ING Fakultät Ingenieurwissenschaften

HTWK Hochschule für Technik, Wirtschaft und Kultur Leipzig

Masterarbeit

Thema: Simulation und Evaluation realistischer Bewegungsabläufe bei der Mensch-Roboter-Kooperation

vorgelegt von: Bastian Hofmann

Studiengang: Elektrotechnik und Informationstechnik
Studienprofil: Mechatronik

Verantwortlicher Hochschullehrer: Prof. Dr.-Ing. J.Jäkel
Betrieblicher Betreuer: Dr.rer.nat. A.Hoffmann

Ausgabetermin: 16.01.2023 Abgabetermin: 03.07.2023

Leipzig, 01.01.2023

Prof. Dr.-Ing. F. Illing
Prüfungsausschussvorsitzender

Inhaltsverzeichnis

ING Fakultät
Ingenieurwissenschaften

HTWK
Hochschule für Technik,
Wirtschaft und Kultur Leipzig

Aufgabenstellung zur Masterarbeit

für Herrn Bastian Hofmann, B.Eng.

Thema
Simulation und Evaluation realistischer Bewegungsabläufen bei der Mensch-Roboter-Kooperation

Beschreibung
Die Automatisierung von Prozessabläufen wird häufig in Etappen durchgeführt. Prozesse, welche selten oder oft unterschiedlich ausgeführt werden sollen, werden häufig nicht automatisiert, da der Aufwand der Automatisierung disproportional zum eigentlichen Aufwand der Aufgabe ist. Die Vollautomatisierung eines Prozesses hingegen ist sinnvoll, falls gleiche Aufgaben häufig wiederholt werden müssen. Hierbei ist der Aufwand der Automatisierung häufig kleiner als der Aufwand des gesamten Prozesses ohne Automatisierung.

Eine Art Zwischenschritt stellt das Feld der Mensch-Roboter-Kooperation dar, in welcher Roboter und Mensch zur selben Zeit am gleichen Ort arbeiten. Dabei wird die Ausdauer von Robotern durch die Flexibilität des Menschen ergänzt, welcher komplizierte oder kognitiv komplexe Tätigkeiten übernehmen kann. Bei diesen Projekten steht vor allem die Sicherheit der Arbeiter im Vordergrund, welche durch Roboter gefährdet werden können.

Für die Modellierung von Abläufen werden häufig Beschreibungssprachen verwendet. Diese Sprachen teilen die Abläufe in kleinere Schritte auf, welche das Verhalten darstellen. Leichte Modifizierbarkeit und Lesbarkeit sind dabei große Vorteile moderner Beschreibungssprachen.

In dieser Arbeit sollen die Vorteile einer Beschreibungssprache genutzt werden, um die Interaktion zwischen Mensch und Roboter in einer Simulation abzubilden. Hierzu sollen realitätsnahe Bewegungsabläufe erstellt werden, mit welchen Mensch und Roboter in Interaktion treten können, beispielsweise das Aufheben von Gegenständen, Kontrolle des Arbeitsbereichs und Übergabe von Gegenständen. Diese Aktionen sollen für die Beschreibungssprache verfügbar gemacht werden, um darin die Interaktion zwischen Mensch und Roboter modelliert werden kann.

Diese Bewegungsabläufe sollen in drei Szenarien mit unterschiedlichem Kollaborationsgrad genutzt werden. Hierfür sollen unterschiedliche Aufgaben für Koexistenz, Kooperation und Kollaboration erdacht und modelliert werden. Hierzu sollen für die Szenarien relevante Bewegungsabläufe, wie zum Beispiel das Greifen in den Arbeitsbereich oder die Übergabe von Objekten, bei der Roboter-Interaktion mit simulierten Personen dargestellt werden.

Diese sollen anhand vorgegebener Regeln in der Beschreibungssprache ausgelöst werden, um eine realitätsnahe Simulation zu ermöglichen. Außerdem sollen durch diese Aktionen bewegte Objekte auch durch die Beschreibungssprache in der Simulation erstellt und bewegt werden können und durch andere Aktoren im System verfolgt werden können. Außerdem ist die Anpassung der Beschreibungssprache auf zufälliges Verhalten vorzunehmen, da menschliches Verhalten nur selten durch lineare Abläufe beschrieben werden kann.

Der Projektablauf soll dabei zuerst mit einer Recherche zu möglichen Ausgangspunkten für eine Simulationsumgebung beginnen. Hierfür sollen einige Daten zu verschiedenen Umgebungen gesammelt und evaluiert werden. Aus diesen Daten soll die am besten passende Umgebung ermittelt werden. In dieser Umgebung soll ein System für menschliche Aktoren implementiert werden, um animierte Aktionen und Bewegungen im Raum darstellen und simulieren zu können. In dem so entwickelten System sollen dann mehrere Beispielszenarien implementiert werden, welche verschiedene Interaktionsgrade zwischen Mensch und Roboter darstellen, um das entwickelte System zu testen.

Verantwortlicher Hochschullehrer: Prof. Dr.-Ing. Jens Jäkel (HTWK Leipzig)
Betrieblicher Betreuer: Dr. Alwin Hoffmann (XITASO GmbH)

Leipzig, 13. Dezember 2022

1 Einleitung

1.1 Motivation

Das Feld der Mensch-Roboter-Kollaboration entwickelt sich mit zunehmender Geschwindigkeit fort. Viele Unternehmen bieten neue Lösungen für die unterschiedlichsten Einsatzszenarien der Endanwender.

Dabei ist eine Prüfung des Anwendungsfalls sinnvoll, um etwaige Probleme der Interaktion früh erkennen und beheben zu können. Diese Prüfung kann durch eine Simulation, in welcher der konkrete Anwendungsfall abgebildet wird, vereinfacht werden.

Außerdem bietet eine Simulation die Möglichkeit, die Aufgabe des Roboters, ohne dessen Anschaffung, evaluieren zu können. Das so gefertigte Modell des Anwendungsfalls könnte später auch als Grundlage für einen digitalen Zwilling dienen. Dieser kann später zur Wartung und Fehlerdiagnose des Systems genutzt werden.

-MRK häufiger ein Thema -Anwendungsfälle sollen evaluiert werden -Erprobung von Szenarien ohne Roboter

->Simulation eine kompletten Szenarios mit Roboter und Mensch

1.2 Stand der Wissenschaft

Aktuelle Arbeiten:

-Planung von Interaktionen

-Parametervergleiche von maschinellen und menschlichen Bewegungen

-Vermeidung von Kollisionen und Strategie

-Steuerung von Robotern mit Behavior Trees

-> Keine allgemeine Simulation einen gesamten Szenarios mit Mensch.

`https://www.sciencedirect.com/science/article/pii/S2351978918311442` `https://www.researchgate.net/publication/319888916_Interactive_Simulation_of_Human`

-robot_Collaboration_Using_a_Force_Feedback_Device https://elib.dlr.de/120687/1/human_motion_projection.pdf https://www.researchgate.net/publication/220065749_Human-Robot_Collaboration_a_Survey

1.3 Welche Szenarien

Die drei zu modellierenden Szenarien sollten so gewählt werden, dass in vorherigen Szenarien genutzte Bestandteile in späteren, komplexeren Szenarien weiter genutzt werden können. Hierfür kommen bestimmte Aufgaben, wie zum Beispiel die Interaktion mit Objekten, besonders in Frage, da diese viele ähnliche Bestandteile haben, jedoch mehrere Szenarien denkbar sind. Dazu zählen zum Beispiel das Hineingreifen in einen Prozess, das Aufheben von Material oder das begutachten eines Objekts, welche alle nur eine Bewegungsabfolge eines Menschen sind.

Das erste abgebildete Szenario soll sich mit der Simulation einer bereits vollautomatisierten Fertigungsaufgabe handeln, in welcher ein Roboter im Arbeitsbereich eines Menschen Teile fertigt. Die zu erwartende Interaktion beschränkt sich hierbei auf die Anpassung der Fahrgeschwindigkeit bei Annäherung des Menschen, um Kollisionen zu vermeiden.

Dieses Szenario ist ein Beispiel für eine Koexistenz zwischen Roboter und Mensch, wo beide an unterschiedlichen Aufgabe, jedoch im selben Raum, arbeiten. Außerdem werden grundlegende Aspekte der Simulation getestet, wie zum Beispiel das Bewegen von Mensch und Roboter und die sicherheitsrelevante Aktion der Geschwindigkeitsanpassung.

Im zweiten Szenario prüft und sortiert der Roboter Teile und legt die guten Exemplare auf einem Fließband zur Weiterverarbeitung ab. Die Mängelexemplare werden hingegen in einer besonderen Zone abgelegt, von wo sie vom Menschen abtransportiert werden.

In diesem Szenario wird das vorhergegangene Szenario um weitere Aspekte ergänzt, was dieses zu einem

Die dritte simulierte Aufgabe stellt ein Kollaborationsszenario dar, in welchem Mensch und Roboter an der selben Aufgabe arbeiten. Hierbei soll eine Palette entladen werden, wobei der Roboter nicht jedes Objekt ausreichend manipulieren kann. Dies resultiert in Problemen beim Aufheben, Transport und Ablegen der Objekte. In diesen Fällen muss nun ein Mensch aushelfen, wodurch er mit dem Roboter in Interaktion tritt.

1.4 Welcher Nutzen / Contributions

Durch diese Arbeit soll in zukünftigen Projekten die Möglichkeit geschaffen werden, konzeptionelle Probleme bei der Erstellung neuer Aufgabenbereiche eines Roboters frühzeitig durch Simulation erkennbar zu machen.

Dazu ist eine schnelle Konfiguration von sowohl Roboter als auch Mensch auf unterschiedliche Szenarien nötig, welche durch eine Beschreibungssprache definiert werden sollen. Durch deren einfache Struktur soll komplexes Verhalten einfach und überschaubar definierbar sein, welches dann in der Simulation getestet werden kann. - Erkennen von konzeptionellen Problemen vor Ersteinsatz

- Definition von Interaktion mit einfacheren Strukturen als State-Machines

2 Konzept

Die zu entwickelnde Simulation soll die bisher meißt separaten Zweige der Roboter- und Menschensimulation verbinden. Um die beiden Akteure in der simulierten Umgebung zu steuern, werden Befehle von außerhalb der Simulation eingesetzt. Diese Befehle werden dabei von externer Software unter der Verwendung einer Beschreibungssprache und Feedback aus der Simulation generiert.

