HTWK
Hochschule für Technik,
Wirtschaft und Kultur Leipzig

# Verwendung von Behavior-Trees in einem industriellen MRK-Szenario

## mechatronisches Forschungsprojekt

von

Bastian M. Hofmann

**Hochschuldozent:** Prof. Dr.-Ing. Jens Jäkel
Fakultät Elektro- und Informationstechnik
Wächterstraße 13
04107 Leipzig

**Hochschulbetreuer:** M.Eng. Christian Rickert
Fakultät Elektro- und Informationstechnik
Wächterstraße 13
04107 Leipzig

**Einreichung:** 19.04.2022

# Inhaltsverzeichnis

# Aufgabenstellung mechatronisches Forschungsprojekt

Eine häufig gewünschte Aufgabe ist die Kollaboration von Roboter und Mensch in einem gemeinsamen Arbeitsraum. Dabei ist das Ziel den Menschen zu unterstützen, Sicherheitsbarrieren durch Sensorik zu ersetzen und dabei flexiblere und individuellere Abläufe zu ermöglichen. Der Roboter muss in diesem Fall Rücksicht auf den Kooperationspartner nehmen, um diesen nicht zu schädigen. Dies erlaubt auch den Einsatz von Mensch und Roboter auf kleinerem Raum.

Um dieses Problem zu simulieren, soll ein Ablauf erdacht werden, in welchem Mensch-Roboter-Kooperation genutzt wird. Dabei soll es sich um einen Ablauf handeln, welcher aufgrund der Natur der Aufgabe aus sich immer wieder ändernden oder teilweise zufälligen Aspekten in der zu erledigenden Aufgabe oder der Umgebung nicht vollständig automatisierbar ist. Ein Mensch kann in solchen Situationen unterstützen, muss jedoch mit der Maschine kooperieren, wobei diese durch Sensoren dessen Position erfassen und auf dessen Aktionen entsprechend reagieren muss.

Ziel der Arbeit ist die Evaluierung von deklarativen Beschreibungssprachen zur Definition von kollaborativem Verhalten, in diesem Fall Erkennung und Vermeidung von Menschen im Arbeitsbereich der Maschine. Das kollaborative Verhalten soll hierfür in kleinere Verhaltens-Einheiten zerlegt werden. Diese Einheiten sollen modular konzipiert werden, um in späteren Projekten weiter eingesetzt werden zu können.

Um dieses Ziel bearbeiten zu können, soll eine Simulation eines Roboters mit dem zu entwickelnden Szenario erstellt werden. Anhand der Simulation soll die Tauglichkeit der Beschreibungssprache getestet werden.

Das Projekt soll dabei folgende Punkte umfassen:

- Literaturrecherche MRK
- Festlegung einer geeigneten Aufgabe für MRK
- Erprobung geeigneter Simulationssoftware
- Auswahl einer deklarativen Beschreibungssprache zur Problembearbeitung
- Erstellen einer virtuellen Roboterumgebung
- Auswahl von Sensoren
- Simulation des Ablaufs der Aufgabe mit:
    - Definiertem Verhalten
    - Probabilistischem Verhalten mit menschlicher Interaktion
- Diskussion der Ergebnisse

# 1 Einleitung

Behavior Trees werden immer häufiger als Beschreibungssprache für Systemverhalten eingesetzt. Sie verdrängen in vielen Bereichen Finite-State-Machines, welche die gleiche Aufgabe erfüllen. Dabei zeichnen sich Behavior Trees vor allem durch eine bessere Übersichtlichkeit aus. Aus diesem Grund wurden diese schon früh in unübersichtlichen Systemen, wie zum Beispiel der "künstlichen Intelligenz" von Videospielen eingesetzt[6]. In diesem Bereich erlangten sie eine starke Dominanz als Beschreibungssprache, da komplexes Systemverhalten auf eine übersichtliche Weise aus anderen, kleineren Bausteinen ausgedrückt werden konnte.

Bei Industrieaufgaben ist auch das Konzept der MRK immer gefragter. Hierbei können 3 unterschiedliche Definitionen unterschieden werden. Mensch-Roboter-Koexistenz bedeutet die gleichzeitige Nutzung eines Raumes durch Mensch und Roboter. Hierbei ist zu beachten, dass deren Arbeitsräume trotzdem getrennt sind. Bei der Mensch-Roboter-Kooperation teilen sich Mensch und Roboter den selben Arbeitsraum, jedoch findet keine Interaktion zwischen beiden statt. Die letzte Art ist die Mensch-Roboter-Kollaboration, bei welcher sowohl Mensch und Roboter in Interaktion treten. Hierbei befinden sie sich im selben Arbeitsbereich und bearbeiten die selbe Aufgabe.[9]

Für alle diese Szenarien sind neue Industrieroboter mit erweiterter Sensorausstattung nötig, um Kollisionen zu vermeiden und aktiv vorzubeugen. Diese neuen Roboter werden auch als Cobots bezeichnet, ein Kofferwort aus "collaborative" und "robot". Sie besitzen zum Beispiel Sensoren zur Erfassung von Drehmomenten an deren Gelenken, um Kollisionen zu erkennen. Natürlich können diese auch für andere Zwecke genutzt werden, wie der einfacheren Programmierung von Bewegungsabläufen durch Bewegen des Arms in die gewünschte Position.[3]

Um diese Felder miteinander zu verbinden, soll in dieser Arbeit eine Möglichkeit erprobt werden, über Behavior Trees das Kollaborationsverhalten von Cobots in einer Simulation zu beschreiben.

# 2 Aufgabe

Dieses Projekt befasst sich mit der Simulation eines Mensch-Roboter-Kollaboration (MRK) Szenarios. Hierfür soll zuerst eine geeignete Aufgabe gefunden werden, welche später in der Simulation umgesetzt wird. Für das Finden eines geeigneten Kollaborationsszenarios ist eine genaue Festlegung der Anforderungen an das Szenario erforderlich. Die Aufgabe soll MRK nutzen, um einen fehleranfälligen Ablauf zu automatisieren. Im Fehlerfall soll ein Mensch den Roboter unterstützen, so dass dieser die Aufgabe weiterhin erfüllen kann. Da sich keine Barrieren im Arbeitsbereich befinden sollen, muss der Roboter mit Sensoren den Mensch erkennen und auf ihn entsprechend reagieren.

Das Szenario einer Pick-And-Place von zum Beispiel Paketwaren ist dabei interessant, da es aufgrund unterschiedlicher Beschaffenheiten und Maßen der Pakete zu Fehlern kommen kann. Auf diese kann der Roboter dann dementsprechend reagieren oder gegebenenfalls vom Mensch unterstützt werden, um seine Arbeit fortsetzen zu können.

