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

Abbildungsverzeichnis

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

Die Simulation von Maschinen wird im industriellen Umfeld immer beliebter, um deren Verhalten schon vor der eigentlichen Produktion zu testen. Dazu wird ein Modell des gesamten Prozesses in der Simulation geschaffen, der durch die Maschine beeinflusst werden soll. Das Modell wird dann um die Maschine selbst erweitert, welche Einfluss auf das System nimmt. Die Veränderungen durch die Maschine werden nun analysiert, und erlauben Rückschlüsse auf die Funktion des Systems. Ein solches Modell kann nun für die Erkennung von Fehlverhalten und Problemen schon weit vor der eigentlichen Inbetriebnahme der Maschine genutzt werden.

Im wachsenden Feld der Mensch-Roboter-Kollaboration existieren bereits einige Lösungen, um auch die namensgebende Interaktion von Mensch und Roboter simulieren zu können. Eine häufige Einschränkung der Simulatoren ist dabei, dass der Bewegungsablauf in der Simulation bereits vor deren Ausführung fest definiert werden muss. Dies erlaubt die Reproduktion des genauen Bewegungsablaufes, aber nicht verschiedene Variationen im gesamten Prozess, welche durch die Ereignisse in der Simulation ausgelöst werden. Um eine solche Funktionalität bereitstellen zu können, muss der Bewegungsablauf von sowohl Roboter und Mensch zur Laufzeit der Simulation gesteuert werden.

Dies soll durch eine eingängliche Beschreibungssprache ermöglicht werden, die einfach erweitert und auf neue Szenarien angepasst werden kann. Um diese Funktionalität zu demonstrieren, sollen 3 unterschiedliche Testszenarien in der Simulationsumgebung abgebildet werden. Diese sollen durch verschiedene Aufgaben unterschiedliche Interaktionsgrade zwischen Mensch und Roboter simulieren.

1.2 Stand der Wissenschaft

Aktuelle wissenschaftliche Arbeiten befassen sich mit vielen unterschiedlichen Teilaspekten einer solchen Simulation.

Die Planung von unterschiedlichen Reaktionen von Roboter auf den Menschen in verschiedenen Interaktionsszenarien stellt eine Grundlage für spätere Projekte dar.[8] Hierbei wird die erwünschte Interaktion betrachtet und aus den gewonnenen Daten werden Einschränkungen

generiert. Diese Einschränkungen können nun in der Interaktion verwendet werden, um Verletzungen durch den Roboter auszuschließen.

Ein anderer Weg der Kollisionsvermeidung ist die Planung der maximal zurücklegbaren Distanz eines Menschen aus seiner aktuellen Position.[7] Dafür werden die maximalen Beschleunigungen einzelner Körperteile ermittelt, um diese während der Interaktion überwachen zu können. Sollte ein Mensch den Roboter erreichen können, muss dieser in der dafür benötigten Zeit stoppen können. Dies sorgt für eine Geschwindigkeitsanpassung im Nahfeld, da hier schnellere Bewegungen möglich sind.

Es existieren auch zahlreiche Wege, die Bewegungs- und Interaktionsplanung eines Roboters mit Beschreibungssprachen abzudecken.[6] Dabei kommt es auf die Umsetzung in der Beschreibungssprache, aber auch auf Anforderungen an das System an, um zum Beispiel das Abbrechen von Aktionen zu ermöglichen.

In Computerspielen werden Beschreibungssprachen schon seit langer Zeit eingesetzt, um verschiedene Systeme, wie zum Beispiel die Steuerung von Nichtspielercharakteren, zu beschreiben.[12]

Eine vollständige Umsetzung einer erweiterbaren Simulation mit Mensch und Roboter, gesteuert durch eine Beschreibungssprache konnte nicht gefunden werden.

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 Umstände denkbar sind, in welchen diese verwendet werden können. Dazu zählen zum Beispiel das Hineingreifen in einen Prozess, das Aufheben von Material oder das Begutachten eines Objekts, welche alle nur eine Bewegungsabfolge des beteiligten Menschen sind.

Das erste Szenario soll sich mit der Simulation einer bereits vollautomatisierten Fertigungsaufgabe befassen, 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. Der Mensch soll in diesem Szenario auch arbeiten, jedoch nur vereinzelt den Arbeitsbereich des Roboters betreten, um diesen zu beobachten.

Dieses Szenario ist ein Beispiel für eine Koexistenz zwischen Roboter und Mensch, wo beide an unterschiedlichen Aufgaben, 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. Auch hier soll der Mensch solange eigenständig arbeiten, bis der Roboter ein aussortiertes Teil ablegt, welches weiter transportiert werden muss.

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. Dieser soll, wenn seine Hilfe nicht benötigt wird, andere Roboter kontrollieren, was durch das Laufen im Raum abgebildet werden soll.

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.

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.

Hierfür wird die Beschreibungssprache in der Dienstumgebung ausgeführt, in welcher auch die Bewegungsplanung stattfindet. Die Beschreibungssprache kommuniziert direkt mit der Simulation und Bewegungsplanung des simulierten Menschen. Damit der Roboter in der Simulation von der Bewegungsplanung gesteuert werden kann, werden zwischen diesen auch Nachrichten ausgetauscht. Der gesamte Vorgang ist in Abbildung 2.1 visualisiert.

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 aufzubauen.

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

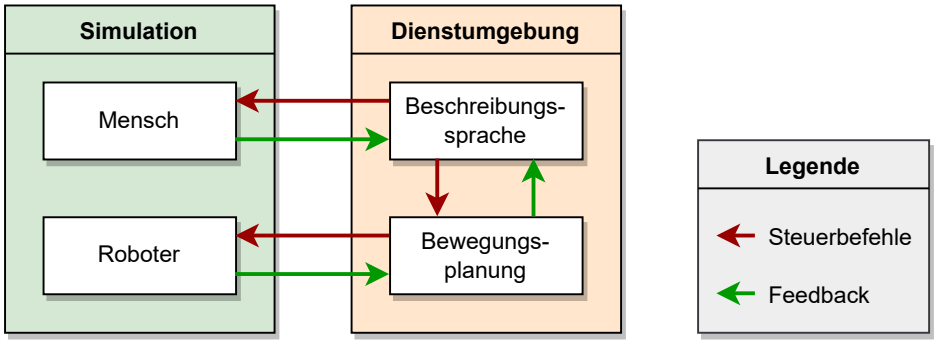

Abbildung 2.1: Visualisierung des Konzepts

Kriterien eignet sich der KUKA LBR iisy, welcher als Cobot vermarktet wird. Die Bezeichnung als Cobot[4] ist dabei ein Portemanteau aus Collaborative und Robot, was die besondere Eignung für Mensch-Roboter-Kollaboration noch einmal unterstreicht. Der Roboter besitzt auch einen modifizierbaren Endeffektor, um unterschiedlichste Aufgaben erfüllen zu können.

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 abbilden. 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 sollen in der Simulationsumgebung mehrere Animationen verwendet werden, 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 in eine bestimmte Richtung” auszuführen.

Um diese Animationen erstellen zu können, wird zuerst ein animierbares Modell des Menschen benötigt. Dieses Modell soll dabei um weitere Animationen erweiterbar sein. Es werden mehrere Animationen und Übergänge zwischen diesen benötigt, um bestimmte Bewegungen darstellen zu können.

Die so erstellten Animationen müssen nun von außerhalb der Simulationsumgebung ausführbar gemacht werden, um diese später mit einer Beschreibungssprache steuern zu können. Hierfür muss eine Komponente entwickelt werden, welche in der Simulation die Anfragen der Beschreibungssprache entgegennimmt und umsetzt. Um die spätere Steuerung des Menschen von außerhalb zu erleichtern, müssen diese Anfragen im Fortschritt überwacht und abgebrochen werden können.

2.3 Behavior Trees als Beschreibungssprache

Bislang wird das Verhalten von Akteuren in Simationsumgebungen, darunter Roboter und Menschen, häufig in Form von State-Machines ausgedrückt. Diese besitzen jedoch einen großen Nachteil, welcher vor allem bei komplexeren Abläufen hervortreten kann. Dabei handelt es sich um die Übersichtlichkeit, welche bei einer wachsenden State-Machine leicht verloren geht. Dies erschwert die schnelle Erfassung von Abfolgen und Zustandsübergängen bei Änderungen am Code, was zu Fehlern im späteren Betrieb führen kann.

Behavior Trees lösen dieses Problem, in dem sie sogenannte Nodes definieren, welche in einer übersichtlichen Baumstruktur angeordnet werden. Die einzelnen Nodes verändern dabei das System und lösen den Wechsel zu neuen Nodes aus.

Ursprünglich wurde das Konzept von Rodney Brooks entwickelt, welcher diese für mobile Roboter einsetzen wollte. [2] Das System setzte sich jedoch erst später in der Spieleindustrie, für die Beschreibung von menschlichem Verhalten durch. [15]

Der Ablauf eines Behavior Trees startet vom sogenannten Root, der Wurzel des Baums. Von dort an werden Nodes, welche je nach Node unterschiedliches Verhalten abbilden, miteinander verbunden. Die Nodes werden untereinander angeordnet, welches die Relation der Nodes zueinander beschreibt. Jede Node ist entweder direkt unter der Root-Node oder einer anderen Node angeordnet. Außerdem kann jede Node eine beliebige Anzahl an untergeordneten Nodes besitzen. Es gibt mehrere grundlegende Arten von Tree-Nodes, die in vier Kategorien unterschieden werden können.

Aktions-Nodes beschreiben einzelne ausführbare Aktionen, die das System beeinflussen können. Jede Aktion liefert dabei einen Rückgabewert über ihren Erfolg, welcher durch die darüber liegende Node ausgewertet werden kann.

Dekorations-Nodes können den Rückgabewert einer anderen Node modifizieren. Häufig existieren hier die Negation, aber auch das forcieren eines bestimmten Rückgabewertes für die unterliegende Node.

Sequenz-Nodes beschreiben eine nacheinander ausgeführte Abfolge von darunter liegenden Nodes. Sollte eine Node einen Fehler zurückgeben, wird die Ausführung der Sequenz abgebrochen, was wiederum zur Rückgabe eines Fehlers durch die Sequenz selbst führt. Beim erfolgreichen Durchlauf gibt die Sequenz einen Erfolg zurück.

Fallback-Nodes verhalten sich ähnlich zu Sequenz-Nodes, jedoch werden darunter liegenden Nodes nacheinander ausgeführt, bis eine Node Erfolg zurück gibt. In diesem Fall wird die Ausführung der Fallback-Node mit einem Erfolg abgebrochen. Ein Fehler wird hier zurückgegeben, wenn alle Nodes ohne eine Erfolgsmeldung durchlaufen wurden.

Das in Abbildung 2.2 visualisierte Beispiel zeigt die Abfolge, um eine Tür zu öffnen und zu durchschreiten. Die Ausführung des Baumes beginnt an der Root-Node. Von dort an wird als erstes die Sequenz-Node ausgeführt, welche drei untergeordnete Nodes besitzt.