Die zu erarbeitende Softwareumgebung soll einfach erweiterbar sein, um weitere Modifikationen und die Umsetzung anderer Projekte zuzulassen. Hierzu zählt die Austauschbarkeit und Erweiterbarkeit von Komponenten wie der simulierten Welt, dem Roboter oder dem simulierten Mensch. Um diese Möglichkeiten zu schaffen, sind die Systeme modular aufgebaut.

2.1 Simulation des Roboters

Der simulierte Roboter soll für viele unterschiedliche Szenarien nutzbar sein, was spezialisierte Robotertypen ausschließt. Außerdem ist die enge Interaktion mit Menschen interessant, was einen für Mensch-Roboter-Kollaboration ausgelegten Roboter spricht. Für diese beschriebenen Kriterien eignet sich der KUKA LBR iisy, welcher als Cobot vermarktet wird. Cobot ist dabei ein Portemanteau aus Collaborative und Robot, was die besondere Eignung für Mensch-Roboter-Kollaboration noch einmal unterstreicht. Er besitzt auch einen modifizierbaren Endeffektor, um unterschiedlichste Aufgaben erfüllen zu können.

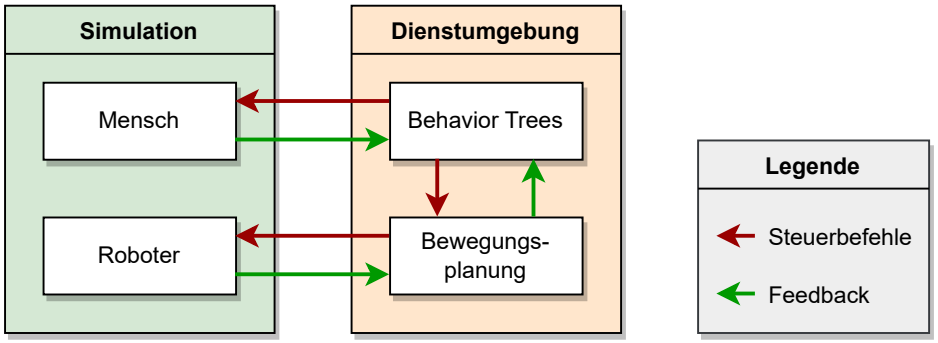

Abbildung 2.1: Visualisierung des Konzepts

Um den Kuka iisy in der Simulation verwenden zu können, muss ein Modell des Roboterarms erstellt werden. Dieses Modell sollte die physikalischen Eigenschaften des Roboters möglichst gut wiederspiegeln. Anhand dieses Modells kann der Roboter dann in der Simulation dargestellt werden und mit anderen Objekten interagieren.

2.2 Simulation des Menschen

Der Mensch soll in der Simulation typische Aufgaben erledigen und häufiges Verhalten abbilden können. Hierzu werden Animationen verwendet, welche die aktuelle Tätigkeit darstellen. Für komplexere Verhaltensweisen können Animationen und andere Aktionen, wie zum Beispiel eine Bewegung und Rotation kombiniert werden, um zum Beispiel die Aktion “laufen” auszuführen.

Auch hier wird ein Modell der Person für die Simulation benötigt. Außerdem werden mehrere Animationen und Übergänge zwischen diesen benötigt, um bestimmte Bewegungen darstellen zu können. Hinzu kommt noch eine Komponente, welche diese Animationen und andere Parameter von außen entgegennehmen kann, um sie in der Simulation ausführen zu können. Um die spätere Steuerung des Menschen von außerhalb zu erleichtern, müssen diese Aktionen im Fortschritt überwacht und abgebrochen werden können.

2.3 Behavior Trees als Beschreibungssprache

Häufig wird Verhalten in State-Machines ausgedrückt, welche jedoch einige Nachteile besitzen. State-Machines werden ab einer gewissen Größe schnell unübersichtlich. Dies erschwert die schnelle Erfassung von Abfolgen und Zustandsübergängen bei Änderungen am Code, welche jedoch essentiell für den Betrieb einer Maschine sind. Um diese Probleme zu adressieren, entstand das Konzept der Behavior Trees.

Ein Behavior Tree ist eine Struktur, um Verhalten als ein Baum zu beschreiben. Der Ablauf startet vom sogenannten Root, der Wurzel des Baums. Von dort an werden sogenannte Nodes, welche je nach Node unterschiedliches Verhalten abbilden, miteinander verbunden. Die Nodes werden untereinander angeordnet, welches die Relation der Nodes zueinander beschreibt. Jede Node hat dabei entweder die Root-Node oder eine andere Node über ihr im Baum und eine beliebige Anzahl an Nodes unter sich. Hierbei gibt es mehrere grundlegende Arten von Tree-Nodes.

Aktions-Nodes beschreiben einzelne ausführbare Aktionen. Mit Hilfe von Parametern kann ihr Verhalten von anderen Nodes beeinflusst werden.

Dekorations-Nodes können den Rückgabewert einer anderen Node modifizieren. Häufig existieren hier Negation, garantierter Erfolg und garantierter Fehler.

Sequenz-Nodes beschreiben eine nacheinander ausgeführte Abfolge von anderen Nodes, welche mit spezifischen Randbedingungen weiter fortschreitet.

Fallback-Nodes werden verwendet, um Verhalten zu definieren, welches nur bei Fehlern in vorherigen Nodes ausgeführt wird.

In dieser Arbeit sollen BehaviorTrees für die Steuerung von Mensch und Roboter verwendet werden. Die hierfür erstellten Nodes sollen universell gestaltet werden, um alle Szenarien, welche in dieser Arbeit bearbeitet werden, abzudecken.

2.4 Virtualisierungsumgebung als Platform

Aufgrund von vorheriger Erfahrung mit komplizierten Setupprozessen, ist der Einsatz fest definierter Umgebungen unerlässlich. Dies soll in diesem Fall durch eine Virtualisierungsumgebung geschehen, welche die verwendeten Programme auf verschiedenen Systemen lauffähig macht.

3 Komponenten-/Softwareauswahl

Die Auswahl der verwendeten Softwarekomponenten ist ein wichtiger Schritt der Entwicklung, da diese Entscheidungen den späteren Entwicklungsprozess nachträglich beeinflussen. Hierfür werden Komponenten ausgewählt, welche die im Konzept besprochenen Teilbereiche abdecken und miteinander verbunden werden können.

3.1 Dienstumgebung (ROS2)

Die Dienstumgebung bestimmt maßgeblich, wie Software im Entwicklungsprozess geschrieben wird. Durch sie werden häufig benötigte Funktionen bereitgestellt, welche in Programmen in der Umgebung genutzt werden können.

Bei einer Dienstumgebung für Roboter gehören zu den grundlegendn Aspekten die Nachrichtenübergabe zwischen einzelen interagierenden Programmen, um eine gemeinsame Basis für ein einfach erweiterbares System zu schaffen.

3.1.1 Auswahl

Es existieren mehrere Systeme, welche als Dienstumgebung für Roboter in Frage kommen, wenn es nur um die Nachrichtenübergabe zwischen Programmen geht. Jede Lösung, welche Nachrichten zwischen Prozessen austauschen kann, ist ein potentieller Kanidat für ein Roboterframework.

Wichtige Aspekte sind dabei die Geschwindigkeit der Anbindung und die Definition der Nachrichten, welche über das System ausgetauscht werden.

Nutzbare, bereits als IPC integrierte Systeme sind zum Beispiel Pipes, welche Daten zwischen Prozessen über Buffer austauschen. Auch die Nutzung von Message Queues und Shared Memory ist hierfür denkbar. Diese Systeme sind performant, jedoch nicht einfach zu verwalten, da sie von einer direkten Kommunikation von 2 oder mehr Komponenten, welche exakt für diesen Zweck entwickelt wurden, ausgehen.

Eine Alternative stellen Sockets dar, welche Daten zwischen zwei Programmen austauschen können.

In diesem Bereich sticht ROS als Dienstumgebung für Roboter hervor, da es sich um ein etabliertes, quelloffenes und häufig verwendetes System handelt. Es bietet die oben genannten Aspekte und einige weitere Verbesserungen, welche später näher beleuchtet werden.

Die neuste Version ROS2 bietet dabei einige Verbesserungen im Vergleich zu früheren Version ROS1. Ein neues Nachrichtenformat mit Quality of Service kann zum Beispiel Nachrichten vorhalten und über sowohl TCP als auch UDP kommunizieren. Außerdem werden nun neben CMake auch andere Buildsysteme unterstützt, unter anderem auch Python.

Generell existieren im Feld der Roboter-Dienstumgebungen keine Alternativen mit ähnlichem Funktionsumfang und gleicher Reichweite. Vor allem die unzähligen ROS-Bibliotheken, welche von Nutzern des Systems über die Jahre erstellt wurden, machen das System so populär.[6]

ROS kann für sowohl simulierte Umgebungen, aber auch für echte Roboter eingesetzt werden, da beide Anwendungsfälle durch Programme an die Umgebung angebunden werden können.

-Alternative Ökosysteme mit gleichem Umfang wie ROS existieren nicht. -ROS2 -Andere (nur) Messagingsysteme -LCM -ZeroMQ

3.1.2 Beschreibung

ROS2[3], später auch einfach nur ROS genannt, beschreibt sich selbst als “a meta operating system for robots”[2]. Hierbei ist “operating system” nicht in seiner herkömmlichen Bedeutung eines vollständigen Betriebssystems zu verstehen. Es handelt sich dabei um eine gemeinsame Grundlage für Programme und Daten, welche durch ROS bereitgestellt wird.

Einzelne Bestandteile in der Umgebung sind dabei in Pakete gegliedert. Ein Paket kann beliebig viele Daten und Programme beinhalten, welche in zwei Dateien beschrieben werden. In CMakeLists.txt befinden sich Buildinstruktionen für den Compiler, falls sich Programme im Paket befinden. Außerdem können bestimmte Pfade aus dem Paket exportiert werden, sodass diese später im Workspace verfügbar sind. Programme, welche mit anderen Programmen in der Umgebung interagieren, werden in ROS “Nodes” genannt.

Zu den Aufgaben von ROS gehören dabei:

Buildumgebung

ROS benutzt colcon [1], um Pakete in den Workspaces reproduzierbar zu erstellen. Hierfür werden CMake und einige Erweiterungen, wie z.B. ament_cmake eingesetzt.

