. Beim Ausführen der Plugin-Datei werden verschiedene Nodes erstellt. Dabei bekommt jedes Topic ein Präfix im Namen, das hier */wvs* lautet, da es im *robotNamespace* ausgewählt worden ist. So nennt sich beispielsweise das Topic für die Steuerung des ersten Gelenks in der Simulation */wvs/first_joint/command*. Es empfiehlt sich, den Robotернamen als *roboternamespace* zu wählen, um bei Verwendung weiterer Roboter den Überblick zu behalten.

Jedes Gelenk benötigt eine eigene Beschreibung der Übertragung (*transmission*). Diese wird in einem *transmission*-Tag (vgl. Quellcode 9) programmiert.

Quellcode 9: Transmission-Tag in der Xacro-Datei

```
<transmission name="first_axis_joint_trans">
   <type>transmission_interface/SimpleTransmission</type>
   <joint name="first_axis_joint">
     <hardwareInterface>EffortJointInterface</hardwareInterface>
   </joint>
   <actuator name="first_axis_jointMotor">
    <hardwareInterface>EffortJointInterface</hardwareInterface>
     <mechanicalReduction>1</mechanicalReduction>
   </actuator>
 </transmission>
```

Jedes *Transmission*-Tag besitzt einen *joint*-Tag und einen *actuator*-Tag. Diese sind für die Steuerung des Gelenks zuständig und benötigen einen Typ, der Zugriff auf verschiedene Interfaces bietet. Ein Interface steuert das Gelenk mit beliebigen Werten, die durch die Auswahl des Interfaces bestimmt werden. Die verfügbaren Interfaces sind:

Position controller interface

Beim Anfahren einer gewissen Position kann die gewünschte Lage über x-, y- und z-Variablen angegeben werden. Dieses Interface fährt das Gelenk auf die gewünschte Position und ist durch die maximalen Bewegungen des Gelenkes limitiert. Dies wird bei den Joint-Eigenschaften eingestellt.

Velocity controller interface

Wenn die Schnelligkeit eine übergeordnete Rolle spielt, wird ein Velocity-Controller-Interface genutzt. Dieses erwartet Werte für die Geschwindigkeit in Metern pro Sekunde. Besonders interessant ist es für ein Continious Joint, da hiermit eine kontinuierliche Drehung erreicht werden kann, die durch die Geschwindigkeit reguliert wird. Bei anderen Gelenken wird die maximale zu erreichende Lage in dieser Geschwindigkeit angefahren.

Joint state controller interface

Dieses Interface dient als Publisher und informiert über den Zustand des Gelenkes, das in einem Topic publiziert (*published*) wird.

Effort controller interface

Das Effort-Controller-Interface besitzt einen Wert für die Position, die Geschwindigkeit und die Kraft des Gelenkes. Es ist für alle Gelenke geeignet und ermöglicht ein gutes Testen der Gelenkbewegung anhand verschiedener Werte.

Ähnlich wie bei Zahnrädern kann ein Übertragungswert unter *mechanicalReduction* angegeben werden, der das Übersetzungsverhältnis zwischen dem Motor und dem Gelenk bestimmt.

Quellcode 10: yaml-Datei der Wasserversorgungsstation

```yaml
wvs:
  joint_state_controller:
    type: joint_state_controller/JointStateController
    publish_rate: 1000
  first_joint_position_controller:
    type: effort_controllers/JointPositionController
    joint: first_axis_joint
    pid: {p: 10.0, i: 0.0, d: 5.0}
```

Für das Benutzen der Steuerung benötigt jeder Controller eine yaml-Datei, die im Quellcode 10 vorgestellt wird. Die Datei dient im Speziellen dem Einstellen eines PID-Reglers für das jeweilige Gelenk und für das Ausgeben der Jointzustände in einem Topic.

Im obigen Beispiel wird ein Kontroller für die Wasserversorgungsstation beschrieben. Dabei wird der Block *wvs* erstellt, der dem Namen des Projektes entspricht. In diesem Block werden zwei Kontroller beschrieben:

Joint_state_controller

Dieser Kontroller ist für das Publishen der Zustände verantwortlich. Er besitzt als Type den *Joint_state_controller*, der vergleichbar ist mit dem *Joint state controller*-Interface. Die einzige relevante Einstellmöglichkeit ist die Rate beim Publishen von Zustandsinformationen, die hier auf 1000 Hz gesetzt worden ist.

First_joint_position_controller

Dieser Kontroller beschreibt die PID-Werte für die erste Achsenbewegung. Er besitzt ebenfalls einen Typ, der dem *Effort_controller* und einem Gelenk (Joint) entspricht, das dem Kontroller zugeordnet wird. Hier wird der *first_axis_joint* zugeordnet, der in der Xacro-Datei im Transmission-Tag erstellt worden ist. Dieser enthält einen integrierten PID-Regler, der für die Regelung der Bewegung zuständig ist. Dadurch können die Reaktionszeit, die Regelabweichung und die Nachregelung für persönliche Wünsche justiert werden.

Kontaktsensor

Bei der Wasserversorgung wird beim Bewässern die zweite Achse zum Pflanzentopf bewegt. Beim Anfahren könnte die Pflanze beschädigt oder der ganze Pflanzentopf umgestoßen werden. Um dies zu verhindern, wurde nachträglich ein Kontaktsensor ans Ende der beweglichen Zahnstange angebracht. Dabei fährt die Zahnstange nicht schlagartig das Ziel an, sondern tastet sich an die gewünschte Position heran. Bei Kontakt wird der Sensor aktiviert und beobachtet die darauf angewendete Kraft. Übersteigt diese Kraft einen gewissen Wert, löst der Sensor aus und bricht die Bewässerung ab.

Quellcode 11: Kontaktsensor-Plugincode

```
<gazebo reference="contactsensor_link">
  <sensor name="contactsensor_sensor" type="contact">
    <always_on>true</always_on>
    <update_rate>5</update_rate>
    <contact>
<collision>GazeboAchse2Zahnstange_fixed_joint_lump__contactsensor_link_collision_1</col
lision>
    </contact>
    <plugin name="gazebo_ros_bumper_controller" filename="libgazebo_ros_bumper.so">
      <bumperTopicName>contactsensor</bumperTopicName>
      <frameName>contactsensor_link</frameName>
    </plugin>
  </sensor>
 </gazebo>
```

Das Plugin (siehe Quellcode 11) wird in der Xacro-Datei integriert und eingestellt. Im *Gazebo*-Tag benötigt dieser Sensor eine Referenz, die auf einen Link in der Xacro-Datei verweist. Dieser Link wird als Bauteilform des Sensors benutzt und erhält die Fähigkeit, Kontakt zu detektieren. In diesem Beispiel besitzt der Link den Referenznamen *contactsensor_link*. Der Sensor erhält ebenfalls einen Namen und einen Type. Der Type ist einfach zu wählen, da er dem Auslöser des Sensors entspricht. Der Nutzer kann die Rate und die Aktivierung des Sensors bestimmen. Hier entspricht die Rate 5 Hz und der Sensor ist immer aktiviert. Wie bei jedem Sensor in Gazebo werden die Sensordaten auf ein Topic geschrieben, das publiziert wird. Hier

kann der Name des Topics unter dem Tag *bumperTopicname* gewählt werden. Damit das Plugin funktioniert, benötigt es eine Bibliothek, die im Tag *plugin* angegeben wird. Für die Ausgabe von Werten muss der Joint beschrieben werden, der den Kontaktsensor an der Wasserversorgungsstation fixiert. Hierbei ist der Joint eine Fixierung an die Zahnstange, die den Namen *GazeboAchse2Zahnstange* trägt. Wird dies im *collision*-Tag angegeben, kann der Sensor die richtigen Werte anzeigen und die Kollision mit der Zahnstange wird ignoriert.

Infrarotsensor-Plugin

Der Infrarotsensor dient zur Feststellung der Entfernung von der Wasserversorgungsstation bis zum Pflanzentopf. Alle Beschreibungen der Bibliothek sind in [Irs18] zu finden. Er wird über den *Gazebo*-Tag auf den Link referenziert, der die Bauform des Sensors sein wird. Der Type des Sensors ist *ray* und der Name lautet *ir_leftabc (vgl. Quellcode 12)*.