Diese drei Nodes werden in Leserichtung, in diesem Falle von links nach rechts, ausgeführt. Daraus resultierend wird als erstes die linke Node ausgeführt, diese ist jedoch eine Fallback-Node mit weiteren untergeordneten Nodes. Da der Rückgabewert der Fallback-Node von den in ihr befindlichen Nodes bestimmt wird, werden diese nun nacheinander ausgeführt. Hier gelten die oben genannten Regeln für Fallback-Nodes. In diesem Fall wird geprüft, ob die Tür bereits offen ist. Da dies nicht der Fall ist, wird die nächste Node ausgeführt, welche die

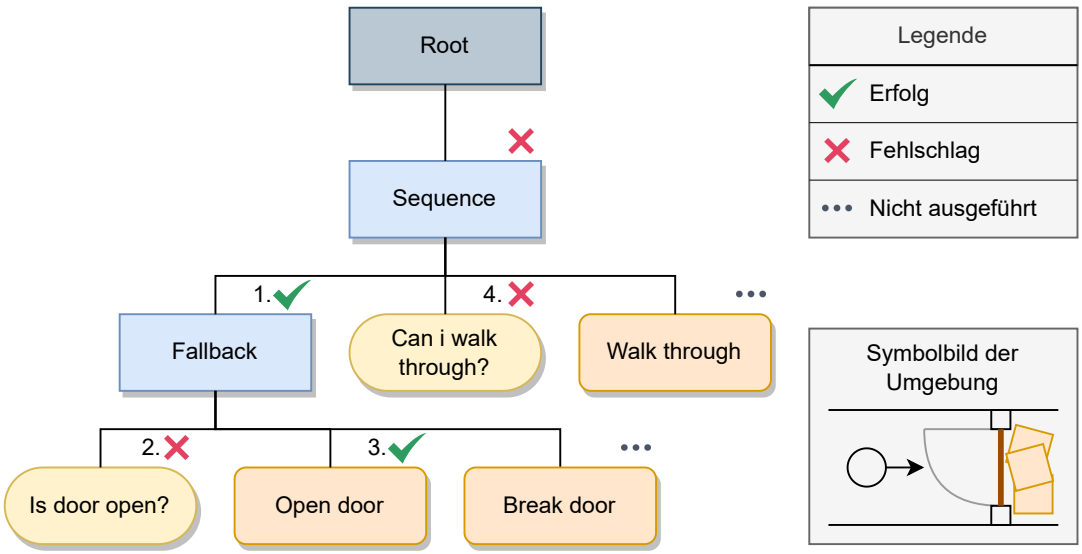

Abbildung 2.2: Beispiel eines BehaviorTrees

Tür öffnen soll. Dieser Versuch gelingt, weshalb die Ausführung der Fallback-Node mit einem Erfolg beendet wird.

Dadurch wird die nächste Node in der Sequenz ausgeführt, welche prüft, ob die Tür durchlaufen werden kann. Da dies nicht möglich ist, da die Tür zwar offen, aber durch dahinter liegende Gegenstände blockiert ist, wird ein Misserfolg gemeldet, welcher die Ausführung der Sequenz abbricht. Diese würde nun von neuem beginnen, da die Root-Node ihre untergeornete Node ohne Berücksichtigung des Rückgabewertes neu startet.

Durch die Definition neuer Nodes und einer anderen Baumstruktur lassen sich so einfach neue Verhalten implementieren. Dies erlaubt die schnelle Anpassung des Verhaltens der gesteuerten Systeme.

In dieser Arbeit sollen deshalb 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 betrachtet werden, abzudecken.

2.4 Virtualisierungsumgebung als Platform

Aufgrund von vorheriger Erfahrung mit involvierten Einrichtungsprozessen, ist der Einsatz fest definierter Umgebungen unerlässlich. Dies kann durch den Einsatz einer Virtualisierungsumgebung geschehen, in der das zu entwerfende System ausgeführt wird.

Dadurch können benötigte Programme, Pfade und Umgebungsvariablen in der Virtualisierungsumgebung hinterlegt werden, welche diese bei der Ausführung auf einem anderen Grundsystem korrekt abbildet. Eine solche Struktur erhöht die Zuverlässigkeit der Umgebung, da alle Änderungen an der Umgebung auf alle ausführenden Systeme gespiegelt werden.

Ein weiterer Vorteil ist die beschleunigte Entwicklung, da Änderungen nicht mehr an einzelne Zielsysteme angepasst werden müssen. Hinzu kommt die einfachere Inbetriebnahme eines bereits entwickelten Systems, da keine Anpassungen am Hostsystem vorgenommen werden müssen.

Natürlich existieren auch Nachteile der Virtualisierung, welche mit den Vorteilen abgewogen werden müssen. Alle Virtualisierungssysteme benötigen zusätzliche Rechenleistung, welche der virtualisierten Anwendung nicht mehr zur Verfügung steht. Außerdem muss bei grafischen Systemen bedacht werden, wie die darzustellenden Daten vom Hostsystem angezeigt werden können.

Die Auswahl einer für das Projekt geeigneten Virtualisierungsumgebung stellt einen wichtigen Schritt in der Entwicklung des Gesamtsystems dar.

3 Komponenten-/Softwareauswahl

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

Wie bereits in Abbildung 2.1 dargestellt, werden für die Erfüllung der Aufgaben eine Dienst- und eine Simulationsumgebung benötigt, die miteinander interagieren. Dieses Kapitel beschreibt die Auswahl der Umgebungen und der in diesen laufenden Prozesse.

3.1 Dienstumgebung

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

Bei einer Dienstumgebung für Roboter gehören zu den grundlegenden Aspekten die Nachrichtenübergabe zwischen einzelen interagierenden Programmen, um eine gemeinsame Basis für ein einfach erweiterbares System zu schaffen. Außerdem sind Werkzeuge zur Einstellungsübergabe an Teilsysteme sinnvoll, um diese einfach an einer Stelle anpassen zu können.

3.1.1 Auswahl

Es existieren mehrere Systeme, die als Dienstumgebung für Roboter in Frage kommen, wenn es lediglich 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 Interprozesskommunikation 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 mehreren Programmen austauschen können. Dabei dient ein Programm als Server, welches Anfragen von anderen Programmen, auch Clients genannt, entgegen nimmt. Die Kommunikation zwischen Client und Server ist bidirektional möglich, was kompliziertere Protokolle ermöglicht.

Alle diese Lösungen besitzen einen gemeinsamen Nachteil in deren Nachrichtendefinition. Dieser Nachteil besteht in der potentiellen Variation zahlreicher Kommunikationsmechanismen, welche zum Datenaustausch über all diese möglichen Systeme verwendet werden können. Bei einem ausreichend großen Projekt treten so unweigerlich Unterschiede in der Handhabung auf, die es zu berücksichtigen gilt.

In diesem Bereich sticht ROS[16] als Dienstumgebung für Roboter hervor, da es sich um ein etabliertes und quelloffenes System handelt. Der oben genannte Nachteil einzelner Systeme wird in ROS durch mehrere standartisierte und erweiterbare Nachrichtendefinition gelöst, welche von den Programmen in der Umgebung genutzt werden. Um diese Nachrichten zu senden und empfangen liefert ROS eine eigene Implementation des Protokolls für zum Beispiel Python[29] und C++[25], welche mehrere Arten der Nachrichtenübergabe unterstützten.

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 zwischenspeichern und über sowohl TCP, als auch UDP kommunizieren. Außerdem werden nun neben CMake[3] auch andere Buildsysteme unterstützt, was zum Beispiel die Verwendung von Python erlaubt.

Generell existieren im Feld der Roboter-Dienstumgebungen keine freien 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.[21]

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

3.1.2 Beschreibung

ROS2[16], später auch einfach nur ROS genannt, beschreibt sich selbst als “a meta operating system for robots”[13]. 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, wobei jedes Paket beliebig viele Daten und Programme beinhalten kann.

Jedes Paket enthält dabei eine `package.xml`-Datei, welche durch die Paketverwaltung genutzt wird. In dieser befindet sich eine in XML verfasste Definition des Paketinhalts, welche verschiedene Aspekte des Pakets beschreibt.

Darunter fallen Informationen über den das Paket selbt, wie dessen Name, Beschreibung und Version. Außerdem sind Name und Mailadresse des Autors, sowie die gewählte Lizenz des Codes vermerkt. Um das Paket später korrekt ausführen zu können, werden außerdem benötigte Pakete für die Erstellung und Ausführung des Pakets eingetragen.

Falls C++ zur Entwicklung des Pakets verwendet wird, befindet sich außerdem eine `CMakeLists.txt`-Datei im Paket. In befinden sich die Buildinstruktionen für den Compiler für die im Paket befindlichen Programme. Durch einen Aufruf von `ament_cmake` werden andere Parameter aus der `package.xml`-Datei ergänzt.

Außerdem können bestimmte Pfade aus dem Paket exportiert werden, sodass diese später im Workspace verfügbar sind. Programme, welche mit anderen Programmen in der Umgebung über die ROS internen Schnittstellen kommunizieren, werden in ROS “Nodes” genannt.

Zu den Aufgaben von ROS gehören dabei:

Buildumgebung

ROS nutzt die eigene Buildumgebung `colcon` [5], um Pakete in den Workspaces reproduzierbar zu erstellen. Zu deren Konfiguration werden CMake[3] und einige Erweiterungen, wie zum Beispiel das oben erwähnte `ament_cmake` eingesetzt. Anhand dieser Dateien werden die fertiggestellten Pakete dann im Workspace installiert, damit diese später verwendet werden können.

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 in diesem enthaltenen Pakete aus, um diese allesamt der Umgebung hinzuzufügen.

Abhängigkeitsverwaltung

ROS kann durch die in den Paketen deklarierten Abhängigkeiten prüfen, ob diese in der aktuellen Umgebung ausführbar sind. Die generierten Warnungen bei fehlenden Paketen vermeiden Abstürze und undefiniertes Verhalten in der Ausführung von Nodes, welche diese benötigen.

Datenübertragung

Um Daten zwischen Nodes austauschen zu können, müssen miteinander auf einem festgelegten Weg kommunizieren können. Ein solcher Aufbau erlaubt den Austausch von Daten in vorher nicht durch die Entwickler bedachten Anwendungsfällen. Die von ROS

gebotene Schnittstelle zur Datenübertragung wird in Form mehrerer Bibliotheken für unterschiedliche Programmiersprachen bereitgestellt.

Der Datenaustausch geschieht in sogenannten Topics, welche Nachrichtenkanäle zwischen den Nodes darstellen. Eine Node kann entweder als Server ein Topic anbieten, was durch andere Nodes gelesen wird, oder als Client auf solche bereitgestellten Topics zugreifen. Durch die Kombination mehrerer Topics ist eine komplexere Interaktion abbildbar, welche zum Beispiel Rückgabewerte zum aktuell ausgeführten Steuerbefehl liefert.

Parameterübergabe

Um Nodes auf neue Anwendungsfälle anpassen zu können, wird ein Konfigurationsmechanismus benötigt. In ROS geschieht dies durch die Übergabe sogenannter Parameter, welche durch die Node gelesen werden können. Diese eignen sich zur Übergabe von Informationen, wie zum Beispiel einem Robotermodell, aber auch um die Topics der Nodes umbenennen zu können.

Startverwaltung

In sogenannten “launch”-Files können verschiedene Nodes und andere “launch”-Files zu komplexen Startvorgängen zusammengefasst werden. Dabei werden diese mit den gewünschten Parametern versehen, welche verschiedene Eigenschaften der Node an die aktuelle Umgebung anpassen können.

Durch eine solche Umgebung kann die gewünschte Simulation einfacher in mehrere Komponenten zerlegt werden, was die spätere Wartung des Projekts vereinfacht.

3.2 Simulationsumgebung (Gazebo)

3.2.1 Auswahl

Als Simulationsumgebung eignen sich verschiedenen Programme, die sich hinsichtlich ihres Funktionsumfangs stark unterscheiden. Hierfür kommen dedizierte Werkzeuge zur Robotersimulation, aber auch beispielsweise universell einsetzbare Gameengines in Frage. Ein Vergleich dieser Werkzeuge ist hierbei sinnvoll, da der gebotene Funktionsumfang der Softwares sich stark unterscheidet. Auch andere Aspekte, wie Lizenzen oder schwer bewertbare Aspekte wie Nutzerfreundlichkeit, sind hierbei zu betrachten. Eine Auswahl der als Simulationsumgebung in Frage kommenden Programme werden hier vorgestellt.

CoppeliaSim[23], 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[10] 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[27] 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[26] 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 die Erstellung eines Plugins zur ROS-Anbindung der Unreal Engine ermöglichte, welche von Nutzern gewartet wird. 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[14] 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.

Alle vorgestellten Softwares besitzten ein integriertes Physiksystem, welches die Simulation von starren Körpern und Gelenken erlaubt. Aus diesen Funktionen kann ein Roboterarm aufgebaut werden, welcher dann durch eine ROS-Brücke gesteuert werden kann.