Workspaceverwaltung

Pakete können in verschiedenen Verzeichnissen installiert werden und müssen für andere Pakete auffindbar sein. ROS nutzt hierfür von colcon generierte Skripte, welche beim Erstellen eines Pakets und eines Workspaces mit angelegt werden. Das Skript des Pakets

fügt nur dieses Paket der Umgebung hinzu, das Skript des Workspaces führt alle Skripte der enthaltenen Pakete aus, um diese der Umgebung hinzuzufügen.

Abhängigkeitsverwaltung

ROS kann durch die in den Paketen deklarierten Abhängigkeiten prüfen, ob diese in der aktuellen Umgebung verfügbar sind. Dies vermeidet Abstürze und undefiniertes Verhalten in der Ausführung von Nodes.

Datenübertragung

Nodes müssen miteinander auf einem festgelegten Weg kommunizieren können, um beliebige Verbindungen dieser zu unterstützen. Dieser wird durch ROS in Form mehrerer Bibliotheken für unterschiedliche Sprachen bereitgestellt.

Parameterübergabe

Nodes benötigen häufig problemspezifische Konfiguration, um in vorher nicht bedachten Szenarien eingesetzt werden zu können. ROS stellt diese durch deklarierfähige und integrierte Argumente bereit.

Startverwaltung

In sogenannten “launch”-Files können verschiedene Nodes und andere “launch”-Files zu komplexen Startvorgängen zusammengefasst werden.

3.2 Simulationsumgebung (Gazebo)

3.2.1 Auswahl

Als Simulationsumgebung können verschiedene Programme genutzt werden, welche sich in ihrem Funktionsumfang stak unterscheiden. Hierfür kommen dedizierte Werkzeuge zur Robotersimulation, aber auch zum Beispiel universell einsetzbare Gameengines in Frage. Diese Werkzeuge müssen hierfür auf ihre Tauglichkeit für die gesetzte Aufgabe geprüft werden. Auch andere Aspekte sind hierbei zu betrachten, wie Lizenzen oder schwer bewertbare Aspekte wie Nutzerfreundlichkeit. Für die Auswahl kommen verschiedene Prgramme in Frage, welche im folgenden weiter beleuchtet werden.

CoppeliaSim, früher auch V-REP genannt, ist eine Robotersimulationsumgebung mit integriertem Editor und ROS-Unterstützung. Es unterstützt viele Sprachen (C/C++, Python, Java, Lua, Matlab oder Octave) zur Entwicklung von Erweiterungen des Simulators. Der Simulator selbst unterstützt Menschliche Aktoren, jedoch können diese nur Animationen abspielen oder zusammen mit Bewegungen abspielen. CoppeliaSim existiert in 3 Versionen, welche sich im Funktionsumfang unterscheiden, jedoch hat nur die professionelle Version Zugriff auf alle Funktionen und Verwendungsszenarien.

Gazebo Ignition ist wie CoppeliaSim eine Robotersimulationsumgebung, jedoch ohne integrierten Editor und direkte ROS-Unterstützung. Gazebo setzt wie CoppeliaSim auf Erweiterungen, welche die gewünschten Funktionen einbinden können. Zum Beispiel existiert auch eine ROS-Brücke, welche die Anbindung an ROS ermöglicht. Auch hier unterstützt der Simulator nur Animationen für menschliche Aktoren. Das Projekt ist Open Source, unter der Apache Lizenz (Version 2.0), was die Verwendung in jeglichen Szenarien erleichtert.

Unity hingegen ist primär eine Grafikengine für Nutzung in Computerspielen. Es existieren mehrere Systeme zur Anbindung der Engine an ROS, vor allem das offizielle "Robotics Simulation"-Paket und ZeroSim. Beide Systeme erlauben die Erweiterung der Gameengine um die Simulation von Robotern. Unity besitzt eine gute Dokumentation, die vor allem auf die Nutzung im Einsteigerbereich zurückzuführen ist. Auch die Optionen zur Menschensimulation sind gut, da diese häufig in Spielen verwendet werden. Ein großer Nachteil hingegen ist die Lizenz, welche nur für Einzelpersonen kostenlos ist.

Die Unreal Engine ist wie Unity eine Grafikengine aus dem Spielebereich. Auch hier ist die Menschensimulation aufgrund oben genannter Gründe gut möglich. Jedoch existiert für Unreal Engine keine offizielle Lösung zur Anbindung an ROS2. Die Programmierung der Engine erfolgt in C++, was einer Drittlösung erlaubte, eine ROS-Anbindung für Unreal Engine zu erstellen. Die Lizenz der Unreal Engine erlaubt die kostenfreie Nutzung bis zu einem gewissen Umsatz mit der erstellten Software.

Eine weitere Möglichkeit zur Simulation stellt die Grafikengine Godot dar. Im Vergleich zu Unity und Unreal Engine ist Godot quelloffene Software unter der MIT-Lizenz. Auch hier stellt die Simulation von menschlichen Aktoren eine Standartaufgabe dar, jedoch befinden sich Teile des dafür verwendeten Systems derzeit in Überarbeitung. Auch für diese Engine existiert eine ROS2-Anbindung, jedoch ist diese nicht offiziell.

Jede der drei Gameengines besitzt ein integriertes Physiksystem, welches die Simulation von starren Körpern und Gelenken erlaubt. Aus diesen Funktionen könnte ein Roboterarm aufgebaut werden, welcher dann durch eine der ROS-Brücken der Engines gesteuert werden kann.

Die Wahl der Simulationsumgebung fiel auf Gazebo Ignition, da dieser Simulator bereits im ROS-Framework etabliert ist. Dabei erlauben die offizielle ROS-Anbindung und offene Lizenz eine zuverlässige Verwendung in unterschidlichsten Szenarien.

3.2.2 Robotersimulation

Für die Robotersimulation wird ein Modell des Roboters benötigt, in welchem dieser für die Siimulationsumgebung beschrieben wird. Gazebo nutzt hierfür .srdf-Dateien, welche auf xml basieren. In diesen werden die einzelnen Glieder des Arms und die verbindenden Gelenke beschrieben.

Jedes Glied des Modells besitzt eine Masse, einen Masseschwerpunkt und eine Trägheitsmatrix für die Physiksimulation in Gazebo. Außerdem werden Modelle für die visuelle Repräsentation in Gazebo und die Kollisionserkennung in der Physiksimulation hinterlegt. Für beide existieren einfache Modelle wie Zylinder, Boxen und Kugeln. Da diese Formen nicht jeden Anwendungsfall abdecken und in der visuellen Repräsentation nicht ausreichen, können auch eigene Modelle hinterlegt werden.

Gelenke werden separat von den Gliedern definiert und verbinden jeweils zwei Glieder miteinander. Durch das Aneinanderreihen von mehreren Gliedern und Gelenken kann so jeder Roboteraufbau beschrieben werden. Jedes Gelenk besitzt eine Position und Rotation im Raum, um dessen Effekte je nach Typ des Gelenks berechnen zu können. Aspekte wie Reibung und Dämpfung können auch für die Physiksimulation angegeben werden. Folgende Typen von Gelenken können in urdf genutzt werden:

freie Gelenke ermöglichen vollständige Bewegung in allen 6 Freiheitsgraden. Sie stellen den normalen Zustand der Gelenke zueinander dar.

planare Gelenke erlauben Bewegungen senkrecht zur Achse des Gelenks. Sie werden für zum Beispiel Bodenkollisionen eingesetzt.

feste Gelenke sperren alle 6 Freiheitsgrade und werden häufig zur Plazierung von Objekten in einer Szene genutzt.

kontinuierliche Gelenke erlauben die beliebige Rotation um die Achse des Gelenks. Sie sind nur selten in rotierenden Gelenken mit Schleifkontakten oder anderen frei rotierbaren Übertragungsmechanismen zu finden.

drehbare Gelenke verhalten sich wie kontinuerliche Gelenke, haben jedoch minimale und maximale Auslenkungen. Sie sind die häufigste Art von Gelenken in Roboterarmen.

prismatische Gelenke ermöglichen die lineare Bewegung entlang der Achse des Gelenks. Denkbare Anwendungsfälle sind simulierte lineare Aktuatoren.

3.2.3 Menschensimulation

Gazebo besitzt bereits ein einfaches Animationssystem für bewegliche Aktoren, welches auch für Menschen nutzbar ist. Es existiert bereits ein Modell eines Menschen mit mehreren Animationen, welche allein abgespielt, oder an Bewegungen gekoppelt werden können. Dadurch ist eine Laufanimation realisierbar, welche synchronisiert zur Bewegung abgespielt wird.

Jedoch ist dies nur unter der Bedingung möglich, dass der gesamte Bewegungsablauf zum Simulationsstart bekannt ist. Dies ist auf die Definition der Pfade, welche die Bewegung auslösen,

zurückzuführen. Diese können nur in der .sdf-Datei des Aktoren definiert werden, was Veränderungen zur Laufzeit ausschließt. Durch diesen Umstand ist der mögliche Simulationsumfang nicht ausreichend.

Um dieses Problem zu beheben, ist die Entwicklung eines eigenen Systems zum Bewegen und Animieren des Menschen unausweichlich. Dieses System muss, wie im Konzept beschrieben, Steuerbefehle von außen empfangen, umsetzen und Feedback liefern können.

Ein solches System sollte als Gazebo-Plugin einbindbar sein, um Modifikationen an der Simulationsumgebung selbst auszuschließen, welche konstant weiter entwickelt wird. Dies erlaubt die einfachere Wartung, da bei Updates der Simulationsumgebung nicht die Menschensimulation an den neuen Code angepasst werden muss.

3.3 Roboterumgebung (MoveIt2)

MoveIt 2 ist das empfohlene ROS2 Paket für Bewegungsplanung von Robotern. Das System besteht aus meheren Komponmenten, welche in ihrer Gesamtheit den Bereich der Bewegungsplanung abdecken. Der Nutzer kann mit MoveIt auf mehreren Wegen Steuerbefehle für den Roboter absenden.

Die einfachste Art der Inbetriebnahme ist über das mitgelieferte RViz-Plugin und die demo-Launch-Files, welche durch den Setupassistenten für den Roboter generiert werden. Dort können Bewegungen durch das Bewegen von Markierungen oder in der Simulation geplant und ausgeführt werden.