Quellcode 12: Infrarot-Plugin für die Xacro-Datei

```
<gazebo reference="lrlink">
  <sensor type="ray" name="ir_leftabc">
    <pose>0 0 0 0 0 0</pose>
    <update_rate>50</update_rate>
    <ray>
      <scan>
        <horizontal>
        <samples>1</samples>
        <resolution>1.0</resolution>
        <min_angle>-0.01</min_angle>
        <max_angle>0.01</max_angle>
        </horizontal>
        <vertical>
        <samples>1</samples>
        <resolution>1</resolution>
        <min_angle>-0.01</min_angle>
        <max_angle>0.01</max_angle>
        </vertical>
      </scan>
```

```
        <range>
          <min>0.01</min>
          <max>3.75</max>
          <resolution>0.02</resolution>
        </range>
      </ray>
    <plugin filename="libgazebo_ros_range.so" name="gazebo_ros_range">
      <gaussianNoise>0.005</gaussianNoise>
      <alwaysOn>true</alwaysOn>
      <updateRate>5</updateRate>
      <topicName>/arduino/sensor/ir_left</topicName>
      <frameName>Irlink</frameName>
      <visualize>true</visualize>
      <radiation>infrared</radiation>
      <fov>0.02</fov>
    </plugin>
    </sensor>
</gazebo>
```

Die Updaterate kann in Hertz angegeben werden und beträgt hier 50 Hz.
Des Weiteren verfügt der Nutzer über verschiedene Einstellmöglichkeiten, um den Sensor realitätsnah abzubilden. Dabei können die Kegelmaße für den Laser verändert werden, die hier in Grad angegeben sind. Da es sich um eine Simulation handelt, können die Werte frei gewählt werden und unendlich groß sein. Für die minimale und maximale Entfernung des Sensors sollten somit Werte gewählt werden, die dem Lasersensor entsprechen. Der Laser wird ein optimales Laserbild ausgeben, wenn keine *gaussianNoise*-Angabe gewählt wird. Diese sorgt für kleinere Unregelmäßigkeiten in den Laserbildern, die auch in der Realität vorkommen.

Kamera-Plugin

Das Kamera-Plugin dient als Reflexlichtschranke in Gazebo, da dieser Sensor nicht verfügbar ist. Aus diesem Grund musste ein vergleichbarer Sensor programmiert werden, der ähnliche Eigenschaften besitzt. Dazu dient eine RGB-Kamera, die ähnlich wie die Reflexlichtschranke ein gewisses Muster oder eine Farbe erkennen kann. Bei

der Auswahl des Plugins muss beachtet werden, dass die richtige Kamera gewählt wird. Gazebo verfügt über eine Vielzahl an Kamera-Plugins, die anhand ihrer Namen schwer zu unterscheiden sind. Die Wasserversorgungsstation arbeitet mit einer Kamera, die vergleichbar ist mit derjenigen der Xbox Kinect [Kin17].

Quellcode 13: Kamera-Plugincode für die Xacro-Datei

```
<gazebo reference="cameralink">
  <sensor type="depth" name="camera">
  <always_on>true</always_on>
  <update_rate>20.0</update_rate>
  <camera>
    <horizontal_fov>${60.0*3/180.0}</horizontal_fov>
    <image>
      <format>B8G8R8</format>
      <width>640</width>
      <height>480</height>
    </image>
    <clip>
      <near>0.05</near>
      <far>8.0</far>
    </clip>
    </camera>
  <plugin name="kinect_camera_controller" filename="libgazebo_ros_openni_kinect.so">
    <cameraName>camera</cameraName>
    <alwaysOn>true</alwaysOn>
    <updateRate>10</updateRate>
    <imageTopicName>rgb/image_raw</imageTopicName>
    <depthImageTopicName>depth/image_raw</depthImageTopicName>
    <pointCloudTopicName>depth/points</pointCloudTopicName>
    <cameraInfoTopicName>rgb/camera_info</cameraInfoTopicName>

<depthImageCameraInfoTopicName>depth/camera_info</depthImageCameraInfoTopicName>
    <frameName>camera</frameName>
    <baseline>0.1</baseline>
    <distortion_k1>0.0</distortion_k1>
    <distortion_k2>0.0</distortion_k2>
    <distortion_k3>0.0</distortion_k3>
```

```
      <distortion_t1>0.0</distortion_t1>
      <distortion_t2>0.0</distortion_t2>
      <pointCloudCutoff>0.4</pointCloudCutoff>
    </plugin>
    </sensor>
</gazebo>
```

Das Kamera-Plugin wird im Quellcode 13 angegeben. Es wird ähnlich wie jedes andere Plugin über einen Pluginnamen eingefügt und an einen Link der Xacro befestigt. Hier wurde ein Kameralink verwendet, der sich auf der Reflexlichtschranke befindet. Diese Lage wurde ausgewählt, da sie die Reflexlichtschrankenfunktion nachahmen soll. Sie besitzt den Type *depth*, da die Kinect-Kamera ebenfalls die Entfernung zu anderen Objekten anhand einer Punktwolke ausgeben kann. Diese Funktion wird jedoch nicht verwendet. Die Kamera besitzt viele Einstellmöglichkeiten für das Bild, die bei Verwendung einer realen Kamera übernommen werden könnten. Die meisten Standardeinstellungen wurden nicht verändert, da die Kamera eine Reflexlichtschranke simulieren soll. Die einzige relevante Kamerainformation ist, dass es sich um eine RGB-Kamera handelt. Dadurch wird eine größere Datenmenge übertragen als bei einer Schwarz-Weiß-Kamera. Um diese Daten zu minimieren und die verfügbaren Ressourcen nicht zu strapazieren, wurde die Auflösung auf kleinere Einstellungen von 640 x 480 Pixeln gebracht. Es werden mehrere Topics für diesen Sensor verwendet, die verschiedene Werte liefern, wie ein *Image_raw*, das das unveränderte Kamerabild liefert. Dieses Topic ist relevant für die Nutzung der Informationen für eine Farbauswertung in einem separaten Python-Skript. Des Weiteren bietet die Kinect-Kamera eine Verbesserung bei Verzerrungen. Hierfür dienen die *distortion*-Variablen, die an die Verzerrungen angepasst werden können und ein entzerrtes Bild liefern, ähnlich wie bei einer Selbstkalibrierung einer Kamera.

Odometrie-Plugin

Das Odometrie-Plugin ist nicht notwendig für das Ausführen der Simulation. Es dient zur Kontrolle, ob die Werte der Gelenke mit den gewünschten Angaben übereinstimmen. Der Odometriesensor gibt die Lage von Komponenten, in Abhängigkeit ihres Ursprungs beim Erzeugen, in der Simulationswelt an

4.5 Python-Skripts

Die Simulation der Wasserversorgungsstation besitzt den Unterordner *scripts*. Dort befinden sich die meisten Python-Dateien, die für die Simulation notwendig sind. Hauptsächlich sind Publisher-Dateien im Ordner enthalten, die zum Starten der Wasserversorgungsstation benutzt werden. Sie publishen ein Topic, das durch eine Statemachine subskribiert wird. Bei einem gewissen Topicinhalt wechselt die Statemachine den Zustand und beginnt einen Prozess abzulaufen.

4.5.1 Detektieren von Farben

Für die Kameraauswertung enthält der Ordner *scripts* die Datei *CircleColorPub.py*, die die Reflexlichtschranke nachahmt. Das Python-Programm arbeitet mit OpenCV und wertet die Daten der Kinect-Kamera aus. Es wurde mit Hilfe des Online-Kurses [Con19] erstellt. Dabei sucht das Programm nach der Farbe Gelb und kontrolliert, ob sie sich in dem gesetzten Rahmen befindet, der ein Anfahren durch die Wasserversorgungsstation ermöglicht. Das Programm wird in diesem Kapitel näher erläutert.

Quellcode 14: Bibliotheken von CircleColorPub.py

```
#!/usr/bin/env python
import rospy
import cv2
import numpy as np
from cv_bridge import CvBridge, CvBridgeError
from geometry_msgs.msg import Twist
from sensor_msgs.msg import Image
from std_msgs.msg import String
target = "notDetected"
```

Diese Bibliotheken (vgl. Quellcode 14) müssen importiert werden, sodass die Bearbeitung und die Ausgabe der Kamerainformation möglich sind. Besonders wichtig ist die *cv_bridge* Bibliothek, die ROS erlaubt, mit OpenCV zu arbeiten.