Um den Entwicklungsvorgang zu beschleunigen, ist die Auswahl einer Umgebung mit bereits existierender ROS-Unterstützung sinnvoll. Durch diese Einschränkung scheiden sowohl Unreal Engine aber auch Godot aus, welche nur durch Nutzer unterstützt werden. Für einen späteren Einsatz ist eine offene Lizenz von Vorteil, da diese in nahezu allen Umständen eingesetzt werden kann.

Die Wahl der Simulationsumgebung fiel deshalb auf Gazebo Ignition, welches gleichzeitig bereits im ROS-Ökosystem etabliert ist. Dabei erlauben die offizielle ROS-Anbindung und offene Lizenz eine zuverlässige Verwendung in unterschiedlichsten Szenarien.

3.2.2 Welt- und Modellbeschreibung

Um die Simulationsumgebung zu beschreiben, nutzt Gazebo das .sdf-Dateiformat[24]. Dieses Format basiert auf XML und wird zur Definition gesamter Welten, aber auch einzelner Objekte innerhalb dieser Welten benutzt.

Um verschiedene Versionen des Formats zu unterstützen, enthält das einzige sdf-Element die gewünschte Versionsnummer. Eine solche Datei kann, wie bereits oben beschrieben, unterschiedliche Daten enthalten. Im Falle eines Objekts ist dies eine einzige Instanz von entweder einem Modell, Actor oder Licht. Andernfalls können in der Datei eine oder mehrere Welten definiert werden.

Eine Welt definiert in Gazebo den kompletten Aubau des Simulators. Zuerst enthält ein Welt-Element die Daten über die physikalischen Konstanten der Simulationsumgebung. Außerdem werden alle benötigten Teile der Nutzeroberfläche deklariert, welche im ausgeführten Simulator verfügbar sein sollen. Letzendlich können auch mehrere Modelle, Aktoren und Lichter in der Welt definiert werden. Diese können auch aus anderen URIs stammen, welche in der Welt deklariert wurden. Dies erlaubt das Laden von zum Beispiel vorher definierten Objekten oder Objekten aus der offiziellen Bibliothek[11].

Ein Modell enthält einen Roboter oder ein anderes physikalisches Objekt in der Simulation. Die Deklaration eines weiteren Untermodells ist möglich, um komplexere Strukturen abbilden zu können. Über ein include-Element können auch andere .sdf-Dateien, welche nur ein Modell enthalten, zum Modell hinzugefügt werden.

Jedes Modell wird über eine Translation und Rotation im Simulationsraum verankert, wobei das Referenzsystem des überliegenden Modells genutzt wird. Außerdem können im Modell Einstellungen für dessen Physiksimulation vorgenommen werden.

Ein Modell enhält meißt mindestens ein Link-Element, welches zur Darstellung von dessen Geometrie verwendet wird. Mehrere Link-Elemente können dabei mit der Welt oder anderen Link-Elementen über Joint-Elemente verbunden werden. Diese Joint-Elemente können jedoch nicht von außerhalb gesteuert werden, was dieses Dateiformat ungeeignet für Roboterdefinitionen macht.

Lichter besitzen einen Lichttyp, welcher die Ausbreitung des Lichtes im Raum bestimmt. Die erste Art ist direktionales Licht, welches parallel zur gewünschten Achse auftrifft, welches vor allem zur grundlegenden Raumausleuchtung genutzt werden kann. Außerdem existieren noch Punktlichtquellen, welche von einer Position im Raum ausgehen und Spots, welche außerdem noch nur einen gewissen Winkel abdecken.

Die Actor-Komponente wird für animierte Modelle in der Simulation eingesetzt. Sie besteht aus einem Namen für das Modell, einer Skin, welche das Aussehen des Modells definiert und mehreren Animationen. Diese können durch in einem Skript definierte Trajectories ausgeführt

werden, was eine einfache Simulation eines Menschen erlaubt. Eine solche Befehlsfolge kann jedoch nicht von außerhalb der Simulation zur Laufzeit angepasst werden, was weitere Entwicklungsarbeit erforderlich macht.

3.2.3 Robotersimulation

Für die Robotersimulation wird ein Modell des Roboters benötigt, in welchem dieser für die Simulationsumgebung beschrieben wird. Gazebo und ROS nutzen hierfür .urdf-Dateien[28], welche auf XML basieren. In diesen werden die einzelnen Glieder des Roboterarms 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 des Roboters 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. Hierbei werden die Modelle für die Physiksimulation und das Aussehen des Roboters unterschieden.

Gelenke werden separat von den Gliedern definiert und verbinden jeweils zwei Glieder miteinander. Durch das Aneinanderreihen von mehreren Gliedern und Gelenken ist die Beschreibung eines beliebigen Roboteraufbaus möglich. Jedes Gelenk besitzt eine Position und Rotation im Raum, um dessen Effekte, welche vom Gelenktyp abhängen, berechnen zu können. Aspekte wie Reibung und Dämpfung können auch hier für die Nutzung in der Physiksimulation beschrieben werden. Folgende Typen von Gelenken können in urdf-Dateien genutzt werden:

freie Gelenke ermöglichen vollständige Bewegung in allen 6 Freiheitsgraden (Rotation und Translation). Sie stellen den normalen Zustand der Glieder zueinander dar.

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

feste Gelenke sperren alle 6 Freiheitsgrade und werden häufig zur Fixierung 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 kontinuierliche 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. Sie werden zur Umsetzung von Linearaktuatore in der Simulation verwendet.

3.2.4 Menschensimulation

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

Dies setzt jedoch voraus, dass der gesamte Bewegungsablauf zum Simulationsstart bekannt ist. Der Grund dafür ist auf die Definition der Pfade, welche die Bewegung auslösen, zurück zu führen. Diese können nur als dem Actor untergeordnetes Element in der .sdf-Datei definiert werden, was Veränderungen zur Laufzeit ausschließt. Durch diesen Umstand ist der somit realisierbare Simulationsumfang nicht ausreichend, um die gewünschten Szenarien abzubilden.

Um diese Einschränkung 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. Dafür soll ein ROS Action-Server verwendet werden, welcher die Befehle entgegennimmt, unter konstantem Feedback ausführt und nach Erfolgreicher Ausführung den Empfang des nächsten Befehlts zulässt.

Ein solches System soll 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[17] ist das meist genutzte ROS2 Paket für Bewegungsplanung von Robotern. Deshalb existiert viel Dokumentation für die zahlreichen Komponenten, was die Entwicklung erleichtert und zahlreiche direkte Integrationen mit anderen ROS-Paketen, wodurch diese einfacher zusammen genutzt werden können. Diese Eigenschaften machen das Paket als Roboterumgebung für dieses Projekt extrem attraktiv.

MoveIt besteht aus meheren Komponmenten, welche in ihrer Gesamtheit den Bereich der Bewegungsplanung abdecken. Der Nutzer kann mit MoveIt auf verschiedenen Wegen Steuerbefehle für den Roboter absenden, welche durch die Software umgesetzt werden.

Die einfachste Art der Inbetriebnahme ist über das mitgelieferte RViz-Plugin und die demo-Launch-Files, welche durch einen mitgelieferten Setupassistenten für den Roboter generiert

werden. Durch die Ausführung dieser Demo startet RViz, eine Test- und Visualisierungsumgebung für ROS. Darin können Roboterbewegungen unter Zuhilfenahme von Markierungen in RViz geplant und ausgeführt werden.

Da sich eine solche Bewegungsplanung nur beschränkt zur Automatisierung durch Software eignet, müssen die der Demo zugrundeliegenden Schnittstellen genutzt werden. Für die Sprache Python existierte für die Vorgängerversion MoveIt noch das moveit_commander Paket, welches den Zugriff auf MoveIt in Pyhon erlaubt, welches aber für MoveIt2 noch nicht portiert wurde. [18]

Die direkte Nutzung der C++-API ist aktuell die einzige offizielle Möglichkeit, mit MoveIt2 direkt zu interagieren. Dabei kann sowohl die Planung und Ausführung von Bewegungen ausgelöst werden, aber auch Exklusionszonen eingerichtet werden. Außerdem können Objekte virtuell mit dem Roboter verbunden werden, wodurch sich diese in RViz mit dem Roboter bewegen. 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.

Um die durch den Setupassistenten generierten Informationen an MoveIt zu übergeben, wird intern ein RobotStatePublisher verwendet. Dieser läd alle Daten des Robotermodells und gibt sie an andere Programme weiter, welche diese zur Laufzeit anfordern, unter diesen auch MoveIt selbst.

Durch die vorher erwähne C++-API erhält die MoveGroup die Informationen über die gewünschte Bewegung. Dabei können auch bestimmte Einschränkungen des Arbeitsraums, spezielle Trajektorien, oder Limitierungen der Gelenke in der Planung berücksichtigt werden.

Diese Daten können durch eine OccupancyMap ergänzt werden, welche die Bereiche beschreibt, die sich um den Roboter befinden. Eine solche Erweiterung erlaubt die automatische 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 die `ros_control` Controller des Roboters weitergegeben.

Diese Abstaktion erlaubt die Nutzung von sowohl simulierten, aber auch echten Robotern. Hiefür kann einfach ein anderer `ros_control` Controller geladen werden, welcher entweder die simulierte oder echte Hardware ansteuert. 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 im Anhang visualisiert.

3.4 Programmiersprache

Als Programmiersprache kommen in ROS standartmäßig Python[29] und C++[25] zum Einsatz. Diese beiden Sprachen sind in der Softwareentwicklung beliebt, unterscheiden sich jedoch stark in Funktionsumfang und Entwicklungsprozess.

Python ist eine interpretierte Skriptsprache, welche zu den hohen Programmiersprachen zählt. Sie wird in ROS zum Beispiel in .launch.py-Dateien eingesetzt, welche den Start von Diensten in der Umgebung verwalten. Die Sprache kann aber auch für die Programmierung von Nodes innerhalb des ROS-Systems verwendet werden.

C++ hingegen ist eine kompillierte, statisch typisierte, maschinennahe Programmiersprache. In ROS wird C++ für Code verwendet, welcher entweder häufig oder in zeitkritischen Szenarien ausgeführt wird. Aus diesem Grund wird C++ in Nodes verwendet, welche schnell auf große Datenmengen reagieren müssen.

Die Nutzung eines Kompilierers beschleunigt C++ deutlich im Vergleich zu Python, ist jedoch weniger geeignet für häufige Modifikation. Dies ist vor allem in häufig geänderten Programmen ein Nachteil, welche dann wieder kompilliert werden müssten. Aus diesem Grund wird Python vor allem in .launch.py-Dateien verwendet, welche die Interaktion der anderen Programme in der Umgebung verwalten.

Um die gewünschten Funktionen für die Simulation umsetzen zu können, ist die Programmierung in C++ nötig. Da zum Beispiel Gazebo-Plugins auf C++ als Programmiersprache ausgelegt sind, was die Nutzung anderer Sprachen stark erschwert. Ein Grund dafür ist die hohe Geschwindigkeit, welche bei einer hohen Anzahl an Simulationsschritten pro Sekunde benötigt wird. Außerdem kann MoveIt2 zur aktuellen Zeit nur mit C++ direkt gesteuert werden.

Die Verwendung von C++ für die zu entwickelnden Nodes erscheint deshalb aus oben genannten Gründen naheliegend. In den Launch-Skripten wird jedoch Python verwendet werden, da hier die Vorteile einer Skriptsprache überwiegen.

3.5 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 und dem geplanten Konzept integrierbar.

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. Darunter fällt die Modifizierung von Parametern der Aktionen, aber auch der vollständige Abbruch einer Aktion durch äußere Einflüsse.

Das .xml-Format der Behavior Trees ermöglicht einen einfachen Austausch des Verhaltens, ohne die unterliegende Programme 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.

Das Blackboard ist eine Schicht, welche den Datenfluss zwischen den Nodes erlaubt. In diesem System kann unter Verwendung einer Zeichenkette als Identifikator ein Wert in Form einer Referenz hinterlegt werden. Sogenannte Ports erlauben Nodes, Daten aus dem Blackboard zu lesen und auf dieses zu schreiben.