Da sich ein solches System nur beschränkt zur Automatisierung durch Software eignet, existieren auch noch andere Schnitstellen. Für die Sprache Python existierte früher noch das moveit_commander Paket, welches den Zugriff auf MoveIt in Pyhon erlaubt, welches aber aktuell noch nicht portiert wurde. [4] Die direkte Nutzung der C++-API ist aktuell die einzige offizielle Möglichkeit, mit MoveIt auf einer abstrakteren Ebene zu interagieren. Natürlich können die Befehle auch direkt an die entsprechenden Topics gesendet werden um einzelne Bereiche des Systems zu testen, jedoch ist so kein einfacher Zugriff auf erweiterte Optionen möglich.

Durch diese Schnittstelle erhält die sogenannte MoveGroup ihre Informationen über die gewünschte Bewegung. Diese Daten können durch eine OccupancyMap ergänzt werden, welche die Bereiche beschreibt, welche sich um den Roboter befinden. Eine solche Erweiterung erlaubt die Nutzung von Kollisionsvermeidung mit Objekten im Planungsbereich.

Die Planung der Bewegung wird durch einen der zahlreichen implementierten Solver erledigt, welcher durch die MoveGroup aufgerufen wird. Um die generierte Bewegung umzusetzen, werden die gewünschten Gelenkpositionen als Abfolge an `ros_control` weitergegeben. Dabei

können sowohl echte Hardwaretreiber, aber auch simulierte Roboter genutzt werden. Der Erfolg der gesamten Pipeline kann dabei durch einen Feedbackmechanismus überwacht werden.

Im Falle von Gazebo wird `ign_ros_control` genutzt, welches die benötigten `ros_control` Controller in die Simulation einbindet. Diese können dann wie normale Controller von `ros_control` genutzt werden.

Dieser Ablauf ist auch im Anhang unter Abbildung 7.1 visualisiert.

3.4 Behavior Trees

Zur Verwaltung der Abläufe sollen BehaviorTrees genutzt werden, welche durch die Bibliothek `BehaviorTree.CPP` bereitgestellt werden. Diese Bibliothek wurde in C++ geschrieben, und ist somit einfach in ROS integrierbar.

Der einzige Nachteil ist der grafische Editor, welcher füher auch ein Open Source Projekt war, jetzt jedoch als closed source weitergeführt wird. Dieser muss jedoch nicht verwendet werden, um BehaviorTrees zu erstellen, da diese ein einfaches .xml-Dateiformat besitzen, welches gut dokumentiert ist.

Es existieren aber auch viele Beispiele und eine gute Dokumentation über die erweiterten Funktionen, welche im folgenden vorgestellt werden.

Asynchrone Nodes sind in `BehaviorTree.CPP` leichter umsetzbar, da diese im Lebenszyklus der Nodes beim Konzept der Bibliothek mit bedacht wurden. Dies resultiert in Nodes, welche ohne spezielle Logik langanhaltende Aktionen ausführen können, ohne die Ausführung des BehaviorTrees zu behindern.

Reaktives Verhalten ist ein neues Konzept, um die Handhabung von asnchronen Nodes zu vereinfachen. Diese Strukturelemente erlauben die parallele Ausführung von mehreren Zweigen, welche aktuell ausführende Aktionen beeinflussen können.

Das .xml-Format ermöglicht einen einfachen Austausch des Verhaltens, ohne die unterliegende Programmierung verändern zu müssen. Dies ist vor allem in kompillierten Sprachen wie C++ sinnvoll, da Änderungen im Verhaltensablauf keiner Neukompillierung bedürfen, was die Iterationszeit für Änderungen verbessert.

Plugins können zum Start geladen werden, um weitere Nodes dem ausgeführten Programm hinzufügen zu können. Dies vereinfacht die Erweiterung um neue Funktionen und das mehrfachen Nutzen von Code.

Datenfluss zwischen den Nodes erlaubt es, von außen konfigurierbares Verhalten der Nodes zu erreichen. Diese können dabei sowohl statisch, als auch dynamisch über sogenannte Ports mit Informationen versorgt werden.

Integriertes Logging erlaubt es, Zustandsänderungen im Behavior Tree zu visualisieren, aufzunehmen und wieder abzuspielen. Dies erleichtert das häufig schwierige Debuggen von Zustandsmaschienen erheblich, da das Verhalten genau untersucht werden kann.

BehaviorTrees werden in `BehaviorTree.CPP` als .xml-Dateien gespeichert. Diese Dateien enthalten die Anordnung der Nodes selbst, aber auch weitere Konfigurationsmöglichkeiten in Form von Ein- und Ausgabeports.

Solche Ports können verwendet werden, um Nodes generischer gestalten zu können. So kann auf feste Parameter in den Nodes selber verzichtet werden, was das erstellen mehrerer Nodes für ähnliche Funktionalitäten verhindert. Diese Daten können sowohl aus einem String ausgelesen werden, falls die ensprechende Funktion, welche diese in den Zieltyp übersetzt, implementiert wurde. Aber sie können auch aus dem sogenannten Blackboard entnommen werden.

Das Blackboard ist ein System, welches die Nutzung von Variablen als Parameter für Ein- und Ausgänge erlaubt. Diese werden im Hintergrund als eine Referenz auf den eigentlichen Wert gespeichert. Eine solche Funktion erlaubt das weitere Zerlegen von Vorgängen innerhalb des BehaviorTrees. Solche kleineren Nodes sind durch ihren limitierten Umfang universeller einsetzbar, da sie nur kleinere Teilprobleme betrachten, welche zu komplexeren Strukturen zusammengesetzt werden können.

Um die dadurch wachsenden Strukturen besser überblicken zu können, lassen sich Nodes als sogenannte SubTrees abspeichern. Diese bilden dann in ihrer Gesamtheit eine neue Node, welche im BehaviorTree eingesetzt werden kann. Um den Einsatz von Variablen innerhalb eines SubTrees zu ermöglichen, besitzt jeder SubTree ein separates Blackboard. Dadurch kann ein Eingriff in äußere Funktionalität verhindert werden.

Natürlich sollte es auch möglich sein, Variablen an solche SubTrees zu übergeben. Diese können, wie auch bei normalen Nodes, einfach als Parameter an den SubTree übergeben werden. Die Bibliothek `BehaviorTree.CPP` verbindet dann diese Werte und erlaubt die Datenübergabe zu und von dem SubTree.

3.4.1 Asynchrone Nodes

Da nicht jeder Prozess in einem einzigen Durchlauf des BehaviorTrees abgebildet werden kann, muss die Möglichkeit geschaffen werden, lang anhaltende Prozesse abzubilden. Dies geschieht in in `behaviortree_cpp_v3` durch asynchrone Nodes.

Eine asynchrone Node besitzt neben den Zuständen SUCCESS und FAILURE auch noch die beiden anderen Zustände RUNNING und IDLE.

Der Zustand RUNNING steht dabei für eine Node, welche sich noch in der Ausführung befindet. So lang dieser Zustand anhält, wird die Node nicht noch ein weiteres Mal gestartet, sondern nur der Zustand abgefragt.

Der IDLE-Zustand ist ein besonderer Zustand, welcher nur durch eine vollständige Ausführung erreichbar ist. Er wird von der Node angenommen, nachdem ein RUNNING Zustand durch entweder SUCCESS oder FAILURE beendet wurde und darf sonst nicht verwendet werden.

3.5 Docker-Compose als Virtualisierungsumgebung

Docker ist eine Virtualisierungsumgebung für Anwendungen, welche die komplette Umgebung für deren Ausführung bereitstellen. Dadurch wird die Inbetriebnahme von Anwendungen, welche spezielle Umgebungen für ihre Ausführung benötigen, ermöglicht.

Dies wird durch den Einsatz von sogenannten Containern erreicht, welche durch Buildfiles definiert werden. Ein Buildfile enthält exakte Instruktionen, wie der Container aus anderen Containern, Dateien oder einer Kombination beider erstellt werden kann. Die so erstellten Container können entweder lokal oder auf einem Server für die Verwendung bereitgehalten werden.

Ein solcher Container enthält ein eigenes Dateisystem, welches aus dem im Buildfile definierten Dateien und einem Overlay besteht. In diesem Overlay werden Änderungen gespeichert, welche am Container während der Laufzeit vorgenommen werden. Sofern nicht definiert, werden diese Änderungen beim Neustart des Containers wieder entfernt.

Um dies zu vermeiden, kann entweder ein Volume, eine Art virtuelles Laufwerk im `/var/lib/docker` Verzeichnis, oder ein "bind mount" in der compose.yml-Datei eingerichtet werden. Ein "bind mount" ist eine direkte Verbindung zu einem Ort des Host-Dateisystems, welche in den Container hereingereicht wird.

Docker-Compose stellt eine Erweiterung von Docker dar, welche die Inbetriebnahme der Container über ein spezielles Dateiformat verwaltet. In dieser Datei werden weitere Optionen angegeben, welche in die Umgebung des laufenden Containers eingreifen. Dazu gehört zum Beispiel das automatisierte Übergeben von Umgebungsvariablen, Einrichten von Netzwekumgebungen und Erstellen von Volumes und "bind mounts".

Diese Automatisierung erleichtert die initiale Einrichtung eines Containers auf einem neuen System, da alle benötigten Aspekte leicht angepasst werden können.

4 Umsetzung

4.1 Grundlegender Systemaufbau

Der grundlegende Systemaufbau musste stark modifiziert werden, wie in Abbildung 4.1 zu sehen, um die gesetzten Aufgaben erfüllen zu können. Dabei fallen vor allem die überarbeitete Kommunikation mit dem simulierten Menschen und die komplexere Steuerung des Roboterarms auf.

Die komplexere Steuerung des Roboters ist auf die Struktur von MoveIt zurückzuführen, welches in viele einzelne Teilmodule aufgeteilt ist. Diese müssen durch ihren modularen Aufbau, welcher für die Vielseitigkeit verantwortlich ist, einzeln konfiguriert werden, um miteinander in Interaktion treten zu können.

Außerdem musste die Kommunikation des Modells des Menschen überarbeitet werden, da die ROS-Kommunikation in Gazebo nur mit Einschränkungen möglich ist.

4.2 Verwendete Datentypen

In der Implementation werden unterschiedliche Datentypen verwendet.

geometry_msgs::msg::Pose ist einer der häufigsten Datentypen im Umgang mit Gazebo. Dieser Datentyp enthält die Information über die Lage und Rotation eines Objekts im Raum. Diese werden als relative Werte im Bezug auf das übergeornete Objekt gespeichert. Objekte wie der Mensch liegen in der Hierarchie der Simulation direkt unter der Welt, welche sich direkt im Nullpunkt und ohne Rotation befindet. Dadurch sind zum Beispiel die Koordinaten des Menschen absolut, da diese nicht durch die Welt verschoben und rotiert werden.