Als Topic-Type wird ein String gewählt, der den Zustand der Detektion publiziert. Hierfür dient die Variable *target*, die einen String-Type besitzt.

Quellcode 15: Objektlasse Colordetection mit Initialisierungsunterfunktion

```
class Colordetector(object):

    def __init__(self):

        self.bridge_object = CvBridge()
        self.image_sub =
rospy.Subscriber("/camera/rgb/image_raw",Image,self.camera_callback)
```

Für die Detektion wird eine Klasse erstellt (vgl. Quellcode 15), die verschiedene Funktionen enthält, darunter die Initialisierungsfunktion, die zum einen die Brücke von ROS zu OpenCv aufbaut und zum anderen die Daten der Kinect-Kamera verarbeitet.

Quellcode 16: Callbackfunktion für die Kameradaten

```
    def camera_callback(self,data):

        try:
            # select bgr8 because it‘s the OpneCV default encoding
            cv_image = self.bridge_object.imgmsg_to_cv2(data, desired_encoding="bgr8")
        except CvBridgeError as e:
            print(e)
```

Mit jeder neuen Kamerainformation wird die *Callback*-Funktion aufgerufen (siehe Quellcode 16). Diese speichert die Daten auf die *cv_image*-Variable für die weitere Verarbeitung. Die gespeicherten Bilder werden im *bgr8*-Format gespeichert, dem Rot/Grün/Blau-Format, das in 8-Bit-Integer dargestellt wird. Dieses Format muss in der zukünftigen Verarbeitung verändert werden.
Sollte keine Brücke zwischen ROS und OpenCV hergestellt werden, wird eine Information auf die Kommandozeile weitergegeben, sodass eine Error-Nachricht erscheint.

Quellcode 17: Kameradatenverarbeitung

```
crop_img = cv_image[200:280][0:280]

# Convert from RGB to HSV
hsv = cv2.cvtColor(crop_img, cv2.COLOR_BGR2HSV)

lower_yellow = np.array([20,100,100])
upper_yellow = np.array([50,255,255])
```

Um die Datengröße zu minimieren, wird ein Teil des Bildes ausgeschnitten (vgl. Quellcode 17). Dadurch entsteht ein Bildschnitt in einer Größe von 280 x 80 Pixel. Dieser Schnitt enthält alle für die Verarbeitung relevanten Informationen.
Das RGB-Bild muss in ein HSV-Bildformat konvertiert werden. Dieses Format arbeitet mit dem HSV-Farbraum, der keine RGB-Werte beschreibt, sondern einen Farbwert, eine Sättigung und eine Dunkelstufe. Diese Werte werden für eine maximale und minimale Farbstufe gewählt und werden in den Variablen *lower_yellow* und *upper_yellow* gespeichert.

Quellcode 18: Maske für das Erkennen der gelben Farbe

```
#HSV image to get only yellow colors
mask = cv2.inRange(hsv, lower_yellow, upper_yellow)
```

Damit nur die gewünschte Farbe im Bild erkannt wird, wird eine Maske erstellt (vgl. Quellcode 18). Diese wandelt alle Farben in Schwarz und Weiß um. Dadurch kann die weitere Verarbeitung der Bilder schneller vollzogen werden.

Quellcode 19: Berechnung des Bildmoments

```
# Calculate center of binary image using ImageMoments
m = cv2.moments(mask, False)
try:
    cx, cy = m['m10']/m['m00'], m['m01']/m['m00']
except ZeroDivisionError:
    cy, cx = height/2, width/2

# and-mask and original image
res = cv2.bitwise_and(crop_img,crop_img, mask= mask)
```

Um dem Nutzer beim Beobachten der Kameraansicht die Farbe zu symbolisieren, wird ein roter Punkt in die Mitte der größten gelben Farbversammlung gezeichnet. Hierfür muss das Moment des Bildes berechnet werden, das die Helligkeitsstufe der einzelnen Pixel berechnet und einen Mittelwert bildet. Jedoch werden keine Helligkeitswerte verwendet, sondern die Farben der zuvor erstellten Maske, die ausschließlich Gelb und Schwarz zeigt. Diese Werte werden in der Variable m gespeichert. Aus den Werten kann ein Schwerpunkt gebildet werden, der den Mittelpunkt der Farbfläche markiert.
Das resultierende Bild wird in der Variable *res* erstellt, die mit der Maske über einen *and*-Operator zusammengefügt und mit dem zuvor bestimmten Schnitt verkleinert ausgegeben wird (vgl. Quellcode 19).

Quellcode 20: Einfügen eines roten Punktes auf die gelbe Fläche

```
# draw red circle in image
cv2.circle(res,(int(cx), int(cy)), 10,(0,0,255),-1)

cv2.imshow("Original", cv_image)
cv2.imshow("RES", res)
```

Die OpenCV-Bibliothek hält viele Funktionen zum Zeichnen vor. Es sind verschiedene Muster möglich, die auf ein Bild projiziert werden können. Bei diesem Beispiel soll ein einfacher Kreis gezeichnet werden, der seine Lage durch die Koordinaten *cx* und *cy* erhält. Dieser Kreis wird auf das resultierende Bild gezeichnet (vgl. Quellcode 20).

Quellcode 21: Abfrage über die Erreichbarkeit des Markers für die Zahnstange

```
if ((int(cx)<420 and int(cx) > 220) and not (int(cx)==320 and int(cy)==240 )):
    rospy.loginfo("%s cx =  %d, cy = %d " ,target,int(cx), int(cy))
    global target
    target="detected"

elif (int(cx)==0 and int(cy)==0):
    global target
    rospy.loginfo("%s cx =  %d, cy = %d " ,target,int(cx), int(cy))
    target = "notDetected"

elif (int(cx)==320 and int(cy)==240):
    global target
```

```
            rospy.loginfo("%s cx =  %d, cy = %d " ,target,int(cx), int(cy))
            target = "notDetected"

        else:
            global target
            rospy.loginfo("%s cx =  %d, cy = %d " ,target,int(cx), int(cy))
            target = "notDetected"
```

Dieser Kreis gilt als Orientierung für die Erreichbarkeit des Pflanzentopfes mit der Zahnstange. Sollte sich der Punkt nicht in einem gewissen Pixelbereich befinden, gilt er als nicht detektiert. Durch eine *if*-Abfrage erhält die Variable *Target* einen *detected*- oder *notDetected*-Wert, der nachfolgend in einem Topic publiziert wird (vgl. Quellcode 21).

Quellcode 22: Main-Funktion von CircleColorPub.py

```
def main():
    pub = rospy.Publisher('/ColorDetection', String, queue_size=10)
    colordetector_object = Colordetector()
    rospy.init_node('ColorDetected', anonymous=True)
    rate = rospy.Rate(5)

    while not rospy.is_shutdown():
        global target
        pub.publish(target)
        rate.sleep()

    if KeyboardInterrupt:
        cv2.destroyAllWindows()

if __name__ == '__main__':
    try:
        main()
    except rospy.ROSInterruptException:
        pass
```

Das *main()*-Programm erstellt das Topic für die *Target*-Variable. Sie nennt sich */Colordetection* und wird dem Node *Colordetection* zugeordnet. Andere Programme können auf dieses Topic zugreifen und die Information verwerten. Das Objekt, das für die Verarbeitung der Kamerafunktion zuständig ist, wird auf der Variable *colordetector_object* erstellt. Gleichzeitig wird jede kommende Kamerainformation subskribiert, verarbeitet und mit der Variable *Target* publiziert. Dieser Prozess findet so lange statt, bis das Programm durch den Nutzer mit einem *KeyboardInterrupt* gestoppt wird (vgl. Quellcode 22).

4.5.2 SMACH-Statemachine

Der Ablauf aller Gelenke und Sensoren wird anhand einer Statemachine vollzogen. Diese subskribiert zu den meisten Topics und steuert die Prozesse. Anders als bei normalen Python-Programmen wird hierfür eine SMACH-Statemachine nach [Sma17] verwendet. Sie arbeitet Hand in Hand mit dem ROS-System und bietet ein gutes Aufbaumuster einer Statemachine, sodass der Überblick für den Nutzer bewahrt werden kann. Die ersten Vorüberlegungen für einen Prozessablauf wurden bereits bei der Konzeptentwicklung angestellt. Für die Visualisierung des Prozessablaufes wurde ein Activity-Diagramm benutzt, das in drei Abläufe unterteilt ist (siehe Abbildungen 21, 22, 23).