Zur Simulation der Aufgabe soll eine geeignete Software verwendet werden, welche Roboter und Mensch unter Zuhilfenahme von Behavior Trees simuliert. Die Auswahl hierfür benötigter Software ist auch Gegenstand der Aufgabe, um diesen Prozess besser nachvollziehen zu können, wird dieser im folgenden Lösungskonzept ausführlich beschrieben.

Ähnliche Projekte sind hierbei zum Beispiel die Steuerung von Robotern mit Sprachkommandos, welche zu Befehlsketten zusammengehängt werden können[2]. Außerdem existiert ein IEEE-Paper, welches Behavior-Trees zur Steuerung von Robotern verwendet[8].

# 3 Lösungskonzept

## 3.1 ROS

Robot Operating System (ROS) ist eine quelloffene Umgebung zur Entwicklung von Roboteranwendungen. Es stellt hierfür eine Buildumgebung, als auch eine Kommunikationsschnittstelle zur Verfügung. Code wird in ROS in Paketen organisiert, welche funktionale Einheiten von zusammenhängender Software darstellen. In größeren Projekten kommt es häufig zu Problemen bei dem Kompilieren von Software auf unterschiedlichen Systemen, da zum Beispiel bestimmte Abhängigkeiten nicht installiert oder im falschen Pfad abgelegt sind. Eine Buildumgebung, wie zum Beispiel die von ROS, kann hierbei auf mehreren Wegen helfen:

**Abhängigkeitsverwaltung:** Äbhängigkeiten von Paketen können automatisiert mit installierten Paketen verglichen werden. Hierdurch können Fehler durch fehlende Pakete vermieden werden. ROS besitzt hierbei auch die Möglichkeit, einige fehlende Pakete automatisch installieren zu können.

**Workspaces:** Pakete werden in Workspaces organisiert, um verschiedene Arbeitsumgebungen gleichzeitig auf einem System verwenden zu können. Jeder Workspace besteht aus einen Ordner mit Quellcode, Kompilationsdaten und installierten Dateien.

**Strukturierung:** Die Workspaces helfen auch in der Strukturierung des Projekts, da Pakete in deren Ordnern separiert werden. Andere benötigte Pakete können durch Referenzierung ihres Paketnamens in den Kompilationsprozess von Paketen eingebunden werden. Außerdem ist der Zugriff auf Dateien von anderen Paketen anhand des Paketnamens und Subpfad möglich.

**Wiederholbarkeit:** Auch die Kompilation erfolgt im Workspace anhand der gegebenen CMakeLists.txt-Dateien. Hierfür werden diese von einem Tool automatisch gelesen und das Paket anhand deren Informationen gebaut. Dies erlaubt einfache Wiederholbarkeit, solange der komplette Quellcode verfügbar ist.

Die Kommunikationsschnittstelle stellt eine vereinheitlichte Kommunikationsebene für alle Nodes in einem Projekt bereit. Hierfür können vorher definierte Nachrichten in unterschiedlichen Wegen übertragen werden:

**Topics** sind eine Übertragungseinrichtung von sogenannten Publishern zu Subscribern. Die Publisher senden hierbei ihre Nachrichten an ein Topic, welche von allen Subscribern des Topics empfangen werden.

**Services** bestehen aus einem Server und einer beliebigen Anzahl an Clients. Clients senden hierbei eine Request-Nachricht an den Server, welcher diese mit einer Response-Nachricht beantwortet.

**Actions** stellen eine lang andauernde Operation durch eine Kombination aus Topics und Services dar. Erst wird ein Ziel als Service-Aufruf an den Actionserver gesendet. Während dessen Ausführung wird Feedback über die Aktion über ein Topic an den Action-Client gegeben. Nach Beendigung der Aktion wird noch ein Ergebnis an den Client in Form eines weiteren Service-Aufrufs übermittelt.

Durch diese Mechanismen können unterschiedlichste Kommunikationszenarien innerhalb eines Projekts umgesetzt werden, ohne dass diese selbst von Grund auf neu implementiert werden müssen.

### 3.1.1 MoveIt

Moveit is ein ROS-Paket das verschiedene Aufgaben erfüllt:

- Bewegungsplanung
- inverse Kinematik
- Robotersteuerung
- Manipulation von Physikobjekten in der Simulation
- Verarbeitung von 3D-Sensordaten

- Kollisionserkennung und Kollisionsvermeidung

In diesem Projekt soll die Roboterbewegung durch MoveIt geplant und ausgeführt werden. Dazu sollen Daten von mehreren Sensoren zu einer sogenannten Octomap zusammengefasst werden, um anhand von dieser Bewegungen planen zu können. Eine Oktomap[10] vereinfacht das Arbeiten mit 3D-Punktwolken, indem Punkte zu quaderförmigen Bereichen zusammengefasst werden. Anhand dieser Bereiche kann nun eine Bewegung einfacher geplant werden, da belegte Regionen anhand ihrer Koordinaten einfacher gefunden werden können.

### 3.1.2 Behavior Tree

Ein Behaviour-Tree ist eine Beschreibungsprache für Systemzustände, ähnlich einer Finite-State-Machine (FSM). Häufig werden Behaviour-Trees in der Spieleindustrie vewendet, um FSM's zu ersetzen. Durch die komplexen Verhaltensweisen in vielen Spielen wurden deren FSM's immer komplexer und undurchsichtiger.

Durch den Einsatz von Behavior-Trees konnten diese Abfolgen in einem anderen, quelltextunabhängigem Format gespeichert werden.

Hierdurch verbesserte sich die Übersichtlichkeit, Lesbarkeit und Anpassbarkeit dieser Verhalten. Ein Behaviour-Tree verhält sich wie ein umgedrehter Baum und besteht aus mehreren sogenannten Nodes. Von einer Root-Node aus werden weitere Nodes hinzugefügt, welche das gewünschte Verhalten abbilden. Jede Node gibt während und nach deren Ausführung Statusinformationen an die Node über ihr zurück. Dies erlaubt der überliegenden Node auf ihr sogenanntes Child zu reagieren, um so komplexeres Verhalten darzustellen.

Die Anordnung der Children unter einer Node bestimmt, wann diese ausgeführt werden. Hier erfolgt meißt eine Ausführung vom ersten zum letzten Child. Bei einigen Nodes werden speziellere Auswahlverfahren genutzt, um das ausgeführte Child zu bestimmen.

Diese Darstellungsform ist bei komplexeren Projekten daher lesbarer als eine equivalente FSM. Die Abkapselung einzelner Funktionen in Module für Behaviour-Trees erleichterte auch das Wiederverwenden von bereits existierendem Code. Ein weiterer Vorteil ist der Kontrollfluss, welcher in einer FSM von oben nach unten erfolgt, in einem B-Tree jedoch zuerst nach unten, um von dort den Rückgabewert wieder nach oben zu übertragen.