Area ist eine Datenstruktur mit einem Vektor an Positionen, welche eine Zone im 2-Dimensionalen Raum definieren. Jede Position ist eine einfache Datenstruktur aus 2 Gleitkommazahlen für je die X- und Y-Koordinate der Position. Der Verwendungszweck dieser Struktur ist die einfache Definition von Zonen, welche für Positionsgenerierungen und Postionsabfragen genutzt werden können.

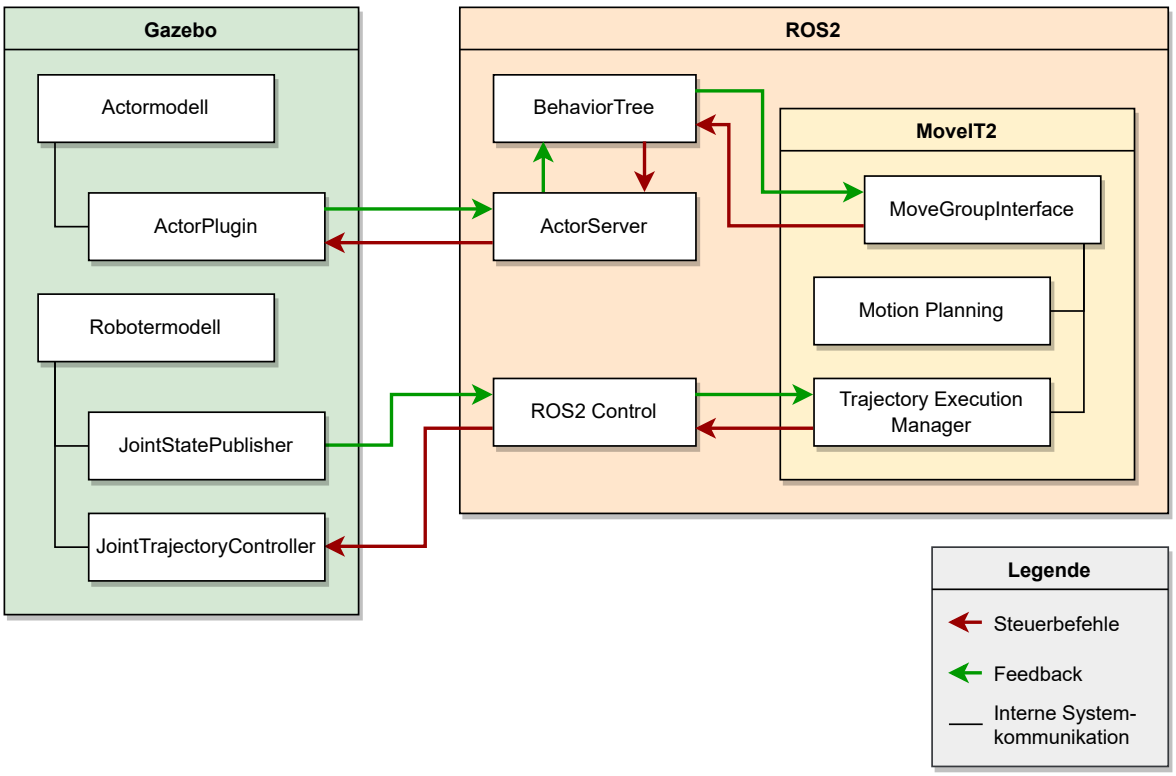

Abbildung 4.1: Visualisierung des überarbeiteten Konzepts

ActorPluginState definiert 4 Werte, welche das ActorPlugin annehmen kann. Diese 4 Werte sind SETUP, IDLE, MOVEMENT und ANIMATION.

FeedbackMessage beschreibt die erste der beiden MessageQueue-Nachrichten, welche vom ActorPlugin an den ActorServer gesendet wird. In dieser Struktur befindet sich der aktuelle Plugin-Zustand als `state`-Parameter vom Typ ActorPluginState. Außerdem ein `progress`-Parameter in Form einer Gleitkommazahl, welche den Fortschritt der aktuellen Aktion angibt. Um bei Bewegungen die aktuelle Position des Menschen zu erhalten, ist auch die aktuelle Pose des Modells im Parameter `current` enthalten.

ActionMessage ist die andere Nachricht, welche über die zweite MessageQueue vom ActorServer an das ActorPlugin gesendet wird. Wie in der FeedbackMessage ist ein `state`-Parameter vom selben Typ enthalten, jedoch dient dieser hier als Vorgabe für den nächsten State. Ein `animationName`-Parameter wird als ein char-Array mit einer maximalen Länge von 255 Zeichen übergeben. Dieser bestimmt später die Animation, welche je nach ActorPluginState während einer Bewegung oder Animation ausgeführt wird. Der `animationSpeed`-Parameter beschreibt entweder die Abspielgeschwindigkeit der Animation, oder die Distanz, welche pro Animationsdurchlauf zurückgelegt wird. Außerdem wird im Falle einer Bewegung der Parameter `target` vom Typ Pose verwendet, um die Endposition und Rotation des Actors zu bestimmen.

4.3 Mensch

4.3.1 Übersicht

Das angepasste Verfahren zur Menschensteuerung in der Simulation verwendet mehrere Kommunikationswege. Als erstes wird eine Bewegungs- oder Animationsanfrage an den ROS-Action-Server im ActorServer gesendet. Wenn die Simulation aktuell keinen Befehl ausführt, wird diese Anfrage akzeptiert, ansonsten wird sie abgebrochen. Daraufhin werden die Daten der Anfrage über eine Posix-Message-Queue vom ActorServer an das ActorPlugin in Gazebo gesendet.

Dieses verwendet die Daten, um eine interne State-Machine in den entsprechenden Zustand zu setzen, welcher zur Ausführung des Befehls benötigt wird.

Um Feedback an den Client des ROS-Action-Servers übertragen zu können, werden bei der Ausführung von Befehlen oder Zustandswechseln des ActorPlugins Feedbackdaten über eine separate MessageQueue zurück an den ActorServer übertragen. Diese werden durch den ActorServer aufbereitet, da nicht alle Daten für die jeweilige laufende Aktion relevant sind und an den ROS-Action-Client gesendet.

Um diese Befehle in der Simulation auch visuell umsetzen zu können, werden weitere Animationen für das Modell des Menschen benötigt, welche im Kontext der zur erfüllenden Aufgabe relevant sind. Dafür muss dan Modell in einen animierbaren Zustand gebracht werden, in welchem dann weitere Animationen erstellt und in die Simulation eingebunden werden können.

4.3.2 Modellierung

Um neue Animationen für den Menschen in der Simulation erstellen zu können, muss ein Modell für diesen erstellt werden. Dafür wurde eine der inkludierten Animationen in Blender geöffnet und das visuelle Modell kopiert.

Dieses Modell war auf Grund von vielen inneren Falten nur schlecht für Animationen geeignet, weshalb das Modell an diesen Stellen vereinfacht wurde. Eine solches Vorgehen beugt Anomalien bei der Animation durch unterschiedliche Verschiebung der Strukturen vor, welche vom inneren des Modells hervortreten können.

Nun musste das visuelle Modell mit neuen Animationsknochen versehen werden, da die in der Animation vorhandenen Knochen nicht verbunden sind. Diese Knochen bilden ein Skelett, welches zur Animation bewegt werden kann.

In Blender können sogenannte Constraints verwendet werden, um die Gelenke in Gruppen zusammenzufassen und genauer zu spezifizieren. Dazu wurde das Plugin “Rigify” verwendet, welches ein komplettes Skelett generiert und konfiguriert.

Dieses generierte Skelett kann nun an das Modell angepasst werden. Um eine bessere Übersicht zu ermöglichen, sollten als erstes alle nicht benötigten Skeletteile, wie zum Beispiel für Gesichtsanimationen, entfernt werden. Danach werden alle Knochen dem visuellen Modell angepasst. Dabei muss auf die Ausrichtung der Knochen zueinander geachtet werden. Das Kreuzprodukt der Vektoren beider Knochensegmente ist die Richtung der Beugeachse, welche sich im Verbindungspunkt beider Knochen befindet. Ist diese nicht richtig ausgerichtet, wenn zum Beispiel beide Knochen auf einer Gerade liegen, verbiegen sich Gelenke bei der Verwendung von inverser Kinematik zur Positionsvorgabe falsch.

Das hier erstellte, verbesserte Rigify-Skelett kann nun einfacher animiert werden. Dies liegt vor allem an neuen Markierungen in Blender, welche durch mehrere Constraints viele andere Knochen beeinflussen. Beispielswise können Fuß- und Handmarkierungen gesetzt werden, welche die Rotation des Fußes oder der Hand, aber auch gleichzeitig die inverse Kinematik des gesamten Beins oder Arms automatisieren. Selbiges gilt für neue Markierungen, welche zum Beispiel Hüft- und Kopfbewegungen vereinfachen.

Das Exportieren eines solchen Rigs ist jedoch schwierig, da viele Grafikengines keine verschachtelten Skelette verstehen. Dies ist auch der Grund, warum die Skelette beim initialen Import mitgelieferter Animationen nicht verbunden waren.

Um aus einem existierenden, vollständig verbundenen Skelett einzelne Knochen zu extrahieren exisitiert ein weiteres Plugin mit dem Namen “GameRig”. Dieses separiert die Knochen wieder, um die Animationen für Grafikengines exportieren zu können.

Nach dieser Veränderung kann die Animation als Collada-Animation exportiert werden. Dabei muss darauf geachtet werden, dass die vorwärts gerichtete Achse auf die richtige Achse gestellt ist. Außerdem ist es später einfacher, wenn nur eine Animation in jeder Datei vorhanden ist. Dies ist dem Fakt geschuldet, dass diese anderen Animationen zwar verfügbar sind, jedoch aber nur durch einen Index unterscheidbar sind, der von der Reihenfolge der exportierten Animationen abhängig ist.

4.3.3 Programmierung

Message Queue

Bei der Implementierung des ActorPlugins stellte sich heraus, dass die nun im ActorServer ausgelagerten Befehle mit den Befehlen im ros_control-Plugin kollidieren. Dies geschieht, da beide Plugins rclcpp, eine Bibliothek zur Kommunikation mit ROS, verwenden.