Fehlerfreier Prozessablauf

Der erste Ablauf in Abbildung 21 stellt einen fehlerfreien Prozess der Wasserversorgung dar. Hierbei werden alle Umstände als optimal betrachtet, sodass beispielsweise die Entfernung von der Wasserstation zu der Roboterplattform als fehlerfrei betrachtet wird.

Fehlpositionierung

Bei einer Fehlpositionierung durch die Roboterplattform kann keine Bewässerung stattfinden. Dadurch können die Pfade aus dem fehlerfreien Prozess nicht abgehandelt werden, wodurch andere Bedingungen und Klassen beigefügt werden müssen.

Diese handeln den Fehler ab und lenken den Prozessablauf in eine andere Richtung. In Abbildung 22 wurde hierfür eine weitere Klasse namens *Signal for wrong position to the robot and customer* hinzugefügt. Diese wird bei einer Fehlpositionierung erreicht und soll dem Kunden und dem Roboter ein Fehlersignal vermitteln.

Leerer Wassertank

Der letzte Ablauf in Abbildung 23 beschreibt den Prozess für den Fehler, dass kein Wasser im Tank vorhanden ist. Diese Klasse handelt ebenfalls einen Signalprozess ab und kann in diesem Beispiel erst verlassen werden, wenn eine Bestätigungstaste an der Wasserversorgungsstation gedrückt wird.

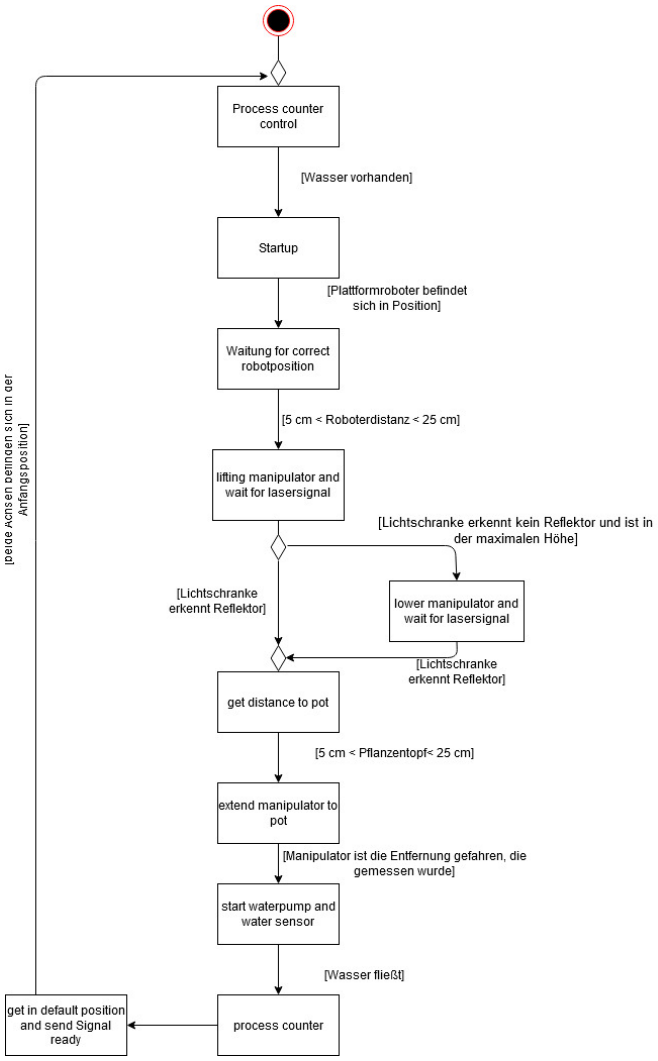

Abbildung 21: Fehlerfreie Bewässerungsprozess

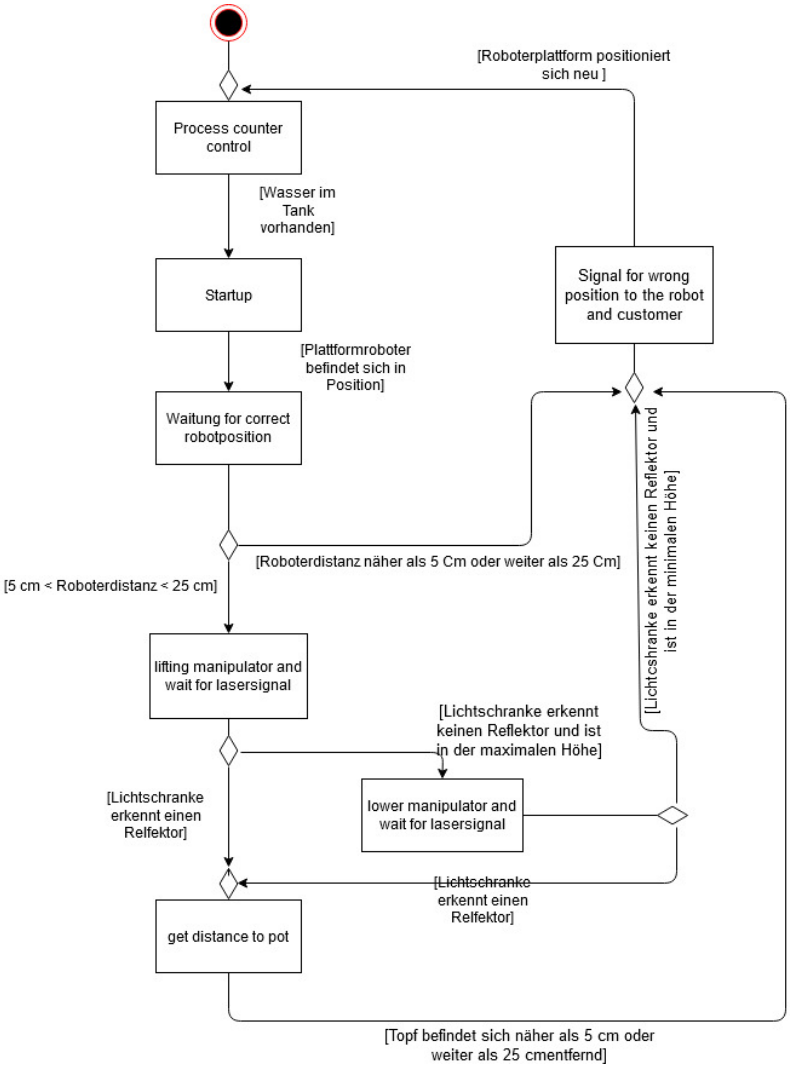

Abbildung 22: Bewässerungsprozess bei Fehlpositionierung des Roboters

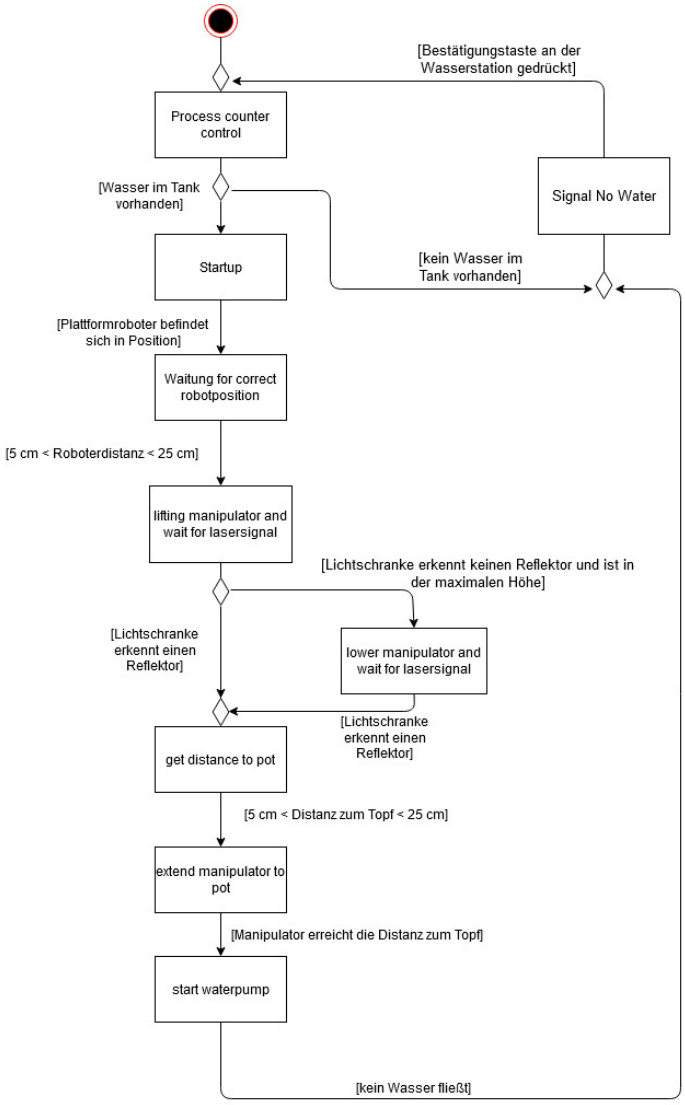

Abbildung 23: Prozess bei leerem Wassertank

Jede Klasse und Bedingung wurde in die SMACH-Statemachine integriert, jedoch können viele Bedingungen nicht umgesetzt werden und müssen mit Topics umgangen werden. Sie simulieren ein Signal, zum Beispiel einen Knopfdruck.