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.

Ports können verwendet werden, um Nodes generischer zu gestalten. Durch veränderbare Parameter im später erstellten Tree können Nodes ohne Programmänderung verändert werden. Falls die Nodes mit Bedacht erstellt wurden, kann so auf viele spezialisierte Nodes verzichtet werden, da deren Funktionen aus einfacheren Nodes mit variablen Parametern abgebildet werden kann. Diese in den Ports übertragenen Daten können sowohl aus einem String ausgelesen werden, aber auch aus dem sogenannten Blackboard entnommen werden.

Um die Übersetzung aus einem String zu ermöglichen, muss eine Funktion implementiert werden, welche diesen String in den gewünschten Zieltyp übersetzt. Viele einfache Datentypen, wie Ganzzahlen und Gleitkommazahlen, werden von BehaviorTree.Cpp bereits durch mitgelieferte Funktionen unterstützt.

Das Blackboard ist ein System, welches die Nutzung von Variablen als Parameter für Ports 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 auch ein Eingriff durch äußere Einflüsse 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.

3.5.1 Asynchrone Nodes

Da nicht jeder Prozess sofort vollständig durchgeführt werden kann, muss die Möglichkeit geschaffen werden, lang anhaltende Prozesse abzubilden. Dies geschieht in `BehaviorTree.CPP` durch asynchrone Nodes.

Eine asynchrone Node besitzt neben den Zuständen SUCCESS und FAILURE einer normalen Node auch noch die beiden neuen Zustände RUNNING und IDLE. Außerdem werden mehrere Funktionen definiert, welche den Lebenszyklus der Node darstellen.

Wird eine Node durch den Aufruf der `onStart`-Funktion gestartet, geht diese in einen der Zustände RUNNING, SUCCESS oder FAILURE über.

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 in der neuen `onRunning`-Funktion abgefragt.

Der IDLE-Zustand ist ein besonderer Zustand, welcher nur durch eine vollständige Ausführung erreichbar ist. Er wird von der Node angenommen, nachdem deren Ausführung durch SUCCESS oder FAILURE beendet wurde.

Im Falle eines Abbruchs, welcher durch andere Nodes im Baum ausgelöst werden könnte, muss die Ausführung der Node vorzeitig beendet werden. Dies geschieht mit der neuen `onHalted`-Funktion, welche die Ausführung der Node abbrechen kann.

3.5.2 Dateiformat

Das in BehaviorTree.Cpp verwendete Dateiformat, um Behavior Trees zu erstellen, basiert auf XML. Jedes Dokument beginnt dabei mit einem Root-Element, welches alle BehaviorTrees und eine Referenz auf die ID des Hauptbaumes enthält. Diese wird benötigt, da auch Unterbäume im selben Dokument deklariert und genuzt werden können, diese aber sonst nicht vom Hauptbaum unterscheidbar sind.

Jeder Baum beginnt mit einem BehaviorTree-Element, welches als Attribut die ID des Baumes besitzen muss. Als untergeornete Elemente des Baumes werden die Nodes entsprechend der gewünschten Baumstruktur angeordnet.

Zur besseren Visualisierung der XML-Struktur wurde hier die bereits aus dem Konzept bekannte Baumstruktur, hier noch einmal in Abbildung 3.1 zu sehen, umgewandelt. Die resultierende Datei ist in Abbildung 3.2 abgebildet. Dabei ist zu beachten, dass die Root-Node nicht in der Struktur existiert, da diese nur den Eintrittspunkt in die Struktur darstellt. Außerdem können selbst definierte Nodes sowohl direkt mit ihrem Namen, aber auch über den Namen Action mit ihrem Namen als ID-Parameter, referenziert werden.

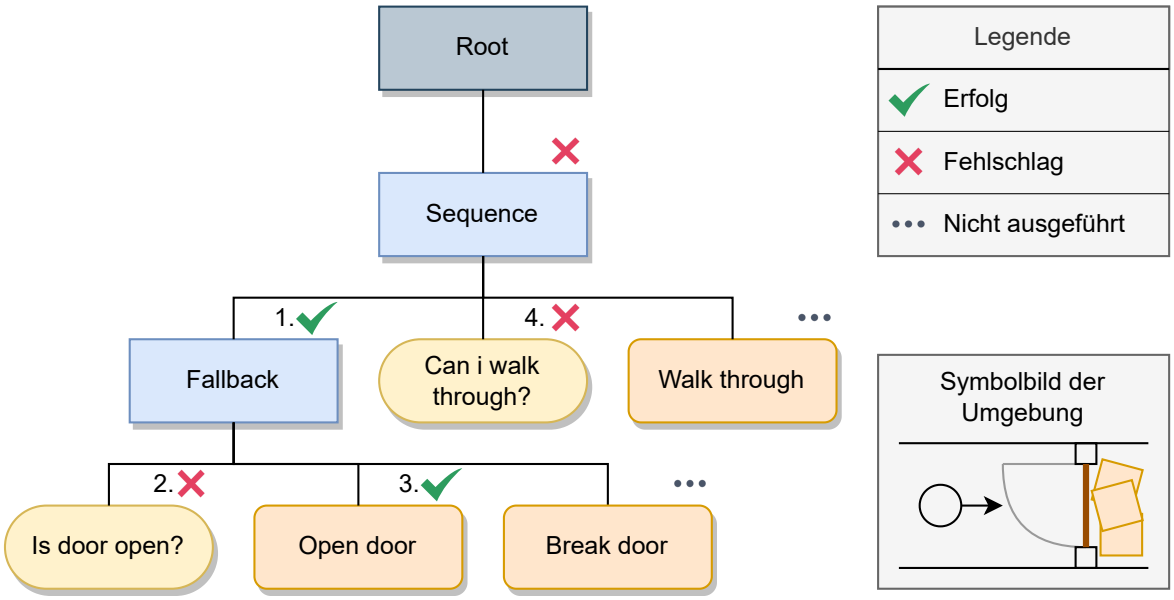

Abbildung 3.1: Beispiel eines BehaviorTrees

```xml
<?xml version="1.0"?>
<root main_tree_to_execute="demoTree">
    <BehaviorTree ID="actorTree">
        <Sequence>
            <Fallback>
                <Action ID="IsDoorOpen"/>
                <Action ID="OpenDoor"/>
                <Action ID="BreakDoor"/>
            </Fallback>
            <CanWalkThrough/>
            <WalkThrough/>
        </Sequence>
    </BehaviorTree>
</root>
```

Abbildung 3.2: Beispiel eines BehaviorTrees als .xml

3.6 Docker-Compose als Virtualisierungsumgebung

Docker ist eine Virtualisierungsumgebung für Anwendungen, welche die komplette Umgebung für deren Ausführung bereitstellt. Dadurch wird die Inbetriebnahme von Anwendungen, welche spezielle Umgebungen für ihre Ausführung benötigen auf beliebigen Systemen 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 in einem Systemverzeichnis des Hostsystems, oder ein “bind mount” 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

Bei der Umsetzung des geplanten Systemaufbaus kam es zu mehreren Problemen, welche den geplanten Systemaufbau, sowie dessen Komplexität, negativ beeinflussen.

Die Kommunikation zwischen dem Actormodell und dem Behavior Tree musste in mehrere Komponenten aufgeteilt werden, um Konflikte innerhalb der Simulationssoftware zu vermeiden.

Zudem ist die Bewegungsplanung mit MoveIt2 deutlich komplexer als vorerst angenommen. Diese Komplexität entsteht aus der Interaktion mehrerer Komponenten, welche das Gesamtsystem zur Ausführung benötigt. Alle Einzelsysteme mussten hierfür konfiguriert werden, um die Kommunikation dieser zu ermöglichen.

Anhand der hier genannten Änderungen wurde die Übersicht des Systems angepasst, welche in Abbildung 4.1 zu sehen ist.

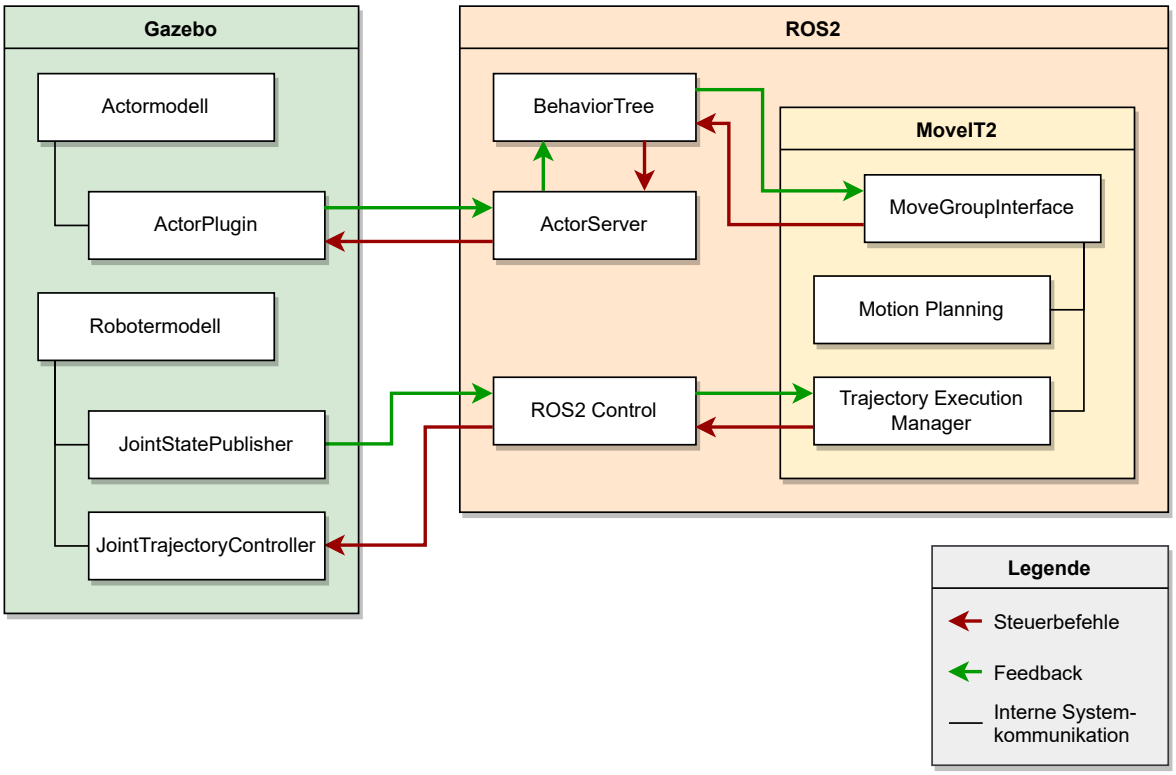

Abbildung 4.1: Visualisierung des überarbeiteten Konzepts

4.1 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 Modifikationen 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.

Zum Starten des Containers muss der Befehl `./start.sh` im Verzeichnis der Containerinstallation ausgeführt werden, welcher die obengenannten Schritte ausführt und den Container startet. Eine Verbindung zum Container kann aufgebaut werden, nachdem die Meldung `ros_1 | Ready to connect.` in der Konsole erscheint.

Dafür muss ein SSH-Client eingesetzt werden, welcher eine Verbindung zu der lokalen Netzadresse des Systems mit dem Benutzer `ros` aufbauen kann. Der Port des SSH-Servers wird dabei durch die Deklaration im docker-compose.yaml an das Hostsystem durchgereicht. Hierbei ist zu beachten, dass der SSH-Server im Container auf Port 2222 ausgeführt wird, was bei der Verbindung beachtet werden muss.