In dieser Bibliothek wird eine globale Instanz angelegt, welche den Zustand des Kommunikationsprotokolls abbildet. Da jedoch von beiden Plugins auf diesen Zustand zugegriffen wird, kommt es zur Problemen, da kein Synchronisationsmechanismus existiert. Die dadurch entstehenden gleichzeitigen Zugriffe auf die selben Ressourcen führen zur Terminierung des Programms.

Eine Anpassung beider Plugins auf die gemeinsame Nutzung einer Ressource ist möglich, erfordert jedoch weitere Anpassungen, welche zeitlich nur schwer planbar sind. Die Nutzung eines separaten Dienstes, welcher keinen globalen Kontext benötigt, ist die sicherste Lösung des Problems. Durch einen solchen Dienst werden auch in Zukunft keine Plugins gestört, auch wenn sie selbigen Dienst zur Kommunikation verwenden.

Die Auswahl eines Dienstes wurde dabei aus einer Reihe an unterschielichen Möglichkeiten getroffen. Eine REST-API hat den Vorteil, dass sie durch fast jede Programmiersprache genutzt werden kann, die Sockets unterstützt, hat jedoch keinen einheitlichen Feedbackmechanismus. Die neueren Websockets bieten die Möglichkeit, bidirektional Daten zu übertragen und erlauben somit Feedback an das aufrufende Programm. Beide Technologien basieren jedoch auf einem Webserver, welcher auf einem bestimmten Port des Systems ausgeführt werden muss, was Kollisionen mit anderen Serveices ermöglicht. Die Portnummer kann zwar geändert werden, ist jedoch nicht einfach mit einer Komponente assoziierbar, was sie zu einer “Magischen Zahl” macht. Dies sorgt für schlechte Lesbarkeit in einem wichtigen Teil des Kontrollflusses. Außerdem besitzen beide Terchnologien durch TCP oder UDP und HTTP relativ großen Protokolloverhead, welcher bei den hohen Updateraten der Gazebo-Simulation zu Problemen führen könnte.

Eine andere Möglichkeit ist die Nutzung von “shared memory”, einem geteilten Speicherbereich zwischen beiden Programmen. Dieser kann zur bidirektionalen Kommunikation genutzt werden, da beide Programme auf den Speicher zugreifen können. Alle Zugriffe auf den Bereich sind extrem schnell, was diese Technik ideal zur schnellen Datenübertragung zwischen Prozessen macht. Durch das erlauben gleichzeitiger Zugriffe kann es hierbei vorkommen, dass die selbe Adresse gleichzeitig von einem Programm gelesen und von einem anderen geschrieben wird. Die dabei gelesenen Daten können Schäden aufweisen, weswegen Zugriffe auf den Speicherbereich koordiniert werden müssen.

Die letzte betrachtete Methode ist die Verwendung einer Message Queue. Hier wird im Betriebssystem ein Speicherbereich mit bestimmter Größe für den Datenaustauch reserviert. Dieser Bereich besitzt ein Identifikationsmerkmal, mit welchem Anwendungen Zugriff auf diesen erlangen können. Ein Programm kann in diesem Bereich Nachrichten ablegen, welche durch das andere Programm gelesen werden können. Die Koordinierung der Zugriffe erfolgt dabei durch das Betriebssystem, was gleichzeitige Zugriffe, wie bei shared memory, aussschließt. Hierdurch kommt es zu einem Anstieg an Latenzzeit, jedoch ist dieser ausreichend gering.

Die Wahl des Dienstes fiel auf eine MessageQueue, jedoch existieren unter Linux 2 unabhängige Implementationen. Die erste Implementation ist die System V MessageQueue, und verwendet zur Identifikation einfache Integer. Eine Spezialität dieser alten Implementation ist das Sortieren der Nachrichten nach Nachrichtentyp in der gleichen Warteschlange. Die neuere Implementation der POSIX MessageQueue bietet einige weitere Funktionen, wie zum Beispiel aynchrone Benachrichtigungen bei neuen Nachrichten, Quality of Service und nutzt bis zu 256 Zeichen lange Strings zur Identifikation.

Die ausgewählte Implementation ist die neuere POSIX-Implementation einer Message Queue, da diese basierend auf den Erfahrungen mit der System V Implementation verbessert wurde.

Nachrichten

Die versendeten Nachrichten für den ActionServer, als auch für die Message Queue sind in den Paketen `ros_actor_action_server_msgs` und `ros_actor_message_queue_msgs` abgelegt. Die beiden ActionServer definieren jeweils 3 Nachrichten, welche eine Startnachricht, das Feedback während der Laufzeit und eine Endnachricht beschreiben.

In der Startnachricht werden alle Daten, welche der Server für die Bearbeitung einer Anfrage benötigt, übergeben. Dies geschieht, damit der Auftrag schon beim Start abgebrochen werden kann, sollte dieser nicht erfüllbar sein.

Die Feedbacknachrichten werden vom Server an den Client zurück gesendet, solange das Programm ausgeführt wird. Dabei ist es Aufgabe des Programms, diese in beliebigen Abständen an den Client zu senden.

Die Endnachricht kann Rückgabewerte für die ausgeführte Aufgabe enthalten, falls diese benötigt werden. Sie werden in diesem Projekt nicht genutzt, da das Beenden eines Auftrags immer mit einer einfachen Erfolgs- oder Misserfolgsmeldung quittiert wird.

ActorServer

Der ActorServer ist die Brücke zwischen ROS und dem ActorPlugin. Es werden zwei ActionServer angeboten, welche jeweils Bewegungen oder Animationen des simulierten Menschen auslösen können. Beide ActionServer prüfen zuerst, ob bereits eine andere Aktion ausgeführt wird. Sollte dies der Fall sein, wird die Anfrage abgelehnt. Im anderen Fall wird die Aufgabe akzeptiert und in das MessageQueue-Format übersetzt und an das ActorPlugin gesandt. Hier kommt es zu einer forcierten Warteschleife, welche die Bestätigung der Aktion vom ActorPlugin in der Feedback-Queue wartet, um das Starten mehrerer gleichzeitiger Aktionen zu unterbinden. Parallel werden alle eingehenden Feedback-Nachrichten der Message Queue des ActorPlugins in Feedback für die aktuell laufende Action umgewandelt.

Im Falle des Bewegungs-ActionServers werden mehrere Parameter benötigt. Zuerst werden Animationsname und -diztanz benötigt, um die richtige Animation auszuwählen und die Bewegung mit der Animation zu synchronisieren. Als Feedbacknachricht erhält der Client die aktuelle Pose des Actors im Simulationsraum.

Soll eine Animation über den Action Server abgespielt werden, wird auch hier ein Animationsname, jedoch auch eine Animationsgeschwindigkeit benötigt. Die Feedbacknachricht enthält den Fortschritt der Animation als Gleitkommazahl.

Gazebo Plugin

Das ActorPlugin nutzt die Daten aus der Message Queue für Befehle, um diese in der Simulation umzusetzen. Es werden dabei mehrere Zustände unterschieden.

Setup wird ausschließlich zu Simulationsbeginn verwendet, um alle benötigten Referenzen aus der Simualtionumgebung im Plugin zu hinerlegen, so dass diese in den anderen Zuständen genutzt werden können.

Movement bedeutet die Ausführung einer Bewegung in potentiell mehreren Schritten. Zuerst wird die Distanz zum Zielpunkt geprüft. Ist diese ausreichend gering, wird nur eine Bewegung in die gewünschte Endausrichtung durchgeführt. Ist diese größer, dreht sich der Actor in Richtung des Ziels und bewegt sich anschließend dorthin. Falls die gewünschte Endrotation nicht einem Null-Quaternion entspricht, wird anschließend noch eine Rotation in die Zielrichtung durchgeführt.

Animation entspricht der Ausführung einer Animation an der aktuellen Position des Actors. Diese kann durch einen Skalierungsfaktor beschleunigt oder verlangsamt werden.

Idle ist der Zustand, welcher nach erfolgreicher Ausführung eines Befehls angenommen wird.

Das ActorPlugin besitzt kein Konzept des ActionServers und verlässt sich auf den ActorServer, welcher die Feedbacknachrichten in das richtige Format bringt. Feedback wird in den Zuständen Movement und Animation in jedem Simulationsschritt gesendet. Um auch Zustandsübergänge erkennen zu können, werden auch diese als Feedback an den ActorServer gesendet.

4.4 Roboter

4.4.1 Übersicht

4.4.2 Modellierung

Für den Kuka LBR iisy existiert kein Simulationsmodell für Gazebo und ROS, weswegen dieses Modell aus Herstellerdaten generiert wurden. Die Maße und Form des Roboters wurden

aus einer .stl-Datei des Arms generiert.

Diese Datei wurde dazu in die unterschidlichen Glieder aufgeteilt, welche danach in Blender weiter bearbeitet wurden. Hiebei wurde die hohe Auflösung der Modelle reduziert, was sich in kleineren Dateien und Startzeiten der Simulation, aber auch in der Renderzeit der Simulation, auswirkt. Außerdem wuden die Glieder so ausgerichtet, dass der Verbindungspunkt zum vorherigen Glied im Nullpunkt des Koordinatensystems befindet.

Um die Simulation weiter zu beschleunigen, wurden die Kollisionsboxen des Arms noch weiter vereinfacht, was die Kollisionsüberprüfung dramatisch beschleunigt. Dabei werden stark simplifizierte Formen verwendet, welche das hochqualitative visuelle Modell mit einfachen Formen umfassen.

Diese Herangehensweise ist nötig, da Kollisionserkennung auf der CPU durchgeführt wird, welche durch komplexe Formen stark verlangsamt wird.

Um aus den generierten Gliedermodellen ein komplettes Robotermodell erstellen zu können, wurde eine .urdf-Datei erstellt. In dieser werden die verwendeten Gelenktypen zwischen den einzelnen Gliedern, aber auch deren Masse, maximale Geschwindigkeit, maximale Motorkraft, Reibung und Dämpfung hinterlegt. Diese Daten können später zur Simulation der Motoren genutzt werden, welche den Arm bewegen.

Die Gelenkpositionen sind dabei relative Angaben, welche sich auf das Glied beziehen, an welchem ein weiteres Glied über das Gelenk verbunden werden soll. Alle kontrollierbaren Gelenke benötigen auch eine Gelenkachse, welche je nach Gelenktyp die mögliche Beweglichkeit bestimmt.