Um die SMACH-Statemachine zu benutzen, muss die Bibliothek *smach* und *smach_ros* integriert werden. Jede Klasse wird objektorientiert programmiert und besteht aus einer *Initialisierungs*-Funktion und einer *Execute*-Funktion.

Quellcode 23: Quellcode für eine Zustandsklasse in der Statemachine

```
class Teststate(smach.State):
    def __init__(self):
        smach.State.__init__(self, outcomes=['return'])

    def execute(self, userdata):
        rospy.loginfo('Executing state teststate')
        time.sleep(1)
        return 'return'
```

Beispielsweise besitzt die Klasse *Teststate* (siehe Quellcode 23) einen vordefinierten Weg, der unter *Outcomes* aufgelistet werden muss. Hierbei reicht der einfache Name des Übergangs, da die Verknüpfung in der *main()* vollzogen wird. Die jeweiligen *Outcomes* werden als *Return*-Wert in der *Execute*-Funktion wiedergegeben und öffnen den Pfad zur nächsten Klasse. Wenn diese mit Sensorwerten arbeitet, muss ein Subscriber in der *Initialisierungs*-Funktion, ähnlich wie beim Programmieren eines Subscribers, hinzugefügt werden. Dasselbe gilt für das Publishen von Werten über eine Klasse, sodass ein Topic und ein Node initialisiert werden müssten.

In der *main()* werden alle Klassen zu einem Prozess verknüpft (siehe Quellcode 24). Hierbei muss ein Node erstellt werden, der beim erfolgreichen Ausführen der Statemachine aktiviert wird. Die Statemachine wird konstruiert und auf eine Variable für weitere Verarbeitung gespeichert. Die vollständige Statemachine selbst besitzt auch Übergänge, die zum Verknüpfen mit weiteren Statemachines dient, um ein Cluster zu erstellen. Im Projekt der Wasserversorgungsstation wurden die Verbindungen erstellt, jedoch werden sie nicht benutzt.

Quellcode 24: Main-Funktion mit Hinzufügen aller Transitionen

```
def main():
    rospy.init_node('smach_example_state_machine')

    # Create a SMACH state machine
    sm = smach.StateMachine(outcomes=['outcome4', 'outcome5'])
```

```
    # Open the container
    with sm:
        smach.StateMachine.add('Teststate', Teststate(),
transitions={'return':'searchforColorYellow'})
```

Zur Statemachine werden alle Zustandsklassen einzeln hinzugefügt und die Pfade, die in der Klassendefinition benutzt werden, müssen unter *transitions* definiert werden. Es ist ebenfalls möglich, die Klasse mit sich selbst zu verbinden, jedoch führt dies öfter zu einer unendlichen Klasse, die auch bei anderen Bedingungen nicht verlassen werden kann.

Die fertige Statemachine der Wasserversorgungsstation ist in Abbildung 24 dargestellt. Sie wurde verändert und entspricht nicht vollständig dem Activity-Diagramm aus Abbildungen 21, 22 oder 23. Sie besitzt elf Zustandsklassen, welche durch 17 Transitionen miteinander verbunden sind.

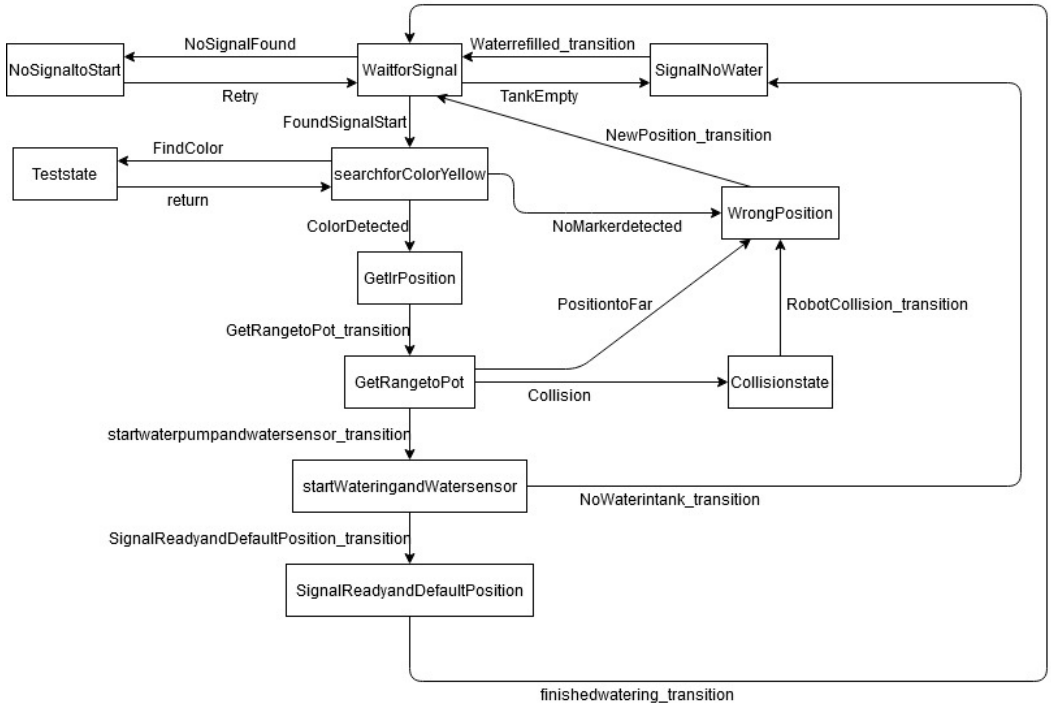

Abbildung 24: Activity-Diagramm der Wasserversorgungsstation für die Simulation

Folgende Klassen sind enthalten:

WaitforSignal

Dieser Zustand dient als Startklasse, die den Prozess der Bewässerung beginnt. Hierfür wird ein Signal benötigt, das durch einen Publisher ausgegeben wird. Sollte dieses nicht vorliegend oder einen falschen Wert enthalten, findet die Transition *NoSignalFound* statt. Bei einem abonnierten Signal mit gültigem Wert beginnt der Bewässerungsprozess mit der Transition *FoundSignalStart*. Der Zustand zählt außerdem die durchlaufenen Prozesse und gleicht die Anzahl mit dem verfügbaren Wasser im Tank ab, sodass bei einer Anzahl von beispielsweise 20 Durchläufen der Tank nachgefüllt werden muss und die Transition *TankEmpty* erfolgt.

NoSignaltoStart

Wenn kein Startsignal publiziert wird, soll dieser Zustand den Nutzer über diese Situation in der Kommandozeile informieren. Nach einer kurzen Information für den Nutzer erfolgt die *retry*-Transition.

SignalNoWater

Bei einem leeren Wassertank informiert die Zustandsklasse den Nutzer über den niedrigen Wasserstand und wartet auf eine Bestätigung, dass das Wasser nachgefüllt worden ist. Diese Bestätigung erfolgt in der Simulation durch eine längere Verzögerung, die anschließend die Transition *Waterrefilled* anleitet.

searchforColorYellow

Beim Erreichen dieses States beginnt die Suche nach dem gelben Marker. Dabei bewegt sich die erste Achse etappenweise zu ihrer maximalen Höhe. Ist die Achse um eine Etappe gestiegen und kein Marker wurde gefunden, erfolgt die Transition *FindColor*. Befindet sich die Achse auf der maximalen Auslenkung und es wurde kein Marker gefunden, erfolgt die Transition *NoMarkerdetected*. Wird der gelbe Marker durch die Kamera entdeckt, wird die Transition *ColorDetected* eingeleitet.