### Beispiel

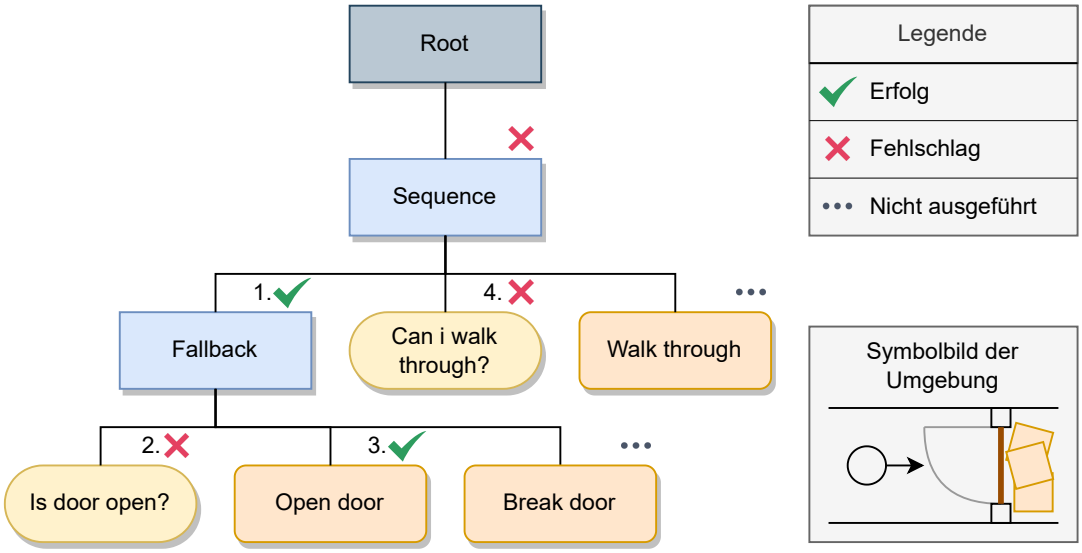


Abbildung 3.1: Modellierung einer Türdurchquerung als Behavior Tree

Der Kontrollfluss des Beispiels in Abbildung 3.1 erfolgt zuerst von der Root-Node zur unterliegenden Sequence. Diese führt ihr erstes Child, eine Fallback-Node, aus. Diese führt ihre Children von links nach rechts aus, bis das erste Child Erfolg zurückmeldet. Die Fallback-Node gibt diesen Erfolg an die darüberliegende Sequence weiter, wodurch die Sequence das nächste Child ausführt. In diesem Fall ist das Durchqueren der Tür nicht möglich, da die Tür auf der anderen Seite zugestellt ist, weswegen die Ausführung der Sequence beendet wird.

### Node-Arten

Um beliebige Vorgänge abbilden zu können, mussen sie in einen Baum umformbar sein. Hierfür wurden mehrere Arten von Nodes geschaffen, welche diese unterschiedlichen Fälle abdecken.

**Action-Nodes** sind Auslöser für Veränderungen des repräsentierten Systems. Diese Actions können entweder sofort, oder über einen unbestimmten Zeitraum lang ausgeführt werden. Der Zustand der Action wird als Rückgabewert der Node ausgedrückt.

**Condition-Nodes** fragen Daten ab, ohne den Systemzustand zu verändern. Auch hier wird der abgefragte Zustand als Rückgabewert der Node repräsentiert.

**Decorator-Nodes** werden verwendet, um den Rückgabewert eines Childs zu verändern. Sie besitzen immer genau eine Child-Node, wessen Rückgabewert modifiziert werden soll. Standartmäßig ist das Invertieren, Forcieren von Erfolg/Fehlschlag und Wiederholen möglich.

**Control-Nodes** sind Nodes mit mindestens einer weiteren Node als sogenanntes Child. Diese Child-Nodes werden durch die Control-Node ausgeführt und die Rückgabewerte weiter verarbeitet. Hierbei kann in verschiedene Untergruppen unterschieden werden:

**Sequence** Als Sequence werden Nodes bezeichent, welche nach erfolgreicher Ausführung eines Childs das nächste in der Reihe aufrufen. Wurden alle Child-Nodes erfolgreich ausgeführt, gibt die Node auch das Erfolgssignal zurück. Sollte hingegen eine Child-Node fehlschlagen, gibt die Sequence sofort das Fehlsschlagssignal zurück und fängt wieder von vorn an. Es existiert auch eine ReactiveSequence, welche alle vorherigen Nodes jeden Tick noch einmal überprüft. Dies kann eingesetzt werden, um bestimmte Aktionen abbrechen zu können, falls sich vorherige Bedingungen ändern.

**Fallback** Um eine alternative Strategie nach einem Fehlschlag einer Action oder einer Condition zu realisieren, können Fallback-Nodes genutzt werden. Diese rufen nach dem Fehlschlag eines Childs das nächste in der Reihe auf. Das Erfolgssignal wird gesendet, wenn ein Child das Erfolgssignal zurückgibt. Nur wenn alle Child-Nodes fehlschlagen, wird die Fallback-Node auch das Fehlsschlagssignal senden und wieder von vorn anfangen. Auch hier existiert ein ReactiveFallback, was die aktuelle Aktion abbricht, wenn eine vorherige Child-Node ein Erfolgssignal zurück gibt.

### Trees

Das Verhalten der Person, später auch Actor genannt, als auch das Verhalten des Roboters soll durch Behavior-Trees ausgedrückt werden. Hierfür sollen 2 Trees erstellt und gleichzeitig ausgeführt werden:

**Actor-Tree** Der Actor stellt die Person, welche sich im Raum mit dem Roboter befindet, dar. Die Aufgabe des Actors ist dabei, dem Roboter zu helfen, wenn dieser Hilfe

benötigt. Um dies in der Simulation umzusetzen, wird er sich dazu in den Sicheheitsbereich des Roboters begeben, was diesen zu einer Reaktion auf die Person zwingt. Dieser idealisierte Ablauf ist in Abbildung 3.2 dargestellt.

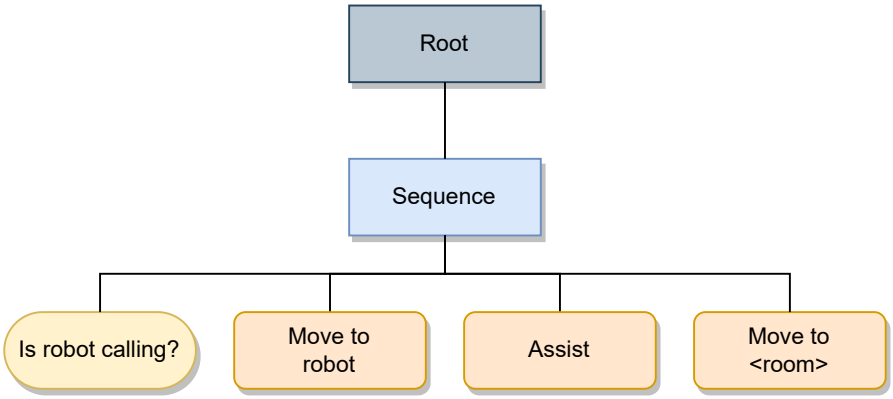


Abbildung 3.2: Idealer Ablauf des Actors

Da eine Person während dieser Tätigkeit noch weitere Aufgaben erfüllen kann, soll auch dies simuliert werden. Hierfür soll der Actor sich im Arbeitsbereich des Roboters aufhalten und umherbewegen. Mit einer bestimmten Wahrscheinlichkeit soll er auch in den Warn- und Sicherheitsbereich des Roboters laufen. Dies sorgt dafür, dass dieser nicht nur bei Hilfeleistung auf den Actor reagieren muss, sondern auch im normalen Betrieb.