Alle hier erstellten Dateien wurden im Paket `iisy_config` zusammengefasst, um diese einfacher wiederauffinden zu können.

4.4.3 MoveIt 2 Konfiguration

Das somit erstellte Paket kann nun mit dem neu implementierten MoveIt Configurator um die benötigten Kontrollstrukturen erweitert werden. Dazu wurde der neue Setupassistent von MoveIt2 verwendet, welcher das Modell analysiert, um es mit für MoveIt benötigten Parameter zu erweitert.

Die Erstellung des erweiterten Modells mit dem Assistenten funktionierte komplett fehlerfrei, jedoch ließen sich die generierten Dateien nicht nutzen.

Um die generierten Dateien nutzen zu können, mussten diese weiter angepasst werden. Dies beinhaltete die korrekte Integration der Roboterdefinitionen im Paket, aber auch zahlreiche Pfadreferenzen auf verwendete Dateien, welche nicht korrekt generiert wurden.

Das so erstellte Modell kann ber den Aufruf von `ros2 launch iisy_config demo.launch.py` in RViz getestet werden. Hierfür erscheint das Robotermodell mit mehreren Markern und Planungsoptionen, welche genutzt werden können, um beliebige Bewegungen zu planen und auszuführen.

4.4.4 Details

Das so erstellte Modell kann nun zur Laufzeit in Gazebo geladen werden. Dafür wird das Paket `ros_gz_sim` verwendet, welches das `create` Programm beinhaltet. Mit diesem Werkzeuk kann ein Modell unter einem bestimmten Namen anhand einer Datei oder eines übergebenen Strings in Gazebo importiert werden. Das Modell kann dabei über Argumente im Raum verschoben und rotiert werden, falls diese Funktionalität benötigt wird.

Im Modell sind auch die verwendeten Gazebo-Plugins deklariert, welche für die Integration mit dem `ros_control` verantwortlich sind.

4.5 Behavior Trees

Alle Behavior Trees wurden im Paket `btree_trees` organisert, welche durch das Paket `btree_nodes` gelesen werden.

Für die Umsetzung des Szenarios wurden neue Nodes für den BehaviorTree erstellt. Diese lassen sich nach Nutzung in verschiedene Gruppen einordnen.

Allgemein nutzbare Nodes

GenerateXYPose generiert eine Pose in einem durch den `area`-Parameter angegebenen Bereich. Um dies zu ermöglichen, wird zuerst die Fläche aller Dreiecke berechnet, welche den Bereich definieren. Diese werden durch den Gesamtinhalt geteilt, um eine Wichtung der Dreiecke zum Gesamtinhalt zu erreichen. Nun wird eine Zufallszahl zwischen 0 und 1 gebildet. Von dieser werden nun die Wichtungen der Dreiecke abgezogen, bis dies Zufallszahl im nächsten abzuziehenden Dreieck liegt. Nun wird in diesem Dreieck eine zufällige Position ermittelt, welche ber den Ausgabeparameter `pose` ausgegeben wird.

InAreaTest prüft, ob eine Pose, vorgegeben durch den `pose`-Parameter, in einer durch den `area`-Parameter definierten Zone liegt. Hierfür wird überprüft, ob die X und Y-Werte der Pose in einem der Dreiecke liegen, welche die Area definieren. Der Rückgabewert ist das Ergebnis dieser Überprüfung, wobei SUCCESS bedeutet, dass sich die Pose in der Area befindet.

OffsetPose wird genuzt, um eine Pose im Raum zu bewegen und/oder deren Orientierung zu verändern. Falls der `offset`-Parameter gesetzt ist, wird dieser mit dem `input`-Parameter summiert. Außerdem wird die Orientierung der Pose auf den `orientation`-Parameter gesetzt, falls dieser vorhanden ist, was den ursprünglichen Wert überschreibt.

InterruptableSequence stellt eine Sequence dar, welche auch nach ihrem Abbruch ihre Position behält. Dies ist von nöten, wenn bestimmtes Verhalten unterbrechbar ist, aber zu einem späteren Zeitpunkt fortgesetzt werden soll.

WeightedRandom ist eine Steuerungsnode, welche mehrere untergeordnete Nodes besitzt. Dabei werden diese nicht, wie bei anderen Steuerungsnodes üblich, sequentiell ausgeführt. Anhand einer vorgegebenen Wichtung im `weights`-Parameter wird eine der untergeordneten Nodes zufällig ausgewählt. Diese Node wird nun als einzige Node ausgeführt, bis diese den SUCCESS-Status zurück gibt. Nach dem dieser Status erreicht wurde, wird bei dem nächsten Durchlauf eine neue Node ausgewählt. Der Rückgabewert ist der Rückgabewert der ausgewählten untergeorneten Node.

IsCalled fragt den aktuellen Called-Status des Actors ab, welcher in einigen Szenarien vom Roboter verwendet wird, um den simulierten Menschen zu rufen. Der Rückgabewert der Node ist dabei SUCCESS, falls der Mensch gerufen wird, oder FAILURE, wenn kein RUF durchgeführt wird.

SetCalledTo setzt den aktuellen Called-Status auf den Wert des übergebenen `state`-Parameters. Da diese Aktion nicht fehlschlagen kann, liefert diese Node immer SUCCESS als Rückgabewert.

RandomFailure generiert eine Zufallszahl von 0 bis 1, welche mit dem `failure_chance`-Parameter verglichen wird. Der Rückgabewert ist das Ergebnis des Vergleichs, FAILURE, wenn die Zufallszahl kleiner als der `failure_chance`-Parameter ist, oder im anderen Falle SUCCESS.

Menschenspezifisch

ActorAnimation wird verwendet, um dem simulierten Menschen eine auszuführende Animation zu senden. Die Node benötigt zur Ausführung einen Animationsname, welcher im `animation_name`-Parameter angegeben wird. Ein optionaler `animation_speed`-Parameter gibt die Ausführungsgeschwindigkeit vor. Der Rückgabewert ist SUCCESS, wenn die komplette Ausführung gelang, oder FAILURE, falls diese abgebrochen oder abgelehnt wurde.

ActorMovement funktioniert wie eine ActorAnimation, sendet jedoch eine Bewegungsanfrage. Auch für diese wird ein Animationsname benötigt, welcher im `animation_name`-Parameter definiert wird. Jedoch wird für die Synchronisation zur Bewegung

ein Diztanzwert bewnötigt, welcher in einem Animationsdurchlauf zurückgelegt wied. Dieser wird im `animation_distance`-Parameter übergeben. Eine Zielpose wird im `target`-Parameter gesetzt. Eine Besonderheit dieses Paramerters ist die Verwendung des Null-Quaternions als Richtungsangabe, was die Endrotation auf die Laufrichtung setzt.

Roboterspezifisch

RobotMove gibt dem Roboter eine neue Zielposition über den `target`-Parameter vor. Bei dieser Node handelt es sich um eine asynchrone Node, welche nur bei der erfolgreiche Ausführung der Bewegung SUCCESS zurück gibt.

SetRobotVelocity setzt eine neue maximale Geschwindigkeit des Roboters, vorgegeben durch den `velocity`-Parameter. Der Rückgabewert ist immer SUCCESS.

Diese beiden roboterspezifischen Nodes beeinflussen im Hintergrund das gleiche System, da Bewegungen bei Geschwindigkeitsänderungen neu geplant und ausgeführt werden müssen. Dazu wird das letzte Ziel bis zu dessen erreichen vorgehalten, um die Ausführung neu zu starten, fall eine Geschwindigkeitsänderung durchgeführt werden muss. Um die RobotMove-Node nicht zu unterbrechen, wird diese nur über einen Callback über den Erfolg oder Misserfolg der gesamten Bewegung und nicht über den Erfolg einzelner Teilbewegungen informiert.

4.6 Docker-Compose

Um Docker für die Verwaltung einer ROS-Installation verwenden zu können, müssen einige Anpassungen vorgenommen werden. Da viele Anwendungen, unter anderem auch die Simulationsumgebung, eine Desktopumgebung benötigen, muss eine Zugriffsmöglichkeit geschaffen werden.

Diese Möglichkeiten können nicht durch Docker allein gelöst werden, da Befehle auf dem Hostsystem ausgeführt werden müssen, um die gewünschten Funktionen zu aktivieren. Um diese Modifikiationen trotzdem reproduzierbar zu machen, wurde ein Shellscript geschrieben, welches zum Starten des Containers verwendet wird.

Dieses Skript erstellt zuerst die benötigten Verzeichnisse für den Container, falls diese noch nicht existieren. Danach werden die SSH-Keys des Hosts in den Container kopiert, um eine SSH-Verbindung zu ermöglichen. Dieser Umweg über SSH ist nötig, da die benötigten Umgebungsvariablen für ROS sonst nicht in allen Fällen gesetzt werden können.

Außerdem werden die benötigten Zugriffe auf den lokalen X-Server durch den Container anhand dessen Hostname erlaubt. Diese Änderung erlaubt es dem Container, Fenster auf dem Desktop anzeigen zu können, solange die benötigten SysFS-Dateien hereingereicht werden.

Dies geschieht durch Einträge in der compose.yml-Datei, welche diese als “bind mount” in den Container hereinreicht.

Um Zugriff auf die Grafikbeschleunigung des Systems zu erhalten, muss deren Repräsentation im SysFS unter `/dev/dri` hineingereicht werden. Der Zugriff auf die Desktopumgebung, welcher bereits entsperrt wurde, wird nun durch das mounten von `/tmp/.X11-unix` erreicht. Dabei handelt es sich um den Unix-Socket des X11 Displayservers.