Nach der Verbindung wird automatisch die ROS2-Umgebung eingerichtet, welche sofort nach Verbindungsaufbau genutzt werden kann. Um die erstellten Pakete zu kompillieren, kann das Skript `build.sh` im `workspace`-Verzeichnis verwendet werden.

```
#!/bin/bash
pushd "$(dirname "$0")" || exit
colcon build --event-handlers console_cohesion+ --cmake-args
↪  -DCMAKE_EXPORT_COMPILE_COMMANDS=ON -G Ninja
popd || exit
```

Dieses Skript nutzt `colcon`, um alle Pakete in `/workspace`-Verzeichnis zu erstellen. Dabei wird auch eine `compile_commands.json`-Datei im build-Unterordner erstellt, welche von Entwicklungsumgebungen zur Syntaxvervollständigung genutzt werden. Um eine Nutzung in allen Umgebungen zu erlauben, wurde diese in das Hauptverzeichnis gelinkt, da einige Umgebung nur dort nach dieser Datei suchen.

Da der Kompilliervorgang parallel abläuft, erscheinen Informationen zu allen Paketen gleichzeitig, was das Finden eines Fehlers erschwert. Um trotzdem alle wichtigen Informationen zu erhalten, kommt der Event-Handler `console_cohesion+` zum Einsatz, welcher die Ausgaben neu formatiert. Durch diesen werden die Ausgaben gruppiert, und erst nach einen vollständigen Kompilliervorgang eines Pakets nacheinander ausgegeben. Dies ermöglicht es, aufgetretene Fehler einfacher auf ein bestimmtes Paket zurückführen zu können, ohne das gesamte Log zu durchsuchen. Hierbei ist es hilfreich, dass die verbleibenden Kompilliervorgänge abgebrochen werden, falls der Kompilliervorgang eines Pakets fehlschlägt. Dadurch befindet sich der Fehlers immer im letzten Paket der Ausgabe, da alle anderen Prozesse abgebrochen und nicht ausgegeben werden.

4.2 Entwicklungsumgebung

Ein einfacher Texteditor ist für das Schreiben von ROS-Packages ausreichend und bietet bei der Arbeit mit Containern sogar einen großen Vorteil. Das Editieren von Dateien ist mit einem Texteditor auch von außerhalb des Containers möglich. Jedoch besitzt ein Texteditor nur wenige Funktionen einer vollständigen Entwicklungsumgebung, welche den Prozess der Softwareentwicklung beschleunigen. Um diese Funktionen bieten zu können, analysieren Entwicklungsumgebungen den geschriebenen Code. Dies geschieht meißt auf eine von zwei unterschiedlichen Weisen.

Die Entwicklungsumgebung kann eine interne Repräsentation des geschriebenen Codes generieren. Diese Repräsentation kann nun genutzt werden, um die implementierten Funktionen der Entwicklungsumgebung bereitzustellen. Um dies zu erreichen, muss für jede unterstützte Sprache Code geschrieben werden, welcher in die Entwicklungsumgebung integriert werden muss.

Als Alternative existiert das Language Server Protocol, kurz LSP, welches eine Schnittstelle in Form eines JSON-RPC Protokolls zur Verfügung stellt, um Infomationen über den Code an die Entwicklungsumgebung zu übergeben. Dies erlaubt einer Entwicklungsumgebung, nur noch den benötigten Server der Sprache zu starten, um Informationen über den Code in einem standartisierten Protokoll zu erhalten.

Der große Vorteil des LSP ist, dass eine Entwicklungsumgebung alle Funktionen der Programmiersprache vollständig unterstützt, solange das Protokoll vollständig implementiert wurde und ein entsprechender Server für die Sprache existiert.

Jedoch können bestimmte Funktionen, welche beim Design des LSP nicht bedacht wurden, einfacher in interner Implementation umgesetzt werden. Deswegen haben Entwicklungsumgebungen mit interner Coderepräsentation häufig mehr Funktionen, spezialisieren sich jedoch auf bestimmte Sprachen.

In diesem Projekt wurden mehrere Entwicklungsumgebungen mit den jeweils unterschiedlichen Verfahren verwendet. Um diese mit dem Docker-Container verwenden zu können, müssen diese Umgebungen die Entwicklung auf entfernten Systemen unterstützten, da der Container vom ausführenden System getrennt ist.

Um dies zu ermöglichen, wird ein Teil der Entwicklungsumgebung im Container ausgeführt, um mit dem inneren Kontext der Maschine arbeiten zu können. Dazu wird beim Start einer Verbindung zu einem entfernten System dieser Server auf das Zielsystem übertragen und gestartet.

Die anfängliche Entwicklung wurde mit PyCharm und CLion durchgeführt, da diese durch ihre interne Codeanalyse die ROS-Umgebung ohne Konfiguration nutzen konnten. Jedoch sind diese Umgebungen sehr ressourcenintensiv, was die gleichzeitige Ausführung erheblich verlangsamt.

Aus diesem Grund wurde später eine Entwicklungsumgebung mit LSP-Unterstützung, in diesem Fall Lapce, verwendet. Dazu wurden dem Container die benötigten LSP-Server hinzugefügt und konfiguriert, um diese mit Lapce nutzen zu können. Unter Verwendung dieser neuen Server kann Lapce nun Codevervollständigung und Codeanalyse wie PyCharm und CLion durchführen. Dabei wird auch der Ressourcenverbrauch gesenkt, da nur ein einziger Editor benötigt wird.

4.3 Verwendete Datentypen

In diesem Projekt werden viele unterschiedliche Datentypen verwendet. Diese Datentypen sind größtenteils aus der Programmiersprache C++ bekannt, jedoch werden auch weitere Typen aus eigenem Code oder eingebundenen Bibliotheken verwendet. Um die Verständlichkeit der

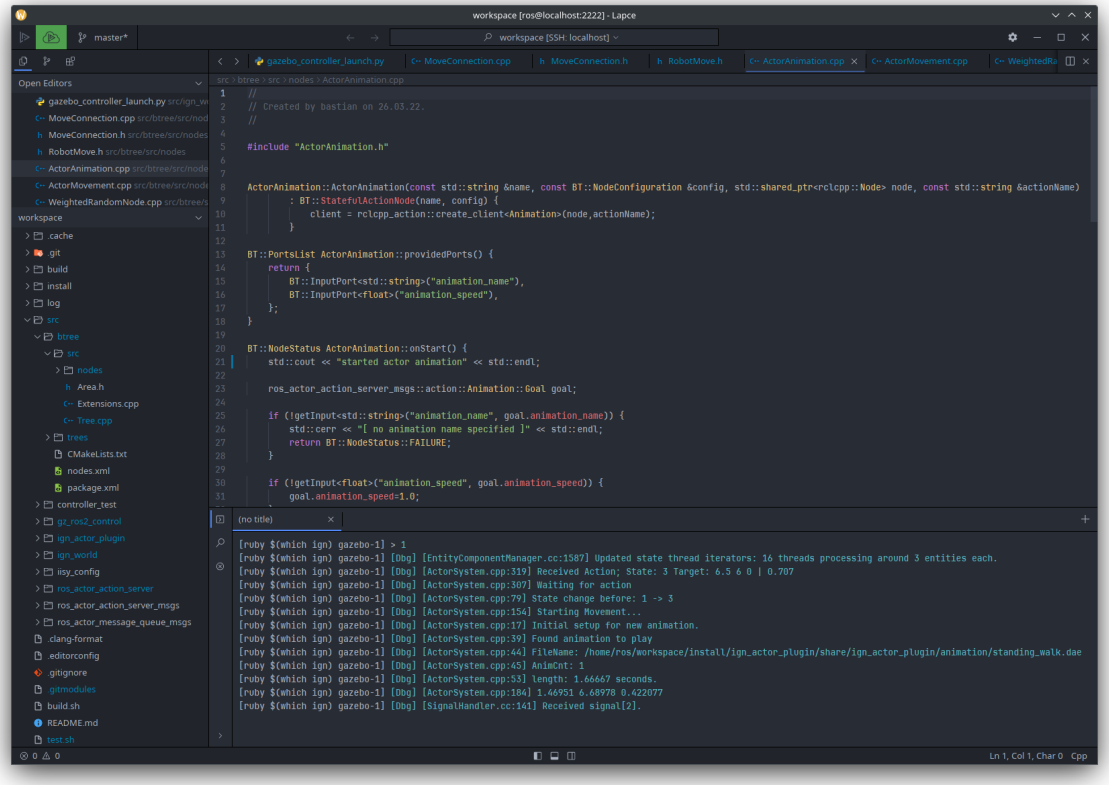

Abbildung 4.2: Entwicklungsumgebung Lapce

Dokumentation zu erleichtern, sind die in der Implementation verwendeten Datentypen hier zusammengefasst und beschrieben.

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 Zweidimensionalen 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.

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.

4.4 Simulationswelt

Die Welt der Simulation wird im Paket `ign_world` zur Verfügung gestellt. In diesem Paket sind sowohl die Geometrien der Welt, aber auch die benötigten Dateien zum Starten der Simulation enthalten.

In diesem Fall handelt es sich um den Raum, in welchem die Interaktion zwischen Mensch und Roboter stattfinden soll. Für diesen Raum wurde zuerst ein Raumplan erstellt, welcher alle benötigten Bereiche für die Szenarien besitzt (Abbildung 4.3).

Zuerst wird ein Stellplatz für den Roboter benötigt, welcher an einer Ecke des Raumes positioniert werden sollte. Diese Position wurde gewählt, um die Verfahrgeschwindigkeit des Roboters höher zu halten, welche je nach Distanz zum Menschen angepasst werden soll.

Des weiteren werden eine Arbeits- und Lagersätte für den Menschen benötigt, welche in den Szenarien genutzt werden sollen. Im Koexistenzszenario soll der Mensch nur an diesen Stellen seine Arbeit verrichten, jedoch sich selten dem Roboter nähern, um dessen Fortschritt zu begutachten. Die Lagerstätte soll im Kooperationsszenario neben der Arbeit des Menschen auch für die fehlerhaften Teile verwendet werden, welche durch den Roboter aussortiert und durch den Menschen dorthin verbracht werden sollen. Eine Nutzung im Kollaborationsszenario ist nicht nicht geplant, da der Mensch den Roboter überwachen und dessen Fehler korrigieren soll.

Der so geplante Raum wurde in Blender modelliert und als .stl-Datei exportiert, um sie in die Welt einbinden zu können. Für das Kooperationsszenario wurde ein Förderband moddeliert, welches in diesem Szenario dem Raum hinzugefügt wird.

Der so erstellte Raum wird in einer .sdf-Datei als Modell referenziert, um diesen später in die Simulation einbinden zu können. Das Förderband erhält ein eigenes Modell in einer weiteren Datei, welche zur Laufzeit in die Simulation geladen wird. Für beide wird, wie später auch für den Roboter selbst, das Paket `ros_gz_sim` verwendet. Dieses veranlasst mit dem `create`-Programm das Erstellen der übergebenen Datenstruktur in Gazebo.

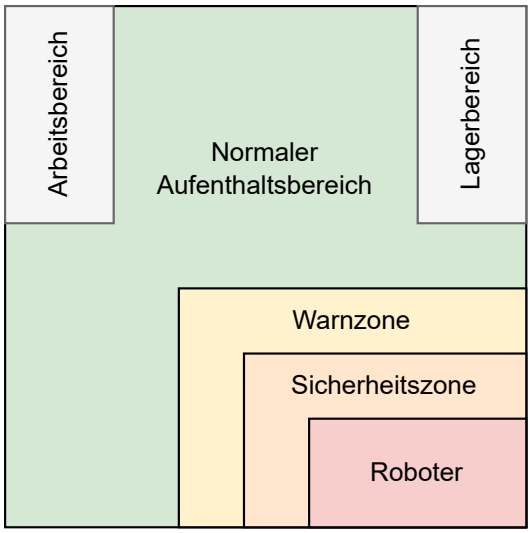

Abbildung 4.3: Geplanter Raum

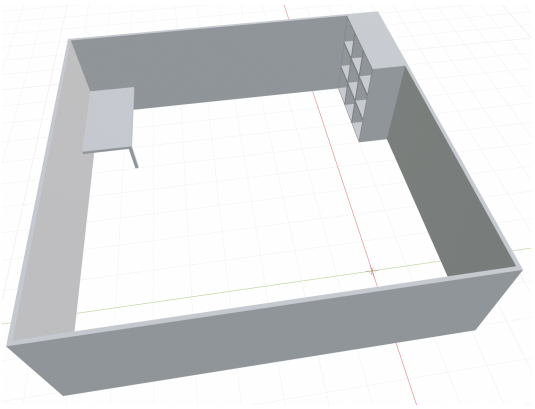

Abbildung 4.4: Umsetzung in Blender

4.5 Mensch

4.5.1 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.