**Roboter-Tree** Der Roboter soll Objekte von einer Platform auf eine andere Platform bewegen, um einen Entpackvorgang zu simulieren. Hierbei soll er bei Betreten des Warnbereiches seine Geschwindigkeit reduzieren. Sollte der Actor den Sicherheitsbereich betreten, ist der Verfahrvorgang sofort zu stoppen, um eine weitere Gefährdung auszuschließen. Bei Fehlern im Arbeitsablauf soll die Person um Hilfe gerufen werden, welche das Problem behebt. Dieser geplante Behavior Tree ist in Abbildung 3.3 dargestellt.

## 3.2 Gazebo

Gazebo ist eine Simulationsumgebung für Roboter, welche diese als Physikobjekte simulieren kann. Hierfür wird zum Start der Software eine Welt aus einer Datei geladen. Um die darin definierten Modelle und/oder Roboter zu bewegen, ist die Kommunikation mit

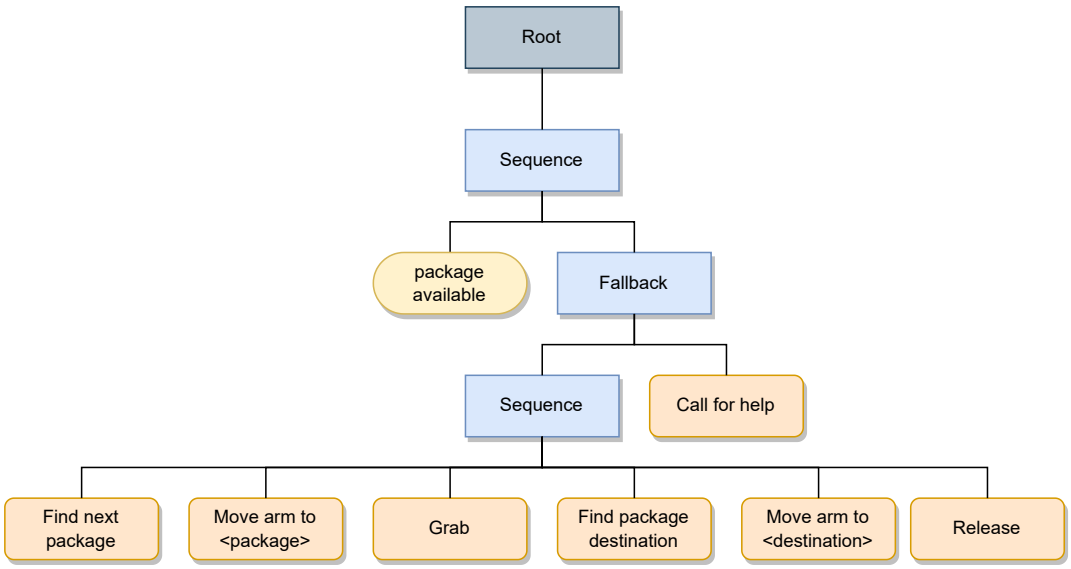


Abbildung 3.3: Idealer Ablauf des Roboters

anderen Systemen wie ROS nötig, welche die Sensordaten abfragen und Steuerdaten liefern. Die Übertragung dieser Daten ist in Gazebo durch Plugins realisiert, welche mit den entsprechenden Objekten verbunden werden. Für die Kommunikation und Konvertierung der Daten zwischen ROS und Gazebo existieren bereits zahlreiche Plugins[11]. Durch diese werden dann Daten, wie zum Beispiel von Laserentfernungsmessern und 3D-Kameras, aus der Simulation erhoben und an ROS als Topic weitergeben. Außerdem ist damit auch die Steuerung und Regelung von Robotern möglich, welche die Simulationsumgebung beinflussen können.

### 3.2.1 Sensoren

In diesem Projekt werden Sensoren zur Erkennung von Personen im Arbeitsbereich des Roboters benötigt. Einfache Lasersensoren sind dabei sehr zuverlässig, können jedoch nur die grobe Position einer Person ermitteln, da sie nur eine Ebene erfassen können. Um mehr Daten über die genaue Körperposition ermitteln zu können, sollen weitere Sensoren in Form von Tiefenkameras genutzt werden. Diese können durch ihren Aufbau auch räumliche Positionen erkennen. Dadurch können auch Körperpositionen erfasst werden, welche im Laserscanner durch dessen Position sicht erfasst werden können.

## 3.3 Roboter

Für ein solches MRK-Szenario existieren bereits spezielle Robotermodelle, welche für diesen Anwendungsfall günstige Eigenschaften besitzen. Hierbei sind vor allem Drehmomentsensoren in den Roboterachsen wichtig, da diese bei Kollision mit anderen Objekten oder Menschen den Roboter anhalten können. Da das Projekt zu einem späteren Zeitpunkt mit einem realen Roboter getestet werden soll, ist die Verwendung des gleichen Roboters in der Simulation vorteilhaft. Für dieses Projekt wurde deshalb der Kuka iisy[7] gewählt, da es sich bei diesem Roboter um einen Cobot handelt, welcher auch für Tests zur Verfügung steht.

Da dieser Roboter erst dieses Jahr eingeführt wurde, ist die Dokumentation noch nicht öffentlich erhältlich. Außerdem sind Modelle für die Simulation noch nicht verfügbar, weswegen diese selbst erstellt werden müssen. Auch die Parametrisierung für MoveIt muss für diesen Roboter noch erfolgen.

## 3.4 Zusammenfassung

Der geplante Systemaufbau besteht aus dem Management, welches die Aktionen und Reaktionen von Roboter und Actor verwaltet. Hierzu wird auf Actorseite ein Behavior Tree eingesetzt, welcher diesen direkt steuert. Auch auf Roboterseite soll ein Behavior Tree eingesetzt werden, welcher Steuerbefehle an MoveIt weitergibt. MoveIt plant aus diesen Befehlen nun die Bewegungen des Roboters und führt durch die virtuellen Aktoren in der Simulation aus.

Auf Simulationsseite werden Befehle für den Actor und die simulierten Aktoren umgesetzt. Sensoren geben dabei Feedback über das aktuelle Geschehen, was durch MoveIt zur Kollisionsvermeidung eingesetzt werden kann.