Teststate

Der Teststate dient als Benachrichtigungszustand, dass kein Marker auf dieser Höhe gefunden worden ist. Er enthält eine kurze Wartezeit und leitet die Transition *return* ein, um wieder in die Zustandklasse *searchforColorYellow* zu gelangen.

WrongPosition

Wenn die Roboterplattform in einer falschen Position steht und die Wasserversorgungsstation dies mit den Sensoren erfasst, erfolgt dieser Zustand. Der Nutzer wird über einen Topic benachrichtigt und die Roboterplattform kann neu positioniert werden. Bei einer realen Implementation würde dieser Zustand zur Kommunikation mit dem Roboter dienen, wodurch dieser sich selbstständig neu positioniert.

GetIrPosition

Bei gefundenem Marker muss die Entfernung bestimmt werden. Dies erfolgt durch den Infrarotlasersensor, der sich in die Höhe des Markers bewegt. Dadurch kann der genaue Abstand zum Marker erkannt werden, der für die zweite Achse zwischengespeichert wird. Nach dem Erfassen der Daten folgt die Transition *GetRangetoPot_transition*.

GetRangetoPot

Diese Klasse ist für das Anfahren der Pflanzentopfes mit der zweiten Achse zuständig. Die zwischengespeicherte Entfernung zum Marker wird in dieser Zustandsklasse verarbeitet. Dafür bewegt sich die Zahnstange in die gemessene Entfernung mit einem Faktor, der sicherstellt, dass die Zahnstange über dem Pflanzentopf zum Stehen kommt. Sollte diese Entfernung erreicht werden kann der nächste Zustand mit *startwaterpumpandwatersensor_transition* gestartet werden. Die Annäherung verhält sich, ähnlich wie bei der ersten Achse, etappenweise. Sollte eine Kollision oder Fehlpositionierung bemerkt werden, geht der Zustand in die Transition *PositiontoFar* oder *Collision*.

Collisionstate

Bei einem Zusammenprall misst der Kontaktsensor eine auf ihn wirkende Kraft. Dadurch startet der *Collisionstate*, der die Achsenbewegung sofort beendet. Nach einer Verzögerung von drei Sekunden wird die *Robotcollision_transition* eingeleitet.

startWateringandWatersensor

Sollten die vorherigen Klassen fehlerfrei durchlaufen werden und ein ausreichender Wasserstand vorhanden sein, startet der Hauptprozess für die Bewässerung. Der Wasserstand verändert sich am Ende dieses Zustandes, wodurch ein realer Wasserverbrauch symbolisiert wird. Mit einem Extra-Topic kann der Nutzer von außen entscheiden, ob ein Durchfluss registriert werden soll oder nicht. Dies soll den Wasserdurchflusssensor einfach darstellen und den Fall von zu wenig Wasser simulieren. Bei Erfolg wird der Übergang *SignalReadyandDefaultPosition_transition* gestartet.

SignalReadyandDefaultPosition

Als Letztes muss die Wasserversorgungsstation die Standardeinstellung der Achsen wieder einnehmen. Dazu werden die Achsen auf die Nullposition gefahren. Nun ist die Bedingung für die *finishedwatering_transition* erreicht, wodurch ein neues Anfahren durch die Roboterplattform gestartet werden kann.

4.6 Launch-Datei

Für das Einbinden von Robotermodellen in die Simulationswelt in Gazebo werden sogenannte *.launch*-Dateien verwendet. Diese werden in XML geschrieben und können ebenfalls mehreren Python-Programmen gleichzeitig starten und ausführen, wenn diese eingebunden werden. Über die Kommandozeile kann die Datei mit folgendem Befehl ausgeführt werden.

```
$ roslaunch <Workspacename> <Dateiname>
```

4.7 Deskriptives Simulationsmodell

Die Simulation wird auf der Seite www.theconstructsim.com gestartet. Für die Roboterplattform wurde ein Dummy erstellt (siehe Abbildung 25), der dem Roboter in Farbe und Größen entspricht.

Dummy

Als Roboterplattformdummy dient ein roter Kasten, der die Maße 455 x 381 mm und eine Höhe von 237 mm aufweist. Diese Maße sind ebenfalls die maximalen Abmessungen des Pioneer 3-DX. Auf dem roten Kasten befindet sich der Pflanzentopf mit einem Marker. Die blaue Box dient als vereinfachter Pflanzentopf und besitzt die Maße 100 x 100 mm und eine Höhe von 200 mm. Am oberen Rand des Topfes wurde ein gelber Marker fixiert. Dieser dient als Reflektor, der von der Reflexlichtschranke erkannt werden soll. Diese Aufgabe übernimmt die RGB-Kamera, die die gelbe Farbe erkennen kann. Um die verschiedenen Platzierungen der Roboterplattform zu ermöglichen, können alle Komponenten verschoben und auch in der Höhe variiert werden.

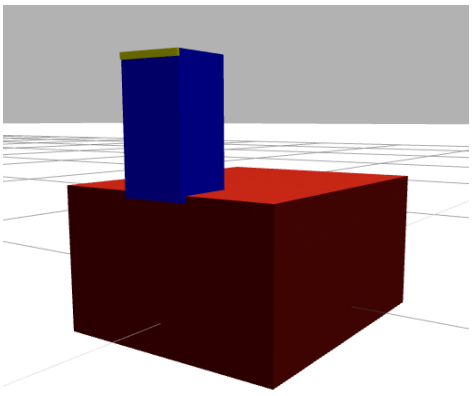

Abbildung 25: Simulationsdummy der Roboterplattform

Wasserversorgungsstation

Die Wasserversorgungsstation in Abbildung 26 besitzt alle Sensoren und für die Simulation relevanten Komponenten. Sie verfügt über keine besondere Farbe, da dies keine Einwirkungen auf die Simulationsergebnisse hat. Die Wasserflasche wurde blau

markiert. Die Zahnstange und die Komponenten, die den Lasersensor und die Kamera beinhalten, wurden mit einer roten Farbe markiert, um ihre Lage im Prozessablauf besser ersichtlich zu machen. Der Nutzer erhält auch die Möglichkeit, die Lage der Zahnstange über die Sensordaten der Odometrie zu ermitteln.

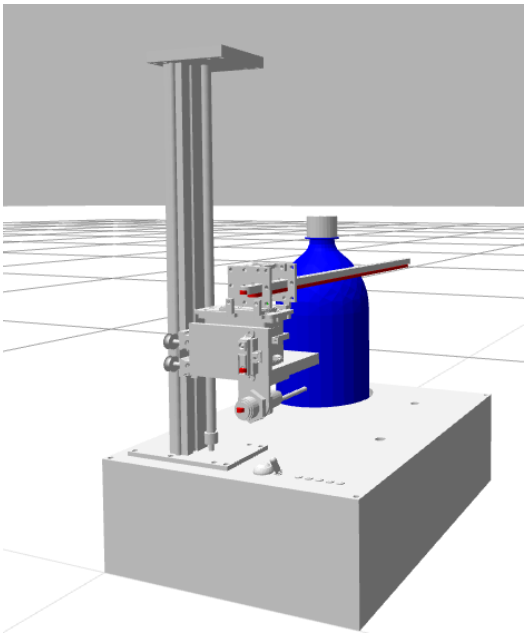

Abbildung 26: Wasserversorgungsstation in der Simulation

Starten der Simulation

Durch das Verwenden der Seite www.theconstructsim.com muss keine Simulationswelt über die Kommandozeile gestartet werden. Beim Arbeiten mit der eigenen Linux-Oberfläche muss dies vor dem Eingeben der nachfolgenden Befehle ausgeführt werden.

Alle Dateien werden anschließend über mehrere Kommandozeilen ausgeführt. Die Befehle der Wasserversorgungsstation sollten in der gezeigten Reihenfolge angegeben werden.

