Hochschule Karlsruhe
University of
Applied Sciences

Fakultät für
Maschinenbau und
Mechatronik

F&E Projekt

SEW-AGV und Igus ReBeL Roboter

Hochschule Karlsruhe (HKA)

Fakultät für Maschinenbau und Mechatronik

Mathias Fuhrer

Hannes Bornemann

Robin Wolf

Betreut durch Simon Holzhauer und Prof. Dr.-Ing. Philipp Nenninger

13.08.2024

Inhaltsverzeichnis

Seite

Abkürzungsverzeichnis

2D ..... *zweidimensional*

3D ..... *dreidimensional*

AGV ..... *Automated Guided Vehicle*

AMCL ..... *Adaptive Monte Carlo Localization*

API ..... *application programming interface*

ca ..... *circa*

CAN ..... *Controller Area Network*

DIO ..... *Digital Input/Output*

DoF ..... *Degrees of freedom*

IK ..... *Inverskinematik*, *Inverse Kinematik*

iRC ..... *igus Robot Control*

KDL ..... *Kinematics and Dynamics Libary*

KLT ..... *Kleinladungsträger*

LIN ..... *Linear*

m ..... *Meter*

mm ..... *Millimeter*

OPML ..... *Open Motion Planning Library*

PRM ..... *Probabilistic Roadmaps*

PTP ..... *Point to Point*

ROS2 ..... *Robot Operating System 2*

RRT ..... *Rapidly Exploring Random Tree*

SLAM ..... *Simultanious Lacalization and Mapping*

SRDF ..... *Semantic Robot Description Format*

TCP ..... *Tool Center Point*

UDP ..... *User Datagram Protocol*

UML ..... *Unified Macros Language*

URDF ..... *Unified Robot Description Format*

xml ..... *Extensible Markup Language*

z.B ..... *zum Beispiel*

Abbildungsverzeichnis

1. Abstract

Im Rahmen des Forschungsprojektes „move.mORe“ wird an der Hochschule Karlsruhe ein multimodales Logistiklabor aufgebaut. Die Versuchsanlage soll der Erforschung innovativer (Intra-)Logistikkonzepte dienen. Die digitale Lagerhaltung soll dabei durch eine autonome Bedienung ergänzt werden. Als ein Kernelement soll der Aufbau eines Hochregallagers stattfinden, das von einem Cobot (6DoF-Knickarmroboter) bedient werden soll. Dazu soll der Roboterarm auf einem Automated Guided Vehicle (AGV) montiert werden, um eine vollständig autonome Be- und Entladung des Hochregallagers zu ermöglichen.

Dieser Projektbericht dokumentiert die Realisierung eines Koordinationskonzeptes zur ganzheitlichen Steuerung des Verbundsystems aus Roboterarm und AGV. Zunächst wird der Aufbau der Simulationsumgebung erläutert. Diese spielt in diesem Projekt eine besonders wichtige Rolle, da die für die kollisionsfreie Bahnplanung des Roboters sowie für das SLAM-Verfahren des AGV notwendigen Tiefenkameras und Lidarsensoren in der Realität noch nicht implementiert sind. Daher müssen diese Sensordaten aus der Simulation gewonnen werden. In diesem Kapitel wird die grundlegende Softwarearchitektur erläutert und weiter auf die URDF-Modelle, die Implementierung des SLAM-Verfahrens sowie die Steuerung des Roboterarms und die Benutzerschnittstelle eingegangen. Es folgt die Erarbeitung und Umsetzung des Koordinationskonzepts sowie die Erläuterung der kollisionsfreien Bahnplanung mit Hilfe der Octomap. Im Kapitel zur Hardware wird die mechanische Montage des Roboters auf dem AGV, sowie die elektrische Verschaltung und der Aufbau des Schaltschrankes beschrieben

Zum Schluss wird erläutert, wie Roboterarm und AGV initial in Betrieb genommen wurden und mit einem Quick-Start Guide aufgezeigt, wie die in dieser Arbeit dargestellten Funktionen anzuwenden sind.