4.5.2 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.

Normalerweise würde an diesem Punkt das Skelett verbunden und angepasst werden, jedoch fehlen dem importierten Skelett viele Knochen, wie zum Beispiel für Finger und Zehen, aber auch Rotationsknochen für Arme und Beine. Durch diese fehlenden Knochen ergeben sich im späteren Schritt der Animation visuelle Fehler.

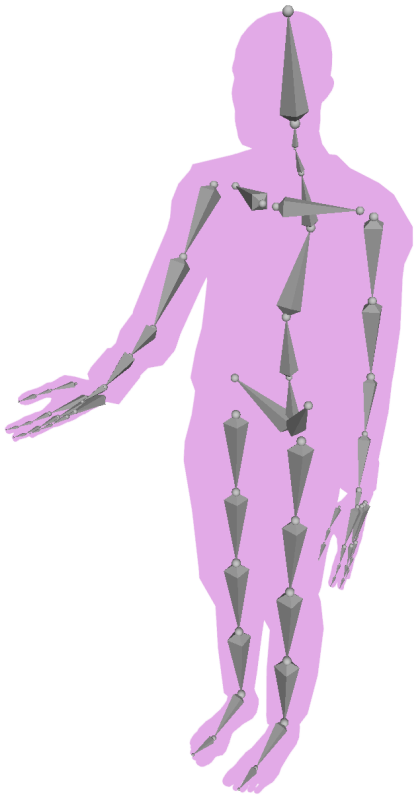

Abbildung 4.5: Knochen des Modells

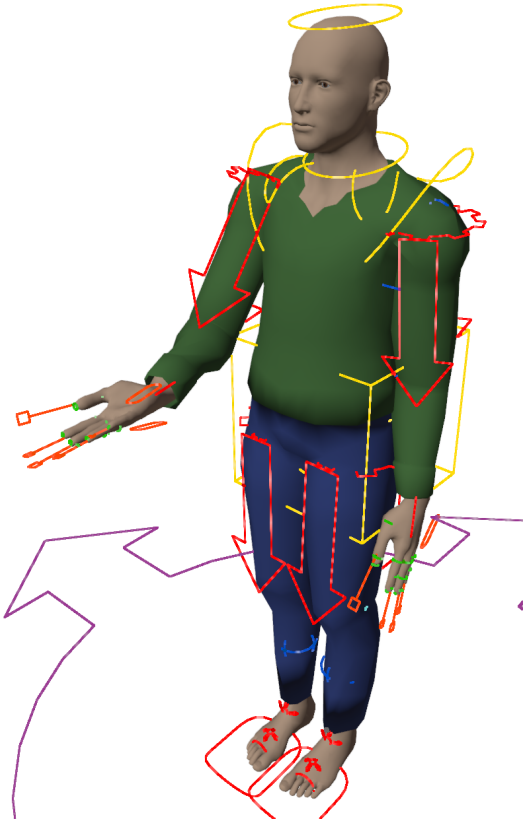

Abbildung 4.6: Armaturen des Modells

Um dieses Problem zu beheben, wurde mit dem “Rigify”[22]-Plugin ein standartisiertes Menschenskelett generiert. 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. Nun müssen die Knochen durch Verschiebung und Skalierung an die richtigen Positionen im Modell gebracht werden.

Dabei muss auf die Ausrichtung der Knochen zueinander geachtet werden. Das Kreuzprodukt der Vektoren beider Knochensegmente bestimmt 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. Dies liegt am Kreuzprodukt, welches im oben genannten Fall ein Nullvektor ist, wodurch keine Beugeachse bestimmt werden kann.

Deswegen muss bei der Platzierung darauf geachtet werden, dass der Startpunkt A des ersten und der Endpunkt C des zweiten Knochens auf einer Gerade liegen. Der Verbindungspunkt B der beiden Knochen wird nun vorerst auf dieser Gerade platziert. Dieser muss nun senkrecht zu dieser Gerade und der gewünschten Biegeachse verschoben werden, wie in Abbildung 4.7 gezeigt.

Das so erstellte Skelett ist in Abbildung 4.5 visualisiert.

Um eine bessere Verformung bei der Bewegung von Knochen zu erreichen, wird das so genannte “weight painting” eingesetzt. Hierfür werden für jeden Knochen entweder automatisch oder manuell Teile des Meshes mit Gewichten versehen. Je höher die Wichtung, desto stärker ist die Verformung an dieser Stelle, wenn der Knochen bewegt oder skaliert wird.

Da das Animieren aller Knochen einzeln sehr zeitaufwändig ist, werden diese in Gruppen zusammengefasst. Hierfür werden in Blender sogenannte Constraints eingesetzt, welche Knochen automatisch durch eingestellte Bedingungen positionieren können.

Durch das Verwenden von Rigify und einem standartisierten Skelett ist die Generierung der Constreaints einfach. Hierfür können die in dem Skelett enthaltenen Informationen vom Plugin genutzt werden, um alle häufig genutzten Constraints automatisch zu generieren. In diesem Schritt werden auch neue Knochen eingefügt, welche das Skelett durch Constraints beeinflussen. Das neue Animationsmodell, visualisiert in Abbildung 4.6, kann nun für Animationen genutzt werden.

Hierfür sehen mehrere Knochengruppen zur Verfügung, welche typische Animationsmethoden abdecken.

In Rot sind hier Knochen markiert, welche für inverse Kinematik genutzt werden, in diesem Fall Arme und Beine. Diese Knochen geben gewünschte Ausrichtung und die Zielposition des untersten Knochens vor. Aus diesen Angaben wird die wirkliche Position der Knochen berechnet, welche durch die Constraints automatisch auf diese übertragen wird.

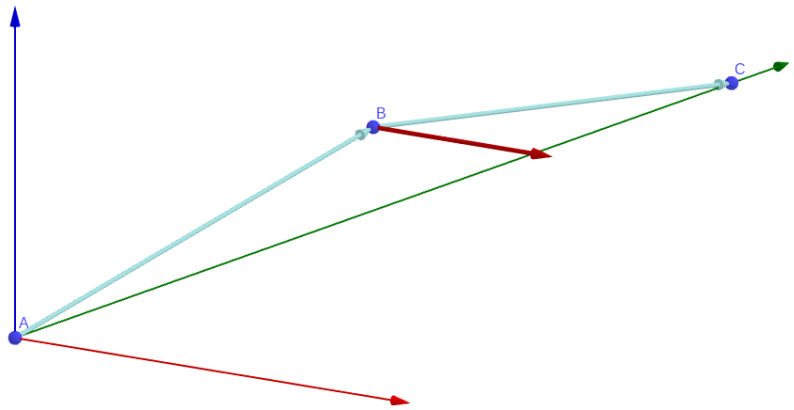

Abbildung 4.7: Visualisierung der generierten Beugeachse

Orange gefärbte Knochen werden für generelle Einstellungen von Fingern, Handfläche und Zehen genutzt. Die Handfläche kann so gekrümmt werden, wodurch sich alle Finger mit der Krümmung mitbewegen. Ein Abknicken aller Zehen kann durch die Rotation des Hilfsknochens an den Zehen erreicht werden. Die Finger können auch einzeln rotiert und eingeknickt werden. Hierbei wird der Grad des Einknickens nicht durch eine Rotation des Knochens ausgedrückt, da diese bereits durch die Rotation des Fingers selbst benutzt wird. Anstatt der Rotation wird die Skalierung des Knochens eingesetzt, wobei ein kleinerer Knochen eine stärkere Knickung aller Fingerglieder bewirkt.

Die gelben Knochen beeinflussen die generelle Pose des Modells. Der Quader in der Hüfte gibt die gewünschte Höhe des Modells vor, welche auch für die inverse Kinematik benutzt wird. Die anderen Knochen beeinflussen die Rotation des Beckens, der Wirbelsäule, der Schultern und des Kopfes.

Das hier erstellte, verbesserte Rigify-Skelett kann nun durch den Einsatz der neuen Constraints einfacher animiert werden. Jedoch ist das Exportieren eines solchen Rigs ist schwierig, da viele Grafikengines verschachtelten Skelette und Constraints nicht verwenden können.

4.5.3 Export der Modellanimationen

Um aus einem existierenden, vollständig verbundenen Skelett einzelne Knochen zu extrahieren, exisitiert ein weiteres Plugin mit dem Namen “GameRig”[1]. Dieses separiert die neuen Steuerknochen wieder vom ursprünglichen Modell, wodurch in diesem nur noch die deformierenden Knochen enthalten sind.

Alle erstellten Animationen der Steuerknochen müssen nun in direkte Bewegungen der deformierenden Knochen des Modells umgewandelt werden. Um dies zu erreichen wird der in Abbildung 4.8 visualisierte Arbeitsablauf verwendet.

Im ersten Schritt wird das zu exportierende Skelett ausgewählt, um diesem später die neue Animation zuweisen zu können. Dann muss im zweiten Schritt die gewünschte Animation der Liste der bekannten Animationen hinzugefügt werden.

Danach muss die durch die Constraints abgebildete Animation auf das ausgewählte Skelett übertragen werden, was mit dem “Bake Action Bakery”-Knopf ausgelöst wird. Dabei wird die Position aller deformierenden Knochen zu jedem Zeitpunkt der Animation bestimmt, und in einer neuen Animation mit modifiziertem Namen abgespeichert.

Diese neu erstellte Animation kann nun im vierten Schritt dem ausgewählten Skelett zugewiesen werden. Nach dieser Veränderung kann die diese Animation als Collada-Animation exportiert werden. Dazu muss noch das visuelle Modell der Selektion hinzugefügt werden, da Gazebo dieses für jede Animation benötigt.

Nun kann der Export über die in Blender integrierte Exportoption ausgelöst werden. Hierfür müssen die in Abbildung 4.9 verwendeten Exporteinstellungen verwendet werden, damit Gazebo die exportierte Animation nutzen kann. Alle in der Blender-Version 3.5 nicht standartmäßigen Einstellungen wurden dabei durch Punkte hervorgehoben.

Zuerst müssen im Reiter “Main” die globalen Exporteinstellungen angepasst werden. Der markierte Punkt 2. bewirkt, dass nur die vorher ausgewählten Modellteile exportiert werden. Dies verkleinert das Modell, da alle Steuerknochen von Gazebo nicht verwendet werden können, was die Ladezeit der Simulation verbessert.

Punkt 3. und 4. bewirken, dass das exportierte Modell in Gazebo verwendete Vorwärtsachse erhält. In Blender ist Y als Vorwärtsachse üblich, jedoch verwendet Gazebo die X-Achse für diesen Zweck. Wird diese Modifikation nicht vorgenommen, sind die Modelle um 90 Grad verdreht.

Nun muss im mit 5 markierten Animations-Reiter noch eine letzte Einstellung vorgenommen werden. Die mit 6 markierte Einstellung bewirkt, dass alle Bewegungen der exportierten Knochen mit gespeichert werden, auch wenn diese sich nicht bewegen. Da sich einige Knochen nur am Anfang der Animation in einer bestimmten Pose befinden, exportiert Blender diese nicht, was in Gazebo zu verdrehten Knochen führt.