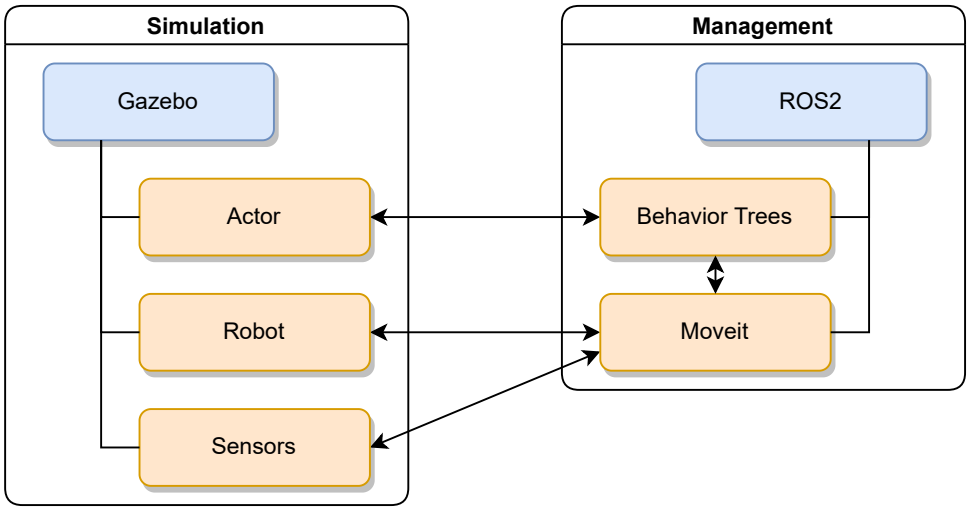


Abbildung 3.4: Visualisierung des geplanten Systemaufbaus

# 4 Umsetzung

## 4.1 ROS

### 4.1.1 Auswahl der ROS-Umgebung

Aktuell existieren mehrere Versionen der ROS-Umgebung, welche aktuell noch unterstützt werden. Der größte Unterschied dieser Versionen liegt in der Buildumgebung. In ROS wird `catkin` als Buildtool verwendet, was in ROS 2 durch `colcon` ersetzt wurde. Hierdurch ergeben sich einige Veränderungen in Aufbau und Nutzung der Workspaces. Häufig kommt es durch einen solchen Versionssprung auch zu Problemen mit bereits bestehenden Paketen, welche erst auf neue Systeme umgestellt werden müssen, bevor diese wieder verwendet werden können. In ROS 1 wird für die Kommunikationsebene TCP/IP verwendet, UDP/IP ist nur über das `roscpp`-Paket verfügbar. ROS 2 hingegen unterstützt sowohl TCP/IP, UDP/IP und sogar Quality of Service. Dies sorgt für kleinere Unterschiede bei der verwendung von Topics, welche nun zusätzliche Daten benötigen. Da dieses Projekt auf keinem voherigen Stand basiert, ist die Auswahl der ROS-Umgebung nur von den zu verwendenden Paketen abhängig.

### 4.1.2 Gazebo

Auch Gazebo existiert in 2 Versionen, welche jedoch von ROS unabhängig sind. Hier handelt es sich um das ältere Gazebo 11 und das neuere Gazebo Ignition. Beide unterstützen den Anwendungsfall, jedoch ist die Dokumentation von Gazebo Ignition noch nicht auf dem selben Stand wie dessen Vorgänger. Zuzüglich ist zu beachten, dass sich die Physiksimulation zwischen beiden Programmen geändert hat, was zu unterschiedlichen Simulationsergebnissen führen kann.

Gazebo Ignition setzt bei der Simulation auf eine weiterentwickelte Engine und größeres Pluginsystem, da selbst die Nutzeroberfläche von Gazebo hier auf Plugins basiert.

Gazebo 11 hingegen nutzt eine etwas ältere Engine. Auch hier werden Plugins in der Simulation eingesetzt, sind jedoch nur in den simulierten Komponenten zu finden, das Benutzerinterface ist hier statisch.

Bei beiden Versionen kam es bei der Simulation von Lasersensoren zu Problemen, da diese keine Daten lieferten. Diese Sensoren existieren in CPU- und GPU-Ausführung. Die CPU-Ausführung beachtet nur Kollisionen mit den Modellen, welche auch für die Physiksimulation genutzt werden. Die GPU-Variante betrachtet keine Kollisionen der Strahlen mit den Modellen der Physiksimulation, sondern nur mit den graphisch visualisieten Modellen der Simulation. Da der Actor animiert ist, besitzt dieser in Gazebo keine physikalischen Kollisionsmodelle. Dies schließt die Nutzung der CPU-basierten Sensoren aus. Die GPU-basierten Lasersensoren können diesen erfassen, jedoch erfassten sie nach ersten Tests keinerlei Kollisionen mit dem Actor oder der Raumgeometrie. Hierbei handelte es sich um einen Bug, welcher durch das Starten der Serverkomponente von Gazebo ohne Grafikbeschleunigung behoben werden konnte.

Sobald dieser Fix gefunden wurde, konnte eine finale Welt für die Simulation in Form einer .sdf-Datei erstellt werden. In dieser Welt befinden sich der Raum selbst, die Lasersensoren, der Actor und der Roboter. Der Roboter wird dabei aus anderen Definitiondateien erstellt, und an einer bestimmten Position in der simulierten Welt verankert.

Um den Kuka iisy in MoveIt und Gazebo nutzen zu können, mussten hierfür Modelle erstellt werden. Hierzu wurden Daten aus den Datenblättern und Designfiles des Roboters entnommen. Diese wurden im Fall der Modelle manuell bearbeitet und konvertiert, um sie in Gazebo nutzbar zu machen. Außerdem wurden weitere Meshes zur Kollisionsprüfung manuell erstellt, um die Geschwindigkeit der Kollisionserkennung zu erhöhen.

#### .sdf

SDF-Dateien[12] werden zur Erstellung von Modellen in Gazebo genutzt. Jedes Modell besteht dabei aus bestimmten Komponenten, welche dessen Eigenschaften genau beschreiben. Außerdem kann jedes Modell mit sowohl visuellen, als auch physikalischen Modelle versehen werden. Hierbei gilt es, die Art der Sensoren zu beachten, da CPU-basierte Sensoren die physikalischen Modelle nutzen. Die GPU-basierten Sensoren erkennen jedoch nur die visuellen Modelle, welche von den physikalischen Modellen abweichen können. Außerdem ist zu beachten, dass Kollisionserkennung auf der CPU deutlich langsamer ist, weshalb die physikalischen Modelle aus möglichst wenigen Punkten bestehen sollten.

Modelle können hierfür sowohl als Standartformen (Box, Kugel und Zylinder), aber auch als Meshes hinterlegt werden.