2. Aufgabenstellung

Aufgabe dieser Projektarbeit ist es, den Cobot (6DoF Igus ReBeL) auf dem AGV zu montieren und ein Koordinationskonzept zur Steuerung der beiden Systeme zu implementieren, sodass eine vorgegebene Lagerplatzposition mit dem TCP des Roboterarms erreicht werden kann.

Der Roboterarm steht als „Open-Source-Variante" ohne Steuerung zur Verfügung. Diese soll mit Hilfe eines Raspberry Pi implementiert werden. Parallel zu dieser Projektarbeit wird im Rahmen einer Abschlussarbeit an der autonomen Navigation des AGV mittels SLAM gearbeitet. Eine weitere Arbeit beschäftigt sich mit der Entwicklung eines Greifers sowie der Greifpositionserkennung, wobei eine zusätzliche Tiefenkamera am TCP des Manipulators angebracht wird. Da die Arbeiten parallel durchgeführt werden, sind die Kamerasysteme zur Navigation des AGV und zur Greifposenerkennung zum Zeitpunkt der Bearbeitung dieser Projektarbeit noch nicht implementiert. Da diese Systeme jedoch für die Bahnplanung des AGV und des Roboterarms benötigt werden, soll in dieser Projektarbeit weitestgehend in einer Simulation gearbeitet werden, in der die fehlenden Sensordaten nachgebildet werden.

3. Anforderungen

Abbildung 3-1 zeigt die Anforderungsliste. Da es sich bei dieser Projektarbeit um die Entwicklung eines ersten funktionsfähigen Konzepts handelt, sind spezifische Anforderungswerte und Daten schwer zu definieren und wurden daher weggelassen.

Für die Steuerung des ReBeL-Cobots ist bereits ein ROS2-Paket der Firma Igus online verfügbar. Dieses soll auf dem Raspberry Pi implementiert werden, um die Steuerung des Roboterarms zu ermöglichen. Für das AGV wurde in einer vorhergehenden Projektarbeit bereits eine ROS-Schnittstelle entwickelt, die es erlaubt, Geschwindigkeit und Bewegungsrichtung zu steuern. Bisher erfolgt die Kommunikation allerdings über ROS1. Ziel ist es, das Kommunikationsprotokoll für die Navigation auf ROS2 umzustellen. Hierzu soll es in das Nav2-Stack und die SLAM-Toolbox integriert werden. Als Test- und Validierungsumgebung soll die „Gazebo Physics Simulation“ funktionsfähig gemacht werden, wobei das gesamte Framework am Ende möglichst in einem Docker-Container vorliegen soll.

Zusätzlich sollen Cobot und AGV mechanisch und elektrisch gekoppelt werden, sodass ein Betrieb des Gesamtsystems möglich ist. Die Funktionsfähigkeit der Simulation soll durch Ergänzung der fehlenden Kameras am AGV und des TCP am Roboterarm auf die reale Welt übertragbar sein.

Lfd. Nr.	Art	Anforderungen
1		**Roboterarm IgusRebel**
1.1	F	Robotersteuerung auf Basis eines RaspberryPi
1.2	F	Implementierung der Steuerung mit ROS2
1.3	W	Dockerisierung der Software
2		**SEW AGV**
2.1	F	Implementierung für Bewegungsbefehle mit ROS2
2.2	F	Navigation auf Karte (so gut ohne sensorisches Feedback möglich)
3		**Kombination AGV und Roboterarm**
3.1		**Mechanische Montage des Roboterarms auf AGV**
3.1.1	F	Verschraubung auf Abdeckplatte
3.2		**Elektrischer Anschluss des Roboterarms und Robotersteuerung an AGV**
3.2.1	F	Anschluss an für Aufbauten vorgesehene 48V Klemme
3.2.2	F	DC/DC-Konverter von 48V auf 24V für Roboter
3.2.3	F	Steckverbindung zwischen AGV und Roboter
3.3		**Koordination Roboter und AGV**
3.3.1	F	Implementierung in ROS2
3.3.2	F	Funktionsfähigkeit in Simulation