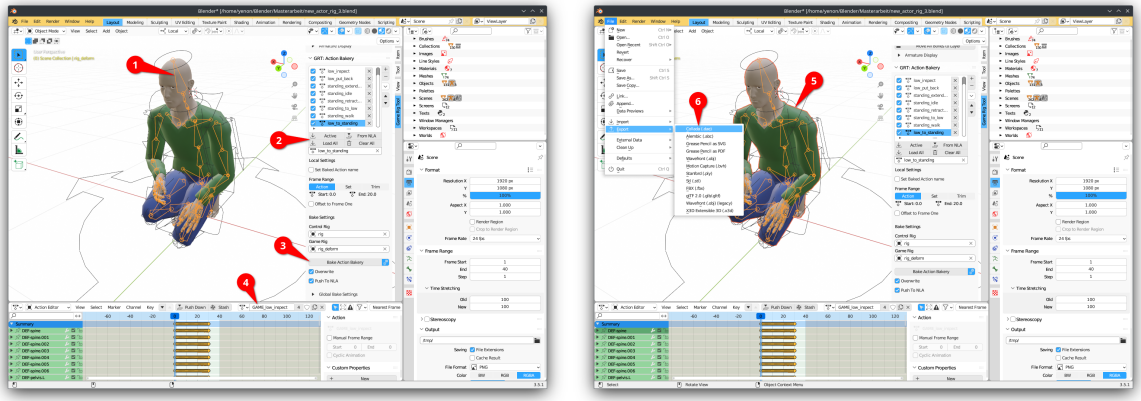

Abbildung 4.8: Vorbereitung zum Export mit GameRig

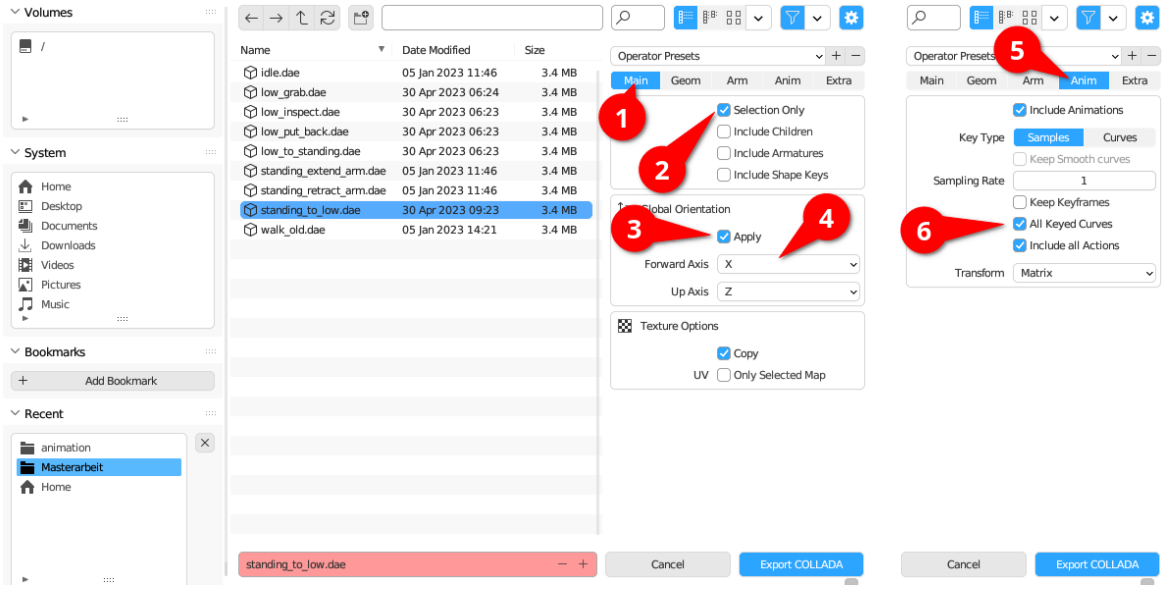

Abbildung 4.9: Benötigte Exporteinstellungen in Blender

4.5.4 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. Sie sind absichtlich nicht in den nutzenden Paketen untergebracht, da sie durch ein externes Programm in den entsprechenden Code umgewandelt werden.

Jede Action definiert 3 Nachrichten, welche zu unterschiedlichen Zeitpunkten in ihrer Ausführung verwendet werden. Die definierten Nachrichten sind eine Startnachricht, eine Feedbacknachricht und eine Endnachricht.

Ein ActionServer definiert 3 Funktionen, welche die Handhabung einer Aktion definieren. Die erste Funktion übergibt den Wert der Startnachricht, welche mit einer Antwort quittiert werden muss. Hierbei sind die Antworten ACCEPT_AND_DEFER, ACCEPT_AND_EXECUTE und REJECT möglich.

ACCEPT_AND_EXECUTE signalisiert die sofortige Ausführung des gewünschten Befehls und ACCEPT_AND_DEFER steht für eine spätere Ausführung der gewünschten Aktion. Die Option REJECT bricht die Ausführung der Aktion vorzeitig ab.

Die zweite Funktion übergibt Abbruchanfragen an den Server, falls die Aktion durch den Client abgebrochen werden soll. Auch diese Anfrage kann entweder mit ACCEPT akzeptiert werden, oder mit REJECT zurückgewiesen werden.

Um Feedback während der Ausführung der Aktion geben zu können, exisitiert die dritte Funktion. Diese erhält als Parameter ein Objekt, mit welchem Feedback- und Endnachrichten an den Client gesendet werden können.

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.

ActorPlugin

Das ActorPlugin nutzt die empfangenen Nachrichten aus der eingehenden Message Queue, um diese in der Simulation umzusetzen. Der Code des Plugins ist dabei im Paket `ign_actor_plugin` organisiert, welches im Gazebo-Modell der Welt referenziert werden muss, um das Plugin zu laden.

Das Plugin erhält durch die empfangenen Nachrichten mehrere Zustände, welche im folgenden erläutert werden:

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 eines ROS-ActionServers und verlässt sich auf den ActorServer, welcher die Feedbacknachrichten in das richtige Format bringt. Feedback wird in den Zuständen Movement und Animation in eine zweiten Message Queue zu jedem Simulationsschritt gesendet. Um Zustandsübergänge erkennen zu können, werden auch diese als Feedback an den ActorServer gesendet.

ActorServer

Der ActorServer ist die Brücke zwischen ROS und dem ActorPlugin und ist im Paket `ros_actor_action_server` enthalten. Dieser weitere Dienst bindet das ActorPlugin an ROS an.

Es werden dafür zwei ROS-ActionServer gestartet, welche jeweils Bewegungen oder Animationen des simulierten Menschen auslösen können. Beide ActionServer prüfen bei dem Empfang eines neuen Ziels zuerst, ob bereits eine andere Aktion ausgeführt wird. Sollte bereits eine andere Aktion ausgeführt werden, wird die Anfrage abgelehnt. Im anderen Fall wird die Aufgabe akzeptiert und in das MessageQueue-Format übersetzt und an das ActorPlugin gesandt.

Um das Starten mehrerer gleichzeitiger Aktionen zu unterbinden, muss der Empfang einer neuen Anfrage bestätigt werden, bevor weitere Befehle über den ROS-ActionServer entgegen genommen werden können. Hierzu wird ein Mutex verwendet, welcher die Auswertung neuer Nachrichten verhindert, so lange der aktuelle Befehl noch nicht durch das Plugin bestätigt wurde. 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.

4.6 Roboter

4.6.1 Übersicht

Der Roboter besteht aus vielen interagierenden Systemen, welche in ihrer Gesamtheit das vollständige Robotermodell in der Simulation verwendbar machen.

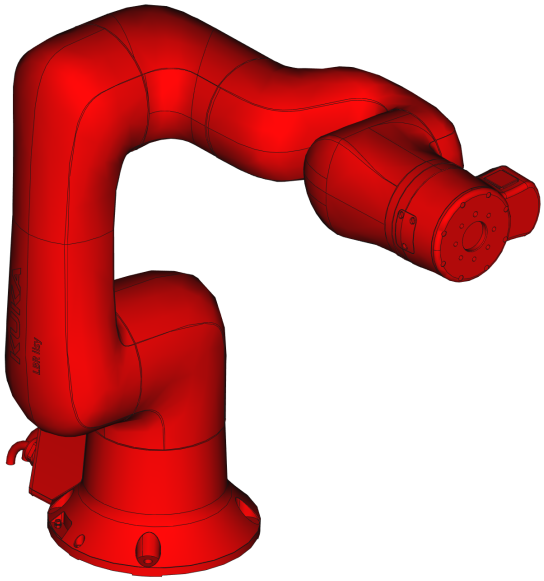

Abbildung 4.10: Rohdaten aus .stl-Datei

Zuerst muss ein Modell des Roboters erstellt werden, welches in Gazebo geladen werden kann. Dieses Modell muss dann für die Bewegungsplanung mit MoveIt erweitert werden. Hierbei werden Controller von ros_control mit dem Modell verbunden, um den aktuellen Zustand der Achsen zu überwachen und diese steuern zu können.

Um diese Vielfalt an Daten standartisiert an andere Software in ROS weitergeben zu können, wird eine MoveGroup in ROS gestartet, welche die Planung von Bewegungen durch andere Programme zulässt.

4.6.2 Modellierung

Für den Kuka LBR iisy existiert kein Simulationsmodell für Gazebo und ROS, weswegen dieses Modell aus Herstellerdaten generiert wurden. Hierzu stand eine .stl-Datei des Herstellers zur Verfügung. Aus dieser Datei wurden mit FreeCAD[9] alle Glieder des Roboters als Colladа-Dateien exportiert. Dabei muss darauf geachtet werden, dass die exportierten Daten eine ausreichende Meshgröße haben, welche vorher in FreeCAD eingestellt werden muss. Dies erlaubt das Erhalten der feinen Strukturen des Arms, auch visualisiert in Abbildung 4.10, welche erst später in Blender reduziert werden sollen.

Diese Dateien können dann in Blender bearbeitet werden, um sie für die Simulation tauglich zu machen. Hierfür 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. Das vollständige visuelle Modell ist in Abbildung 4.11 zu sehen.

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. Das resultierende Modell, welches in Abbildung 4.12 dargestellt wird, wird später zur Kollisionserkennung verwendet.

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.

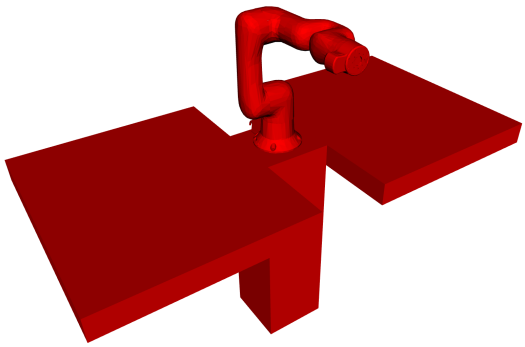

Abbildung 4.11: Visuelles Modell

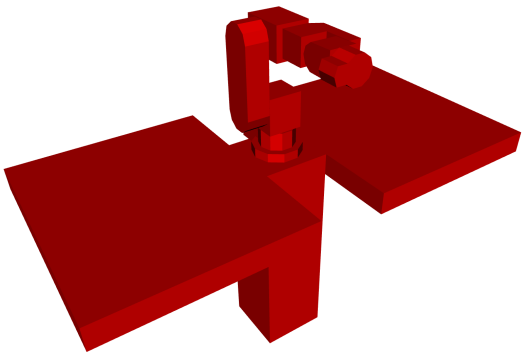

Abbildung 4.12: Kollisionsmodell

4.6.3 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. Diese können standartmäßig nicht in Gazebo, RViz und MoveGroup gleichzeitig geladen werden, da diese Programme unterschiedliche Pfadangaben verlangen, was die Nutzung verhindert.

Eine Möglichkeit zur Lösung des Problems ist die Verwendung von `file://`-URIs, welche von allen dieser Programme gelesen werden können. Jedoch ist hier als Nachteil zu verzeichnen, dass der Moveit Setup Assistent diese URIs nicht lesen kann, was zu Fehlern bei einer Rekonfiguration führen kann.

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.

4.6.4 Integration mit Gazebo

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 Werkzeug 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.

In diesem Fall wird das Modell als String geladen, welcher durch `xacro` erstellt wurde. Dies ist nötig, um Informationen aus anderen Dateien in das generierte XML übernehmen zu können.

Im Modell sind auch die verwendeten Gazebo-Plugins deklariert, welche für die Integration mit `ros_control` verantwortlich sind. Das Gazebo-Plugin läd dabei die verwendeten Controller und versorgt diese mit Informationen über den Roboter.