#### .urdf

In diesen Dateien werden die Roboterdaten für die Simulation gespeichert. Dazu gehören Aktuatoren, deren Anordnung im Modell und Modelle für sowohl die Physiksimulation, als auch die visuelle Anzeige. Das Format ähnelt dabei den .sdf-Dateien stark, jedoch sorgen kleine Änderungen für grundsätzliche Inkompatibilität[13]. Diese Änderungen sind vor allem in den Offsets und den physikalischen/visuellen Kollisionen zu finden. Außerdem unterstützt dieses Dateiformat zusätzliche Einträge für Plugins und deren Konfiguration, welche in .sdf nicht unterstützt werden. Hierbei handelt es sich vor allem um die Steuerung/Regelung der Aktuatoren.

#### ros_actor_plugin

Dieses Plugin wurde entwickelt, um die Steuerung eines simulierten Actors durch zwei ROS-Topics zu ermöglichen. Im ersten Topic wird die aktuelle Position des Actors gesendet, im zweiten Topic kann dem Actor eine neue Zielposition gesendet werden. Sobald eine neue Zielposition an den Actor übermittelt wird, beginnt dieser mit dem Drehen des Körpers in Richtung der Zielposition. Nach Abschluss dieser Drehung beginnt der mit der Bewegung zum eigentlichen Ziel. Um dieses Plugin verwenden zu können, muss es an einem Actor in der .sdf-Datei referenziert werden. Hiebei können die erstellten Topics umgeleitet werden, um mehrere Actors separat steuern zu können.

### 4.1.3 MoveIt

MoveIt wird als Paket für sowohl für ROS (MoveIt) als auch ROS 2 (MoveIt 2) angeboten. Der größte Unterschied zwischen beiden Versionen ist die Verfügbarkeit des Setup-Assistenten für selbst erstellte Roboter[4]. Dieser funktioniert nur in MoveIt, jedoch nicht in MoveIt 2. Weitere Komponenten wurden zwar verändert, funktionieren jedoch ähnlich oder wurden lediglich umbenannt. Die Entscheidung zwischen beiden Komponenten fiel größtenteils nach Studium der Dokumentation, welche MoveIt 2 als sowohl aktuellste, als auch als empfohlene Version ausweist.

Um die Funktionen von MoveIt zu nutzen, muss das Robotermodell für Gazebo um weitere Daten erweitert werden. Bei komplexeren Robotern ist es manchmal sinnvoll, wenn bestimmte Teile des Roboters als separate Gruppe zu einem bestimmten Endpunkt bewegt werden können. Deshalb werden in dieser Datei Gruppen definiert, welche durch MoveIt später separat geplant werden können. Außerdem können bestimmte Positionen der Aktuatoren als Zustand gespeichert werden, um diese zum Beispiel als Ausgangsposition zu verwenden. Diese Position aller Gelenke wird häufig in der Gruppe `home` vermerkt. Durch Analyse der in den Gruppen befindlichen Aktuatoren können bestimmte Kollisionsabfragen übersprungen werden, wenn diese im Modell nicht auftreten können. Hierzu werden vorher durch die Simulation einer großen Menge von Roboterpositionen Aktuatoren gefunden, welche sich gegenseitig nicht überschneiden können.

Die so gewonnenen Daten werden beim Setup automatisch vermerkt, und als Ausschlusskriterien für die Kollisionsprüfung verwendet. All diese Daten werden in einer .srdf-Datei vermerkt, um sie später am Beginn der Simulation laden und nutzen zu können.

### 4.1.4 Behaviour-Trees

Zur Implementierung von Behavior Trees soll die C++-Bibliothek `BehaviorTree.CPP` genutzt werden. Diese Bibliothek wurde für Roboteranwendungen geschrieben, wird jedoch auch in anderen Feldern eingesetzt und ist gut dokumentiert. Um die gewünschten Verhalten abbilden zu können, müssen Nodes erstellt werden, welche die gewünschten Aktionen, Abfragen und Kontrollstrukturen definieren. Diese werden in einer .xml-Datei als Liste abgelegt, um sie bei der Erstellung der Trees laden zu können. Um die einfache Editierbarkeit zu gewährleisten, erfolgt die Erstellung der Trees mit dem graphischen Tool Groot. In diesem können Trees erstellt und sogar überwacht werden, die Überwachung wurde in diesem Projekt jedoch nicht implementiert. Der Editor speichert dann die erstellten Trees in Form einer .xml-Datei, welche zur Laufzeit des Programms geladen wird.

#### Erstellte Nodes

Zur Erstellung der zu erstellenden Bäume sind mehrere Action- und Condition-Nodes nötig, welche zuerst definiert wurden. Dabei werden folgende grundlegenden Funktionen benötigt:

| Actor | Robot |
|---|---|
| - zu Ziel in Zone bewegen<br>- Abfrage von Hilfestatus<br>- Setzen von Hilfestatus<br>- Zufällige Auswahl von Nodes | - zu Ziel in Zone bewegen<br>- Setzen von Hilfestatus<br>- Abfrage der Zone des Actors<br>- Setzen der Verfahrgeschwindigkeit<br>- Zufällige Fehler |

Einige dieser Nodes besitzen ähnliches Funktionen, was ähnliche Implementationen zur Folge hätte. Stattdessen können diese Funktionen in ihre Bestandteile aufgeteilt werden, um die Wartbarkeit des Codes zu erhöhen und Fehler zu reduzieren. Hierzu wird das das Ziel in einfachere Teilvorgänge zerlegt, welche universeller einsetzbar sind. Zum Beispiel sollen sowohl Actor als auch Roboter sich zu einem Ziel in einer Zone bewegen können. Hierfür könnte aber auch ein Ziel in einer Zone generiert werden, welches dann von einer anderen Node an Roboter oder Actor weitergegeben werden kann, welche dann die Bewegung ausführen.

Im Behavior Tree werden außerdem mehrere Datenstrukturen verwendet, um Daten zwischen Nodes austauschen zu können:

**Pose:** Eine Pose beschreibt die Position und Rotation eines Objekts im Raum durch ein Quaternion [5]. Die Position wird dabei als 3-Dimensionaler, die Rotation als 4-Dimensionaler Vektor abgebildet.

**Position2D:** Eine Position im 2 dimensionalem Raum, dargestellt durch eine X- und Y-Koordinate.

**Area:** Eine Zone im Raum, dargestellt durch “Triangle Strips”[1] . Diese enthalten eine Liste von Koordinaten, welche zu Dreiecken verbunden werden, welche die gewünschte Fläche abbilden.

Unter Zuhilfenahme dieser Datentypen können nun Nodes entworfen werden, welche das gewünschte Verhalten abbilden:

**MoveActorToTarget** sendet ein Bewegungssignal anhand einer gegebenen Ziel-Pose an den Actor. Hierfür wird die Zielposition an das `gazebo_ros_actor`-Plugin übertragen, welches dann den Actor bewegt. Diese Node wartet auf das Erreichen der Zielposition durch den Actor und gibt in diesem Fall den Erfolgsstatus zurück.