5 Szenarienbasierte Evaluation

5.1 Simulation des Menschen

-Animationen und Bewegungen funktionieren

-Kollisionen möglich, aber mit Animationen schwer zu synchronisieren

5.2 Bewegung des Roboters

-Roboter kann sich bewegen

-Kollisionen verfügbar

-Interaktion technisch möglich, nicht implementiert. (Kooperation MoveIt/Gazebo nötig)

5.3 BehaviorTrees

5.3.1 Nodes

-Nodes für implementierte Funktionen verfügbar

-Einige andere, technisch relevante Nodes auch implementiert

5.3.2 Kombinieren von Nodes zu einer Request

-MoveIt Planung benötigt mehrere Parameter, sollen aber in einzelnen Nodes gesetzt werden

-Kombination nötig, Beschreibung hier

6 Diskussion

6.1 Lessons Learned bei der Umsetzung

-Viele kleine Unannehmlichkeiten, kombiniert ein großes Problemen

-Dokumentation für spätere Projekte

6.1.1 Erstellung des Robotermodells

-Keine direkten technischen Daten zum Roboter von Kuka

-Nur CAD-Dateien der Außenhülle

-Schätzung von Spezifikationen anhand Marketingmaterial

6.1.2 Gazebo

Upgrade auf Ignition

Gazebo ist zu diesem Zeitpunkt in mehreren, teilweise gleichnamigen Versionen verfügbar, welche sich jedoch grundlegend unterscheiden. Ein altes Projekt mit dem früheren Namen Gazebo, welches nun in Gazebo Classic umbenannt wurde, die Neuimplementation der Simulationssoftware mit dem Namen Gazebo Ignition und die nächste Version, welche nur noch den Namen Gazebo tragen soll. Dies ist darauf zurückzuführen, dass Gazebo Classic und Ignition eine Zeit lang gleichzeitig existierten, jedoch unterschiedliche Funktionen und interne Schnittstellen besitzen.

Das Upgrade von Gazebo Classic auf Gazebo Ignition gestaltete sich aus mehreren Gründen schwierig. Dies ist am leichtesten an der geänderten Nutzeroberfläche sichtbar, welche in Ignition nun modular ausgeführt ist. Um dieses neue modulare System nutzen zu können, wurde das Dateiformat für die Simulationen angepasst. In diesem werden die benötigten Physikparameter, Nutzeroberflächen, Plugins und die Simulationswelt definiert.

Um alte Simulationsdateien zu migieren, empfiehlt sich die Erstellung einer neuen, leeren Welt in Gazebo Ignition, sodass alle benötigten Physik, Nuteroberflächen und Pluginreferenzen

erstellt werden. Danach kann die Welt Stück für Stück in das neue Format übertragen werden, wobei nach jedem Eintrag ein Test zur Funktionsfähigkeit gemacht werden sollte, da sich bestimmte Parameterdeklarationen geändert haben. Die Arbeit in kleine Stücke aufzuteilen ist hierbei hilfreich, da Fehler während des Ladevorgangs im Log nicht weiter beschrieben werden.

Auch das alte Plugin musste auf das neue System angepasst werden, was auch hier zu einer kompletten Neuimplementation führte, da die internen Systeme zu große Unterschiede aufwiesen. Darunter fallen eine grundlegende Strukturänderung der unterliegenden Engine, welche auf ein Entity-Component-System umstellte, aber auch sahlreiche kleinere Änderungen

Pluginarchitektur

-Plugins werden im selben Kontext geladen, verwendung von Librarys mit globalem Kontext schwer

Doppelte Messagedienste

-Duplizierung von Messagediensten in ROS und Gazebo

-Potentielle Lösung in einer Präsentation angekündigt (mal sehen, ob ich die wiederfinde)

Fehlende Animationsgrundlagen

-Animationen werden als .col bereitgestellt

-Enthalten Textur, Mesh und Bones, jedoch nicht verbunden -> schlecht für Animation

6.1.3 ROS2

Nachrichten und deren Echtzeitfähigkeit

-TCP und UDP verfügbar, jedoch nicht sofort kompatibel

Änderung der Compilertoolchain

-Änderung des Buildsystems von rosbuild auf catkin

-Benötigte Änderungen in CMakeFile und package

6.1.4 MoveIt2

Upgrade auf MoveIt2

-Unterschiede in der Deklaration des Roboters selbst

-Änderungen der Deklaration der ros_control Anbindung

-Vorerst kein Setup, manuelle Migration erforderlich

->Lesson learned: Projekte häufig auf Updates prüfen

Fehlerhafte Generierung der Roboter

-Einige Aspekte der Generierung nicht erforderlich oder falsch

Controller

-Controller benötigte Anpassungen und manuelle Tests

6.2 Lessons Learned bei den Szenarien

6.2.1 Debugging

-async tty Option

-gdb ist ein Muss

-“Aufhängen” von trees

7 Zusammenfassung und Ausblick

7.1 Ergebnisse

7.1.1 Graphische Repräsentation der Szenarien

-Szenarien graphisch abgebildet

7.1.2 Anpassung der Behavior Trees an Szenarien

-Trees angepasst

7.2 Diskussion

-Viele Iterationen benötigt

-Funktionstüchtige Simulation der Szenarien

-Bewegung der Objekte schwerer als gedacht

-Synchronisationsmechanismus Gazebo <-> MoveIt benötigt

7.3 Ausblick

7.3.1 Umsetzung in anderem Simulator

-Einfachere ROS Anbindung -Potentiell einfacher ausbaubar auf Basis einer erweiterbaren Gameengine -Mangelhafte Dokumentation von Gazebo

7.3.2 Simulation bewegter Objekte

-Aufgrund von Komplexität des Prozesses nicht integriert

7.3.3 Ergänzung von Umgebungserkennung

-MoveIt hat Unterstützung

-Daten aus Gazebo extrahieren und in MoveIt einbinden

-Person in OctoMap erkennen

7.3.4 Zusammenbringen von ActorPlugin und ActorServer

-Mechanismus für Datenaustausch zwischen ROS und Gazebo überdenken/überarbeiten

-Geteilte ROS Instanz zwischen Plugins? Wie?

-Potentielle integration von ROS als Messagedients in Gazebo

Literatur

[1] *colcon - collective construction*. letzter Zugriff: 02.04.2023. URL: https://colcon.readthedocs.io/en/released/.

[2] *GitHub - ros2/ros2: The Robot Operating System, is a meta operating system for robots.* letzter Zugriff: 02.04.2023. URL: https://github.com/ros2/ros2.

[3] Steven Macenski u. a. „Robot Operating System 2: Design, architecture, and uses in the wild“. In: *Science Robotics* 7.66 (2022), eabm6074. DOI: 10.1126/scirobotics.abm6074. URL: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074.

[4] *MoveIt Commander with ROS2 - Issue #337 - ros-planning/moveit2_tutorials*. letzter Zugriff: 10.04.2023. URL: https://github.com/ros-planning/moveit2_tutorials/issues/337.

[5] *moveit2_tutorials/moveit_pipeline.png at humble - ros-planning/moveit2_tutorials*. letzter Zugriff: 02.04.2023. URL: https://github.com/ros-planning/moveit2_tutorials/blob/humble/_static/images/moveit_pipeline.png.

[6] *Packages - /ros2/ubuntu/pool/main/ :: Oregon State University Open Source Lab*. letzter Zugriff: 10.04.2023. URL: http://packages.ros.org/ros2/ubuntu/pool/main/.

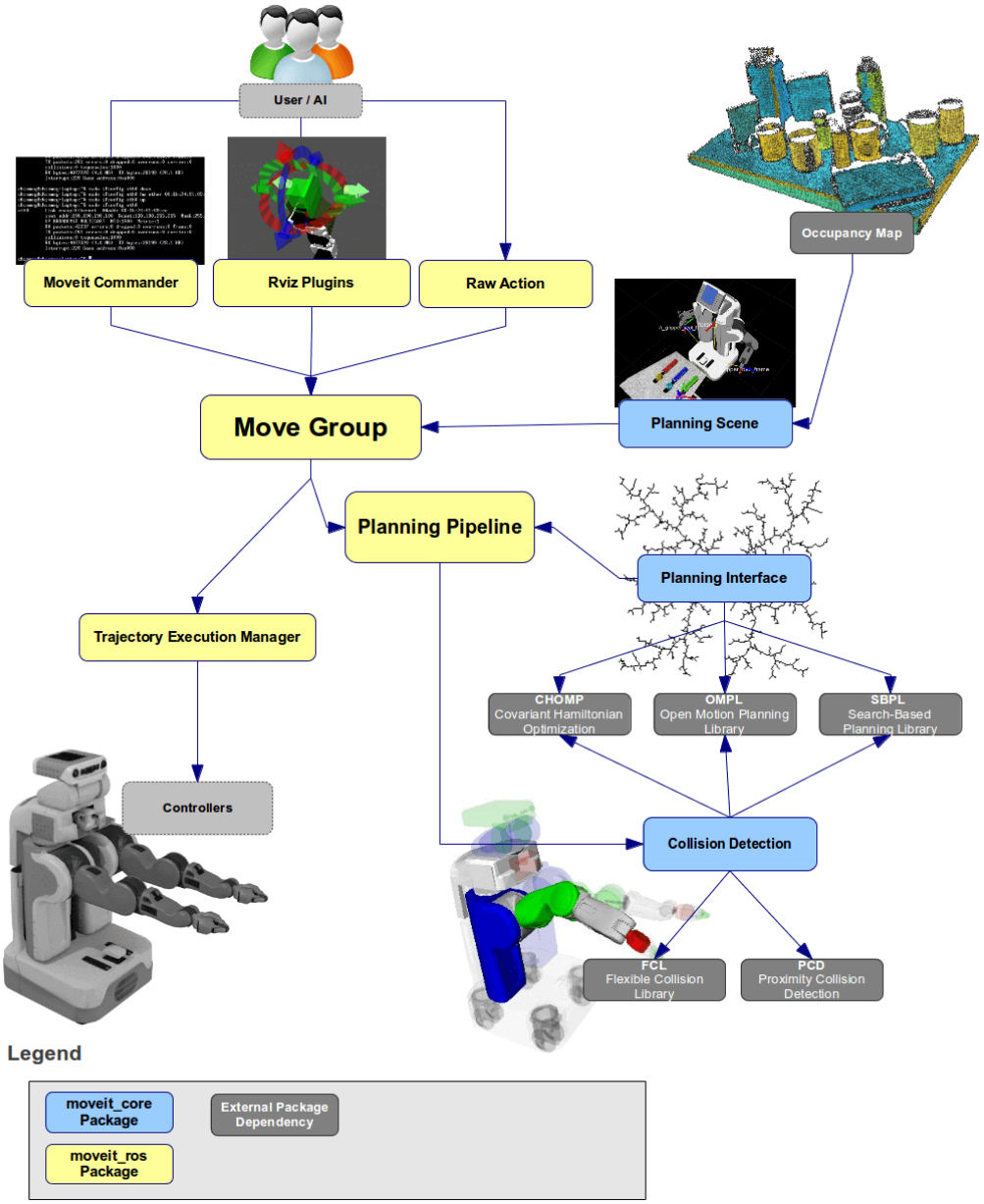

Abbildung 7.1: Visualisierung der MoveIt Pipeline[5]