4.7 Behavior Trees

Alle Behavior Trees wurden im Paket `btree` organisert, welches die Bäume im XML-Format und die Implementation der Nodes und umliegenden Infrastruktur enthält.

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. Hierzu wird der Iterator der unterlegenden Sequenz nur erhöht, wenn die untergeordnete Node SUCCESS zurück gibt. Außerdem wird der Iterator nicht zurückgesetzt, wenn die Ausführung dieser modifizierten Sequenz abgebrochen wird.

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.

4.7.1 Subtrees

Um die Wiederverwendung von bestimmten Verhaltensweisen zu erleichtern, wurden diese in Subtrees ausgelagert. In diesem Fall wurden die Aktionen “Arbeiten an der Werkbank” und “Ablegen eines Objekts im Lagerregal” des Menschen ausgelagert, da diese in mehreren Fällen verwendet werden und die Lesbarkeit der Bäume verbessert wird.

4.7.2 Verhalten des Roboters

4.7.3 Verhalten des Menschen

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 Erweiterung des Robotermodells mit MoveIt

-Fehler durch falsche Pfade des Setups

-Fehler durch fehlerhafte Config (Mal nachsehen, was das genau war)

6.1.3 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 Veränderungen an den verwendeten Objekten innerhalb dieses Systems.

Pluginarchitektur

Die von Gazebo verwendete Plugininfrastruktur ist ideal, um schnell neue Funktionen in Gazebo zu entwickeln. Jedoch existiert damit jedes Plugin im selben Prozess, was bei der Nutzung von Bibliotheken, welche nicht für die mehrfache Nutzung im selben Prozess entsprechend ausgelegt wurden, zu Problemen führen kann.

Zur Kommunikation des ActorPlugins mit dem ROS-ActionServer wurde die benötigte Funktionalität vorerst im Plugin selbst implementiert. Da diese Funktionalität zuerst entwickelt wurde, konnte sie zu diesem Zeitpunkt nur alleinstehend getestet werden. Diese Tests verliefen vorerst erfolgreich, jedoch scheiterten sie bei der Integration des Robotermodells, welches über die `ros_control`-Integration ebenfalls `rclcpp` nutzt.

Um dieses Problem zu umgehen, sind mehrere Lösungsansätze denkbar.

Separater Nachrichtendienst Ein solcher losgelöster Dienst kann die Nachrichten mit einem anderen Programm auszutauschen, welches die Kommunikation nach außen übernimmt.

Nutzung der gleichen ROS-Instanz Um dem Problem zu begegnen, könnte auch eine globale ROS-Instanz von Gazebo verwaltet werden, welche dann von den Plugins genutzt werden kann. Dies könnte als ein Plugin realisiert werden, welches diese anderen Plugins zur Verfügung stellt, jedoch müssten die anderen Plugins zur Nutzung dieser Schnittstelle modifiziert werden.

Gazebo-ROS-Brücke Die Gazebo-ROS-Brücke kann Nachrichten, welche in beiden Formaten definiert wurden, ineinander umwandeln. Dies ermöglicht die Kommunikation über die Brücke als Mittelmann. Dabei müssten Anpassungen an der Architektur vorgenommen werden, da Gazebo kein Äquivalent zum ActionServer besitzt.

Die Entscheidung fiel auf einen separaten Nachrichtendienst, da dessen Implementation losgelöst von anderen Systemen funktioniert. Natürlich ist die Einführung eines weiteren Nachrichtendienstes ein Bruch der Philosophie, jedoch die einzige Methode, die garantiert funktioniert, weswegen sie letztendlich implementiert wurde.

Fehlende Animationsgrundlagen

-Animationen werden als .col bereitgestellt

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

6.1.4 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

6.1.5 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[20] 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

7.3.5 Separieren der Subtrees in eigene Dateien

Literatur

[1] *Arminando/GameRig: GameRig is an auto rigging for games addon for Blender. Built on top of Rigify, it adds rigs, metarigs and additional functionality that enable game engine friendly rig creation. Open source and can be used for personal and commercial projects.* letzter Zugriff: 23.04.2023. URL: https://github.com/Arminando/GameRig.

[2] R. Brooks. „A robust layered control system for a mobile robot“. In: *IEEE Journal on Robotics and Automation* 2.1 (1986), S. 14–23. DOI: 10.1109/JRA.1986.1087032.

[3] *CMake*. letzter Zugriff: 23.04.2023. URL: https://cmake.org/.

[4] *Cobots: der intelligente Roboter als Kollege*. letzter Zugriff: 18.4.2022. URL: https://www.kuka.com/de-de/future-production/mensch-roboter-kollaboration/cobots.

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

[6] Michele Colledanchise und Petter Ögren. *Behavior Trees in Robotics and AI - An Introduction*. 2018. DOI: https://doi.org/10.1201/9780429489105.

[7] Uwe Dombrowski, Tobias Stefanak und Jérôme Perret. „Interactive Simulation of Human-robot Collaboration Using a Force Feedback Device“. In: *Procedia Manufacturing* 11 (Dez. 2017), S. 124–131. DOI: 10.1016/j.promfg.2017.07.210.

[8] Uwe Dombrowski, Tobias Stefanak und Anne Reimer. „Simulation of human-robot collaboration by means of power and force limiting“. In: *Procedia Manufacturing* 17 (2018). 28th International Conference on Flexible Automation and Intelligent Manufacturing (FAIM2018), June 11-14, 2018, Columbus, OH, USAGlobal Integration of Intelligent Manufacturing and Smart Industry for Good of Humanity, S. 134–141. ISSN: 2351-9789. DOI: https://doi.org/10.1016/j.promfg.2018.10.028. URL: https://www.sciencedirect.com/science/article/pii/S2351978918311442.

[9] *FreeCAD: Ihr parametrischer 3D-Modellierer*. letzter Zugriff: 21.04.2023. URL: https://www.freecad.org/index.php?lang=de.

[10] *Gazebo*. letzter Zugriff: 23.04.2023. URL: https://staging.gazebosim.org/home.

[11] *Gazebo*. letzter Zugriff: 23.04.2023. URL: https://app.gazebosim.org/dashboard.

[12] *GDC 2005 Proceeding: Handling Complexity in the Halo 2 AI*. letzter Zugriff: 18.05.2023. URL: https://www.gamedeveloper.com/programming/gdc-2005-proceeding-handling-complexity-in-the-i-halo-2-i-ai.

[13] *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.

[14] *Godot Engine - Free and open source 2D and 3D game engine.* letzter Zugriff: 10.04.2023. URL: https://godotengine.org.

[15] D Isla. *Handling complexity in the halo 2 ai. GDC 2005 Proceedings.* 2005.

[16] 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.

[17] *MoveIt 2 Documentation.* letzter Zugriff: 13.4.2022. URL: https://moveit.picknik.ai/galactic/index.html.

[18] *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.

[19] *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.

[20] *OctoMap.* letzter Zugriff: 13.4.2022. URL: https://octomap.github.io.

[21] *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/.

[22] *Rigify — Blender Manual.* letzter Zugriff: 23.04.2023. URL: https://docs.blender.org/manual/en/3.5/addons/rigging/rigify/index.html.

[23] *Robot simulator CoppeliaSim: create, compose, simulate, any robot - Coppelia Robotics.* letzter Zugriff: 23.04.2023. URL: https://www.coppeliarobotics.com/.

[24] *SDFormat Specification.* letzter Zugriff: 23.04.2023. URL: https://sdformat.org/spec.

[25] *Standard C++.* letzter Zugriff: 23.04.2023. URL: https://isocpp.org.

[26] *The most powerful real-time 3D creation tool - Unreal Engine.* letzter Zugriff: 10.04.2023. URL: https://www.unrealengine.com.

[27] *Unity Real-Time Development Platform | 3D, 2D, VR & AR Engine.* letzter Zugriff: 10.04.2023. URL: https://unity.com.

[28] *urdf/XML ROS Wiki.* letzter Zugriff: 23.04.2023. URL: http://wiki.ros.org/urdf/XML.

[29] *Welcome to Python.org.* letzter Zugriff: 23.04.2023. URL: https://www.python.org.

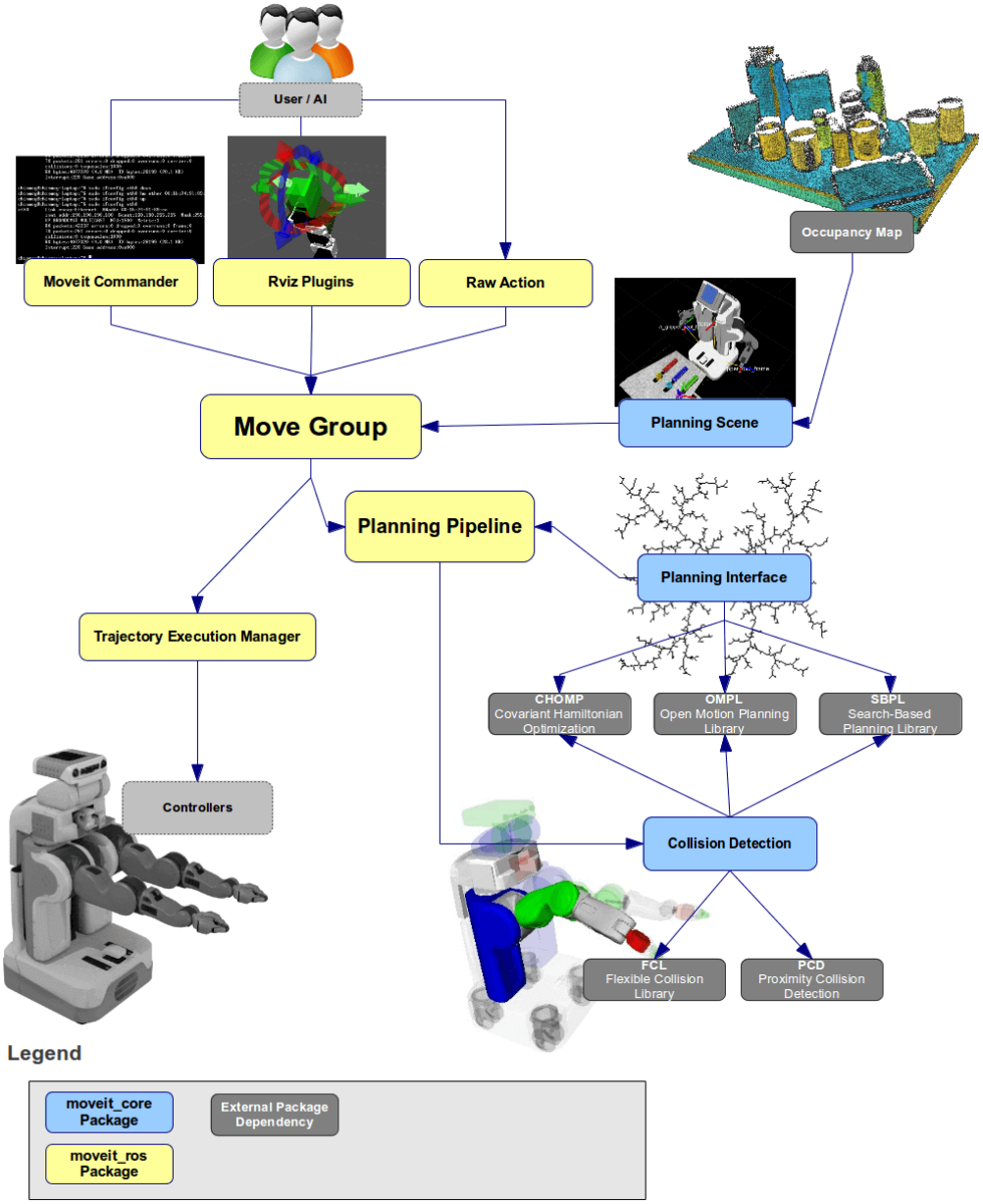

Abbildung 7.1: Visualisierung der MoveIt Pipeline [19]