**RobotMove** Ähnlich wie MoveActorToTarget wird auch hier eine Zielpose an den Roboter weitergegeben. Jedoch wird die Zielposition hier an MoveIt weitergegeben, welches

daraus einen Bewegungsablauf für den Arm plant. Dieser wird dann als eine ROS-Action an die Controller weitergegeben, welche die Bewegung ausführen. Auch hier wird auf den Erfolg der Bewegung gewartet und dieser als Status der Node zurückgegeben.

**GenerateXYPose** generiert aus einer gegebenen Area eine Pose, welche in der Area liegt. Hierfür werden die in den "Triangle Strips" definierten Dreiecke nach Größe gewichtet, und ein Dreieck zufällig ausgewählt. In diesem Dreieck wird dann ein zufälliger Punkt ausgewählt, welcher die X und Y Koordinaten für die generierte Pose liefert.

**WeightedRandom** Diese Node wählt anhand einer vorgegebenen Wichtungstabelle eines ihrer Children zufällig aus, um es dann auszuführen. Sobald das Child fertig ausgeführt wurde, wird beim nächsten Aufruf wieder zufällig ausgewählt.

**OffsetPose** Der Roboter benötigt in der z-Achse verschobene Positionen, da er sich auf einem Tisch befindet. Diese Node ermöglicht das Verschieben der Positon über einen Offset und das Überschreiben der Rotationsdaten der Pose.

**RandomFailure** modelliert die Fehler, welche dem Roboter bei der Ausführung seiner Aufgabe wiederfahren können. Hierfür kann eine Fehlerwahrscheinlichkeit festgelegt werden, welche die Wahrscheinlichkeit des Erfolgs der Node verändert.

**InAreaTest** testet, ob sich eine gegebene Pose in einer gegebenen Area befindet. Hierfür wird überprüft, ob sich die X- und Y-Koordinaten innerhalb einem der Dreiecke der Area befinden.

**SetRobotVelocity** erlaubt das Verändern der Verfahrgeschwindigkeit des Roboters. Hierzu wird der aktuelle Verfahrvorgang abgebrochen und die selbe Bewegung mit einer geringeren Geschwindigkeit erneut geplant und ausgeführt.

**IsCalled** fragt ab, ob der Roboter aktuell Hilfe benötigt, da es zu einem Fehler in der Ausführung der aktuellen Aktion kam.

**SetCalledTo** verändert diesen Hilfezustand. Dies geschieht beim Hilferuf selbst durch den Roboter und nach geleisteter Hilfe durch den Actor, um den Zustand zurückzusetzen.

**InterruptableSequence** verhält sich ähnlich zu einer normalen Sequence, jedoch merkt sich diese bei Abbruch durch höherliegende Nodes ihre Position. Beim nächsten

Aufruf fängt sie dann an der gemerkten Position wieder an, anstatt wie eine Sequence erneut von vorn zu beginnen.

### Besonderheiten der Implementationen

Da sowohl der Actor als auch der Roboter jeweils einen eigenen Behavior Tree besitzen, mussen diese parallel ausgeführt werden. Hierfür kommen Threads zum Einsatz, welche beide Bäume parallel ausführen. Ein weiterer Thread wird verwendet, um Informationen über den Systemzustand an beide Bäume weiterzugeben. Hierbei wird aktuell auf Synchronisationsmechanismen zwischen den Threads verzichtet, was durch die Verwendung von Pointern zur Datenübergabe ermöglicht wird.

Eine weitere Besonderheit der Implementation ist die Verwendung eines Proxy-Objekts zum Zugriff auf MoveIt. Dies emöglicht den Zugriff von mehreren Nodes auf die Planungs- und Ausführungsfunktionen von MoveIt, wie in Abbildung 4.1. Nötig ist diese Verhalten, um die Geschwindigkeit der gewünschten Bewegung anpassen zu können. Hierfür ist eine erneute Planung des Bewegungspfades erforderlich. Da aber eine Bewegung bereits ablaufen kann, während die Geschwindigkeit geändert werden soll, muss die restliche Strecke mit der neuen Geschwindigkeit erneut geplant werden.

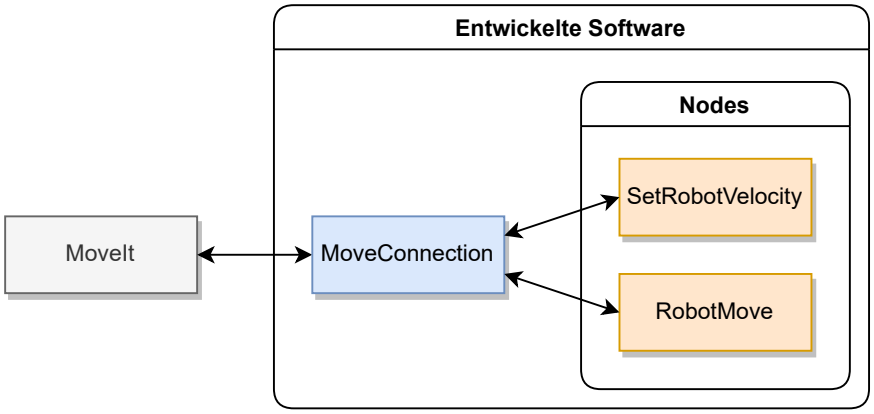


Abbildung 4.1: Visualisierung der Proxy-Funktion

# 5 Ergebnisse

Die Aufgabe konnte in der Simulationsumgebung umgesetzt werden. Durch unvorhergesehene Probleme mit dem verwendeten Softwarestack kam es zu Einschränkungen, auf welche im folgenden auch eingegangen wird.