Abbildung 3-1: Anforderungsliste

4. Zeitplan

Der Zeitplan (Abbildung 4-1) dient dazu, das Gesamtprojekt in kleinere Teilaufgaben zu unterteilen. Dabei werden Meilensteine festgelegt, an welchen bestimmte Aufgaben erledigt sein müssen. Ein Meilenstein besteht dann wiederum aus mehreren Arbeitspaketen, welche nacheinander abgearbeitet werden. Dadurch wird eine zeitliche Bindung an gewisse Aufgabenpakete erreicht. Der Zeitplan dient uns als Leitfaden und Übersichtstool. Die blauen Balken stellen die für die bestimmte Aktivität geplante Bearbeitungszeit dar. Im Laufe der Bearbeitungszeit wurde der Zeitplan den entstandenen Verzögerungen angepasst. Dabei sind Verkürzungen der Bearbeitungszeit heller dargestellt und Verlängerungen in dunkelblau angehängt. Eine Herausforderung war es, die Hardware bis Mitte Juli funktionsfähig zu machen, da zu diesem Zeitpunkt ein Gruppenmitglied für ein Auslandssemester Deutschland verlassen hat. Mit den dargestellten Anpassungen konnte das Projekt im Rahmen der gegebenen Zeit abgeschlossen werden.

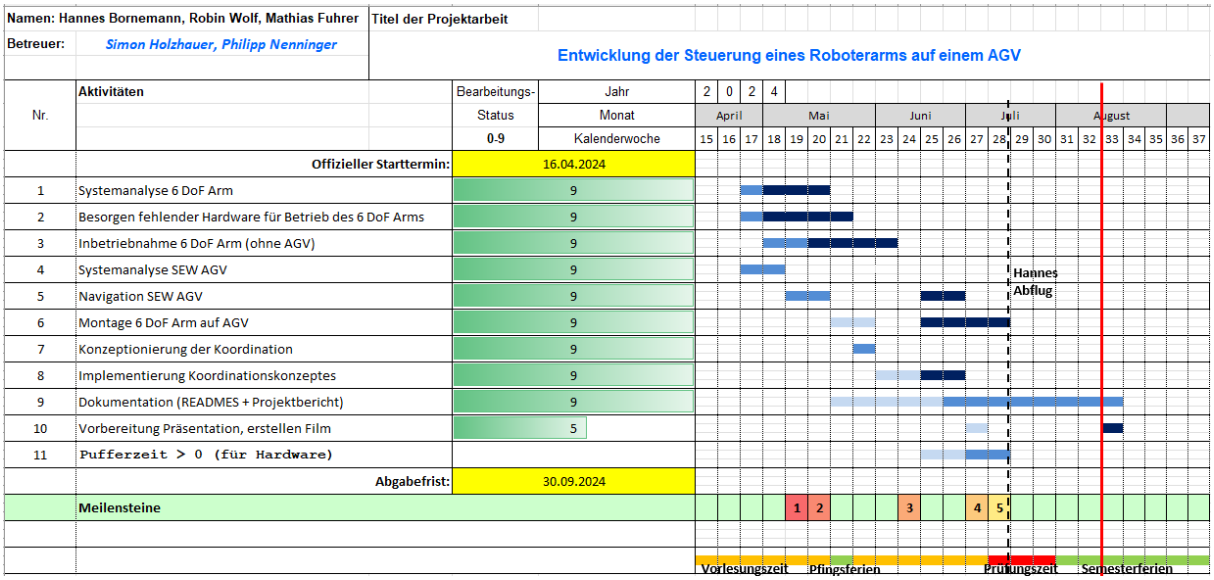

Abbildung 4-1: Zeitlplan

5. Simulationsumgebung

Das übergeordnete Ziel der Entwicklungsarbeit mit dem SEW AGV und dem Igus ReBeL Roboterarm ist es, in einer Logistikumgebung autonom spezifische Aufgaben auszuführen.