Zu Beginn muss die Wasserversorgungsstation erzeugt werden, wofür eine eigenen Kommandozeile verwendet werden muss. Diese aktiviert alle vorhandenen Plugins und führt diese bis zur Schließung durch den Nutzer aus.

```
$ roslaunch wasserversorgungsstation_description xacro_start.launch
```

Über eine weitere Kommandozeile wird der Dummy hinzugefügt, der anschließend nach Belieben platziert werden kann. Die Platzierung erfolgt über die Werkzeuge, die in der Gazebowelt vorhanden sind.

```
$ roslaunch wasserversorgungsstation_description Robotcube.launch
```

Damit sind alle wichtigen Komponenten in der Simulation erzeugt worden und müssen für die Ansteuerung mit der Statemachine und dem Programm für die Farberkennung verbunden werden. In derselben Kommandozeile wird der nachfolgende Befehl gestartet. Diese Kommandozeile gibt nun außerdem Informationen darüber, ob die gelbe Farbe detektiert wird.

```
$ roslaunch wasserversorgungsstation_description pythonfiles.launch
```

In einer dritten Kommandozeile muss die SMACH-Statemachine gestartet werden, die für die Ausgabe des jetzigen Zustands benutzt wird. Der Nutzer kann anhand dieser Kommandozeile jeden Zustandswechsel der Versorgungsstation erkennen und nachvollziehen.

```
$ rosrun wasserversorgunsstation_description statemachine.py
```

Nachdem alle Befehle ausgeführt worden sind, kann die Bewässerung durch die Wasserversorgungsstation begonnen werden. Die Simulation sollte ähnlich wie in Abbildung 27 aussehen.

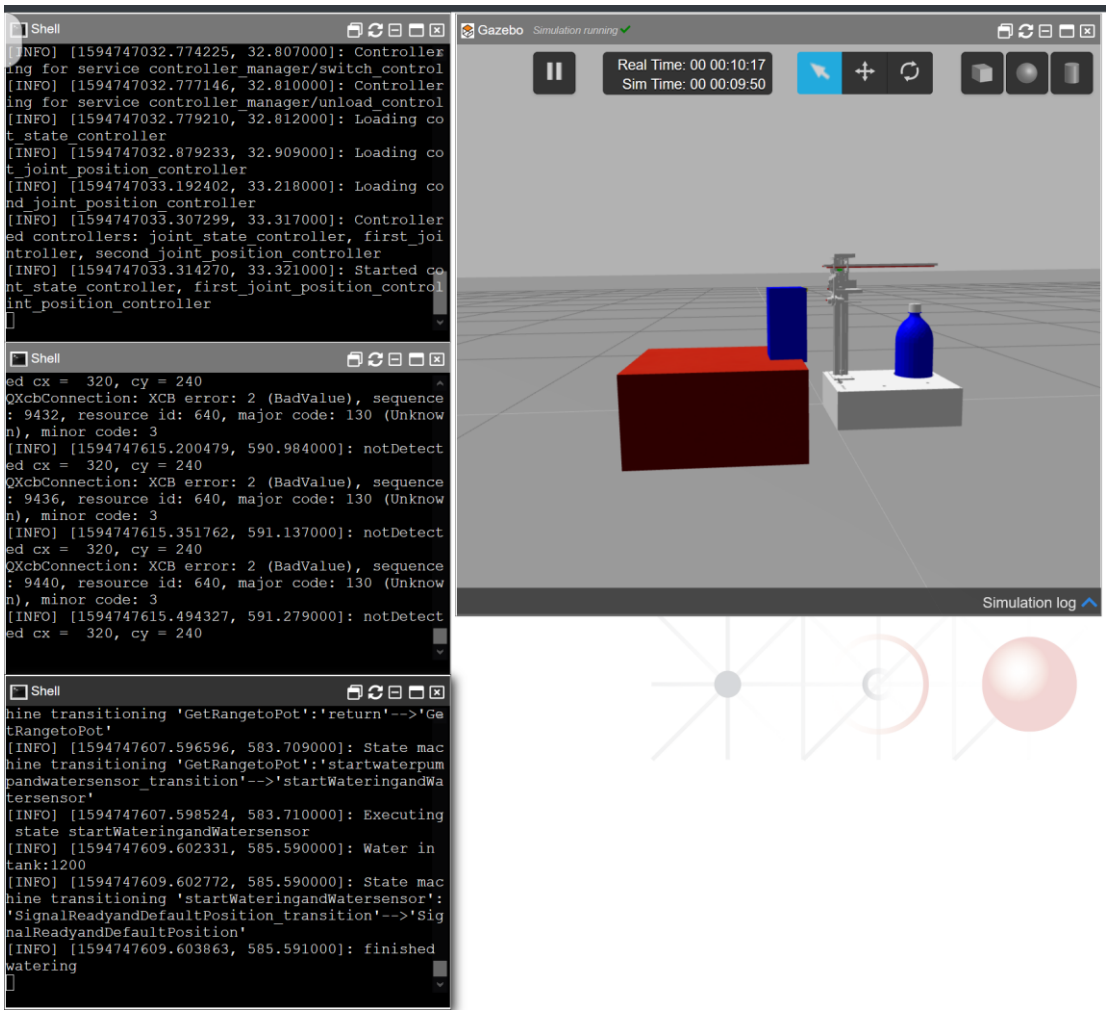

Abbildung 27: Fertiger Simulationsaufbau

5 Testen der Requirements

Mit dem Fertigstellen der Simulation müssen die Anforderungen auf Erfüllbarkeit getestet werden. Sie werden mit Hilfe einer Trace-Matrix visualisiert. Die Tabelle 8 nach [Leh19] beschreibt die Testfälle und welches Requirement getestet wird. Für einen Teil der Anforderungen werden keine Testfälle benötigt, da sie durch die Konstruktion erfüllt worden sind. Einige Anforderungen können nicht mit der Simulation getestet werden, da sie wesentlich mehr Inhalt benötigen würden, beispielsweise eine interaktive Umwelt. Aus diesen Gründen werden nur ausgewählte Requirements getestet.

Tabelle 8: Trace-Matrix mit Testresultaten

Req. Nr.	T1	T2	T3	T4	T5	T6
1.3	**Pass**					
3.1		**Pass**				
3.1.2			**Pass**			
3.1.3				**Pass**		
3.3		**Pass**				
5.2					**Pass**	
5.3					**Pass**	
5.4						**Pass**
5.4.1		**Pass**				

Für die Anforderungen wurden mehrere Testfälle generiert, die mehrfach für das Testen benutzt werden konnten. Die Testfälle lauten:

- T1: Testen der Klassen für nicht vorhandenes Wasser im Tank
- T2: normaler Ablauf einer Befüllung
- T3: Testen beim Anfahren verschiedener Markerhöhen
- T4: Testen beim Anfahren verschiedener Entfernungen zum Marker
- T5: Ablesen der Odometriedaten für das Verifizieren der Höhe
- T6: Ausgabe des Lasersignals in Metern

6 Fazit und Ausblick

Fazit

Als Fazit lässt sich festhalten, dass das Ziel des Projektes erreicht werden konnte. Es wurde eine autonome Wasserversorgungsstation konstruiert, die den Anforderungen der Stakeholder entspricht. Die Wasserversorgungsstation bewässert eine Pflanze selbständig und interagiert mit der Roboterplattform. Es wurde eine Simulation der Wasserversorgungsstation erstellt, um die Funktionalität zu testen und die Interaktion einzelner Komponenten zu prüfen. Aufgetretene Fehler wurden behoben und die Konstruktion so stetig erweitert.

Es wäre interessant zu sehen, wie das Testen der Requirements in der Realität ausfallen würde und welche Probleme beim Bauen entstehen. Eine Simulation birgt zwar ebenfalls viele Probleme in der Programmierung, das Bauen jedoch erlaubt eine andere Sichtweise auf das Projekt und bringt eine andere Art von Problemen mit sich. In der Simulation konnten Komponenten schnell ausgetauscht werden und ein neuer Testlauf gestartet werden. Zudem entstehen während einer Simulation keine Kosten und keine realen Rückschläge. Probleme traten jedoch wiederholt in der Programmierung auf, da häufig Schreibfehler oder Fehler in der Komptabilität der Plugins auftraten. Eine weitere Schwierigkeit stellt die Realitätsnähe dar. Die Umsetzbarkeit des Projektes darf nicht außer Acht gelassen werden. Des Weiteren ist man angewiesen auf die Simulationswelt und ihrem Inhalt, der mitunter nicht alle notwendigen Komponenten bevorratet.