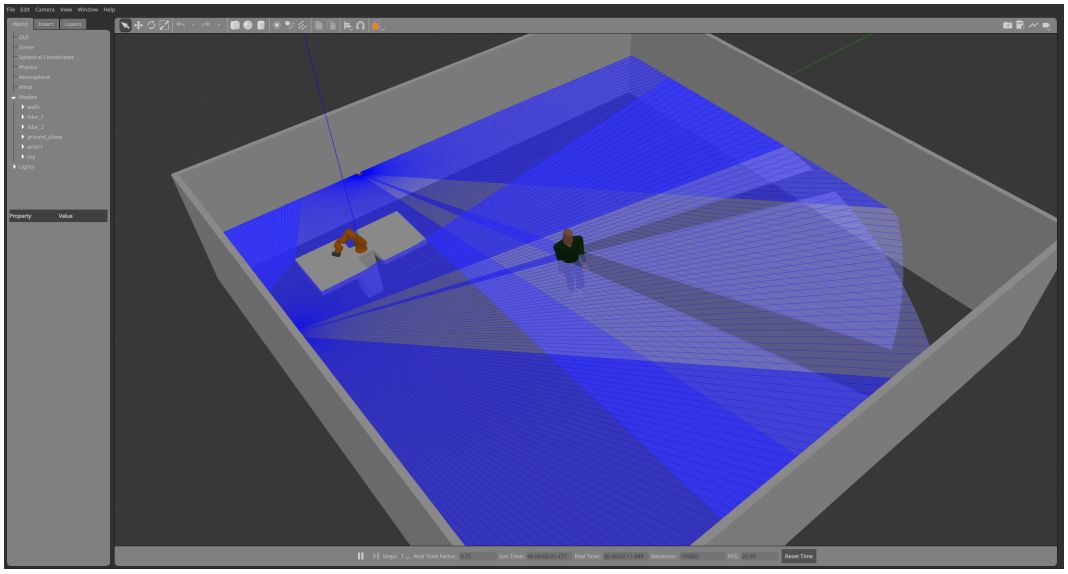

Abbildung 5.1: Simulation in Gazebo

Es wurden sowohl Gazebo Ignition als auch Gazebo getestet, um die Simulation der ausgewählten Aufgabe zu übernehmen. Dabei funktionierte Gazebo Ignition leider nicht zufriedenstellend, da beide Arten von simulierten Lasersensoren nicht funktionieren. In Gazebo funktionierten diese ursprünglich auch nicht, jedoch konnte dieser Fehler durch deaktivieren der GPU-Beschleunigung für die Serverkomponente von Gazebo gelöst werden.

Vorerst funktionierte auch die Bereitstellung der Transformationsdaten der Lasersensoren durch Gazebo nicht, weshalb diese manuell in die dementsprechenden Dateien eingetragen wurden. Nach dieser Änderung waren die Lasersensoren voll funktionsfähig.

Die so gewonnenen Daten müssen nun an MoveIt übergeben werden, welches aus diesen eine Octomap generiert. Diese kann dann zur Kollisionsvermeidung verwendet werden. Das System sollte vorerst mit den Lasersensoren getestet werden, um es dann später mit Tiefenkameras auszustatten. Diese Tiefenkameras sollten benutzt werden, um andere Objekte, die nicht in der Ebene der Lasersensoren liegen, erkennen zu können. Durch das veränderte Format von MoveIt 2 und das alte Setup ist es jedoch nicht möglich, eine Einrichtung dieser Funktion vorzunehmen[4]. Auch manuelle Versuche durch Anpassen der Dateien anhand der Dokumentation verblieben erfolglos. Bei Verfügbarkeit eines Setup-Assistenten für MoveIt2 sollte das Nachrüsten der Funktionalität schnell möglich sein, da die benötigten Topics und Transformationsdaten bereits existieren. Wegen dieses Problems mit der MoveIt-Anbindung wurden auch die geplanten Tiefenkameras noch nicht platziert, da deren Punktwolken mit der aktuellen Version nicht nutzbar sind.

Um trotzdem eine Simulation durchführen zu können, wurden die Positionsdaten des Actors als Topic verfügbar gemacht. Hierzu wurde das Actor-Plugin um ein Topic erweitert, welches bei Bewegung des Actors die aktuelle Position an andere ROS-Komponenten weitergeben kann. Diese Daten können nun zur Positionserkennung genutzt werden, ohne die OctoMap in MoveIt zu verwenden.

# 6 Diskussion

Bei der aktuellen Implementation ist die Bewegungsplanung durch MoveIt durch den Versionssprung auf MoveIt2 noch nicht möglich. Diese kann, sobald der Setup-Assistent[4] funktioniert, nachgerüstet werden. Alle bereits existierenden Laserscanner-Punktwolken können dann transformiert und mit MoveIt verbunden werden. Die Implementation von Tiefenkameras ist möglich, da diese auch Punktwolken ausgeben können, und somit mit entsprechenden Transformationsdaten versehen verwendet werden können. Hierbei ist zu beachten, dass die simulierten Sensoren wahrscheinlich langsamer als eingestellt sind. Dies ist auf den Workaround für Gazebo zurückzuführen, da dieser die GPU-Beschleunigung deaktiviert. Bei der geringen Anzahl an Raycasts eines Lasersensors ist dies noch kein Problem, jedoch müssen Tiefenkameras weitaus mehr Raycasts durchführen, um komplette Tiefenbilder zu generieren.

Eine Anbindung an andere Erkennungssoftware ist auch denkbar, diese kann einfach über das Positionstopic mit der aktuellen Implementation verbunden werden. Außerdem kommt die Erweiterung der Nodes um mehrere Actor-Positionen infrage, da sich mehr als eine Person im Raum aufhalten kann.

# Literatur


[1] *Android Lesson Eight: An Introduction to Index Buffer Objects (IBOs)*. letzter Zugriff: 5.4.2022. URL: https://www.learnopengles.com/tag/triangle-strips/.

[2] Alexandre Angleraud, Quentin Houbre und Roel Pieters. „Teaching semantics and skills for human-robot collaboration“. In: *Paladyn, Journal of Behavioral Robotics* 10.1 (2019), S. 318–329. DOI: doi:10.1515/pjbr-2019-0025. URL: https://doi.org/10.1515/pjbr-2019-0025.

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

[4] *Fortschritt der Entwicklung von MoveIt 2*. letzter Zugriff: 5.4.2022. URL: https://docs.google.com/spreadsheets/d/1aPb3hNP213iPHQIYgcnCYh9cGFUlZmi_06E_9iTSsOI/edit#gid=0.

[5] *Humane Rigging 03 - 3D Bouncy Ball 05 - Quaternion Rotation*. letzter Zugriff: 5.4.2022. URL: https://www.youtube.com/watch?v=4mXL751ko0w.

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

[7] *LBR iisy*. letzter Zugriff: 18.4.2022. URL: https://www.kuka.com/de-de/produkte-leistungen/robotersysteme/industrieroboter/lbr-iisy.

[8] Alejandro Marzinotto u. a. „Towards a unified behavior trees framework for robot control“. In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. 2014, S. 5420–5427. DOI: 10.1109/ICRA.2014.6907656.

[9] *Mensch-Roboter-Kollaboration: Willkommen, Kollege Roboter!* letzter Zugriff: 18.4.2022. URL: https://www.kuka.com/de-de/future-production/mensch-roboter-kollaboration.

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

[11] *Port gazebo_ros_pkgs to ROS 2.0*. letzter Zugriff: 13.4.2022. URL: https://github.com/ros-simulation/gazebo_ros_pkgs/issues/512.

[12] *SDFormat Specification*. letzter Zugriff: 13.4.2022. URL: http://sdformat.org/spec.

[13] *Tutorial: Using a URDF in Gazebo*. letzter Zugriff: 13.4.2022. URL: http://gazebosim.org/tutorials/?tut=ros_urdf.