Das Ergebnis der finalen Simulation ist eine funktionsfähige, automatisierte Wasserversorgungsstation für mobile Pflanzen, die anhand der beschriebenen Bauteile in der Realität gebaut werden kann.

Ausblick

Als Verbesserungsmöglichkeiten können folgende Punkte betrachtet werden:

- Kontaktsensor: In der Simulation beinhaltet die Achse, die sich zum Pflanzentopf bewegt, einen Kontaktsensor, der eine Kollision verhindern soll. In der Konzeptentwicklung ist dieser noch nicht integriert worden, jedoch sollte er beim Bauen des Projekts berücksichtigt werden.
- Verkleidung: Die Wasserversorgungsstation wurde nur für die Simulation konstruiert und enthält keine Verkleidung, die die Achsen und die Flasche versteckt. Die äußere Erscheinung könnte demnach noch verbessert werden.
- Simulationsumgebung: Für die Simulation wurde eine Internetseite namens www.theconstructsim.com benutzt. Sollte sich eine Person mit dem Projekt weiter beschäftigen, ist eine Linux-Umgebung zu empfehlen. Die Simulation über www.theconstructsim.com ermöglicht das Darstellen des Roboters, jedoch arbeitet die Simulation an ihren Grenzen, sodass das Integrieren eines echten Pioneer-3-DX-Modells mit verschiedenen Sensoren schwer realisierbar ist. Zudem könnte man als Simulationswelt den Living Place darstellen, was eine größere Realitätsnähe bringen würde.

Literatur

[Con19] theconstruct. (2020, Juni 15). *ROS Perception in 5 Days*. URL: https://www.theconstructsim.com/robotigniteacademy_learnros/ros-courses-library/ros-perception-in-5-days/

[Dai17] Prof. Dr. Zhen Ru Dai. Software Engineering. Vorlesungsmaterial 2017.

[Gaa18] *Gazebo Answers*. (2019, September 11). Gazebo Answers. URL: https://answers.gazebosim.org/question/20432/ros-gazebo-detecting-collision-with-a-static-object-using-contact-sensor/

[GdR15] Maier, H. & VDE-Verlag Gmbh. (2015). *Grundlagen der Robotik*. Beltz Verlag.

[Ihi19] Institute for Healthcare Improvement. (2019). *Failure Modes and Effects Analysis (FMEA) Tool*. URL: http://www.ihi.org/resources/Pages/Tools/FailureModesandEffectsAnalysisTool.aspx

[Ir18] *GP2Y0A41SK0F | Sharp Reflektierender Sensor, 300mm | RS Components*. (o. J.). rs-online. URL: https://de.rs-online.com/web/p/products/6666568?cm_mmc=DE-PPC-DS3A-_-google-_-3_DE_DE_M_SEMIS_Phrase-_-sharp%7Creflexions_lichtschranke-_-gp2y0a41sk0f&matchtype=p&kwd-10777241727&gclid=EAIaIQobChMItcqAzvDP6gIVweJ3Ch1FWA-yEAAYASAAEgLpuvD_BwE&gclsrc=aw.ds

[Irs18] T. (2018, Dezember 7). *Integrating sonar and IR sensor plugin to robot model in Gazebo with ROS.* Medium. URL: https://medium.com/teamarimac/integrating-sonar-and-ir-sensor-plugin-to-robot-model-in-gazebo-with-ros-656fd9452607

[Joi20] *urdf/xml/joint*. (o. J.). Ros.org. Abgerufen 2020, von URL: http://wiki.ros.org/urdf/XML/joint

[Kin17] *kinect_camera - ROS Wiki*. (o. J.). wiki.ros.org. Abgerufen 2020, von URL: http://wiki.ros.org/kinect_camera

[Lea15] Joseph, L. (2015). *Learning Robotics Using Python*. Van Haren Publishing.

[Leh19] Prof. Dr. Thomas Lehmann. Systems und Software Engineering. Vorlesungsmaterial. 2019.

[Liv19] *Living Place* |. (2019). Haw-LivingPlace. URL: https://livingplace.haw-hamburg.de/

[Lix20] *urdf/xml/link*. (o. J.). Ros.org. Abgerufen 2020, von URL: http://wiki.ros.org/urdf/XML/link

[Rco16] Meeussen, W. (2016). *ros_controll*. Ros.org. URL: http://wiki.ros.org/ros_control

[Ref20] *Pepperl+Fuchs 246430 | Reflektor rechteckig 180x40x8mm Befestigungsbohrungen PMMA/ABS | Schäcke*. (o. J.). schaeke. https://www.schaecke.at/aus/Kategorien/Steuern-%26-Regeln/Sensorik/Reflektor-f%C3%BCr-Lichtschranke/Reflektor-rechteckig-180x40x8mm-Befestigungsbohrungen-PMMA-ABS/p/4371712

[Rfx20] Components-Store.com. (o. J.). *CY-192B-P-Y, Panasonic CY-192B-P-Y in Stock available. Buy CY-192B-P-Y with best price at*. Abgerufen 2020, von https://www.components-store.com/product/Panasonic/CY-192B-P-Y.html

[Rob18] Joseph, L. (2018). *Robot Operating System (ROS) for Absolute Beginners*. Apress.

[Rog19] *ROS.org | About ROS*. (o. J.). www.ros.org. https://www.ros.org/about-ros/

[Ros20] ROS Official. (o. J.). *ROS Tutorials*. ROS.org. Abgerufen 2020, von URL: https://wiki.ros.org/ROS/Tutorials

[Sch19] *Schrittmotor ACT 23HM6620 0,9°, 2/4 Phasen, 5,04 V/3,6 V online kaufen.* Pollin Electronic GmbH. Abgerufen 2020, von https://www.pollin.de/p/schrittmotor-act-23hm6620-0-90-2-4-phasen-5-04-v-3-6-v-310787

[Sec19] *785 Gear Rack Kit (Single Perpendicular)*. (o. J.). Servocity. Abgerufen 2020, von https://www.servocity.com/785-gear-rack-kit-637170

[Sma17] *smach/Tutorials/Simple State Machine - ROS Wiki*. (2016). wiki.ros.org URL: http://wiki.ros.org/smach/Tutorials/Simple%20State%20Machine

[Wap20] *Wasserpumpe*. (o. J.). Funduinoshop.com. Abgerufen 2020, von https://www.funduinoshop.com/Wasserpumpe

[Wfs20] *Water Flow Sensor YF-S401*. (o. J.). yf-s401-pvc-water-flow-hall-sensor-flowmeter-counter-wh.pdf. Abgerufen 2020, von https://5.imimg.com/data5/VQ/ME/MY-1833510/yf-s401-pvc-water-flow-hall-sensor-flowmeter-counter-wh.pdf

[Zpk3] *Mobiler Roboter Pioneer P3-DX*. (o. J.). Génération Robots. https://www.generationrobots.com/de/402395-robot-mobile-pioneer-3-dx.html

[Abu19] Prof. Dr. Jutta Abulawi. Konstruktion Maschinenelemente. Vorlesungsmaterial. 2019

[Zmk17] Prof. Dr. Hans-Joachim Schelberg. Poduct Design. Vorlesungsmaterial. 2017

[Zga20] *Gazebo*. (o. J.). Gazebosim. http://gazebosim.org/

[Zle20] *SMBD08014 LED-Signalleuchte Rot 8.2 mm 28 V Signal-Construct*. (o. J.). Distrelec Deutschland. https://www.distrelec.de/de/led-signalleuchte-rot-mm-28-signal-construct-smbd08014/p/13371114

[Zzc27] *Netzteilbaustein Mean Well LRS-150-12 AC/DC kaufen*. (o. J.). www.conrad.de. Abgerufen 2020, von https://www.conrad.de/de/p/mean-well-lrs-150-12-ac-dc-netzteilbaustein-geschlossen-12-5-a-150-w-12-v-dc-1439463.html

Anhang

A.1. Inhalt der beigefügten CD

- Bachelorthesis
- CAD Daten
- Quellcode
- Videodatei
- Datenblätter
- Tabellen
- Bilder

A.2. Verwendete Geräte und Software

- Creo Parametric 2019
- Linux Oberfläche auf der Seite www.theconstructsim.com
- Ubuntu 18.04
- ROS Kinetic
- Blender 2.83.2
- FreeCAD 2020

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, den _____ _____