Als Entwicklungsziel wird im Folgenden definiert, dass der mobile Manipulator durch das Lager navigieren und eigenständig kleinere Kisten (KLT) manipulieren kann. Dabei können die Kisten in Regalen oder auf Tischen positioniert sein.

Dabei wird zur Systementwicklung eine Simulationsumgebung in Gazebo-Physics aufgesetzt, welche mit ROS2 Humble kompatibel ist.

Ein großer Vorteil von ROS2 ist, dass sämtliche Hardware durch geeignete Hardware-Interfaces abstrahiert wird. Dies bedeutet, dass die komplette Software, welche mit simulierter Hardware entwickelt wird, ohne Anpassungen auch für das reale Robotersystem funktioniert. Es müssen lediglich die Hardwareinterfaces getauscht werden.

So kann das komplette Robotersystem zuvor in einer Simulationsumgebung unabhängig von der Hardware entwickelt werden. [1]

Außerdem werden für die auszuführenden Tasks Funktionalitäten benötigt, die zum jetzigen Zeitpunkt noch nicht in Hardware umgesetzt sind und die Umsetzung davon auch nicht Teil dieser Projektarbeit ist. Speziell geht es darum Sensorik zur Perzeption wie ein 2D-Lidarscanner und eine Tiefenbildkamera simulativ abzubilden, um das definierte Entwicklungsziel erreichen zu können.

5.1 Softwarearchitektur und ROS2-Packages

Das komplette Robotersystem kombiniert das AGV des Herstellers SEW mit dem sechsachsigen Roboterarm des Herstellers Igus zu einem mobilen Manipulator. [2] [3]

Dazu müssen die beiden Einzelsysteme sowohl kinematisch als auch logisch verknüpft werden und die Navigation durch das Lager und die Bahnplanung für den Roboterarm eingerichtet werden. Es wird die ROS2 Distribution Humble mit Long Time Support verwendet. [4]

Jede ROS2-Applikation besteht aus unterschiedlichen Packages, die entweder installiert werden können oder selbst entwickelt werden können. Grundsätzlich benötigt dabei ein Robotersystem mindestens drei Hauptbestandteile:

- Definition der kinematischen Kette

- Hardwaretreiber
- Bahnplaner oder Navigation
- Anwendung, die vom User in möglichst stark abstrahiertem bzw. vereinfachtem Code geschrieben werden kann

Für das kombinierte Robotersystem müssen die kinematischen Ketten des AGV und des Roboters kombiniert werden.

Die Hardwaretreiber bleiben weiterhin getrennt, greifen jedoch auf die jeweils zugehörigen Gelenke/ Räder des Gesamtsystems zu.

Für das AGV muss zusätzlich die Navigation mittels Nav2-Stacks, für den Roboterarm die Bahnplanung mittels MoveIt aufgesetzt werden.

Außerdem werden diverse Client-Packages aufgesetzt, um komplexe Funktionalitäten der ROS2-Umgebung auf einfache Python Methoden zu abstrahieren. So soll das entwickelte System auch für User, die keine tiefere Kenntnis in ROS2 besitzen, zu bedienen sein.

Die nachfolgende Grafik zeigt die wichtigsten ROS2-Packages, die für die Umsetzung des mobilen Manipulators notwendig sind und wie diese miteinander agieren.

Grau hinterlegte Packages werden für den Betrieb des Igus Roboterarms, blau hinterlegte Packages für den Betrieb des AGV und weiß hinterlegte Packages zur User-Interaktion benötigt.

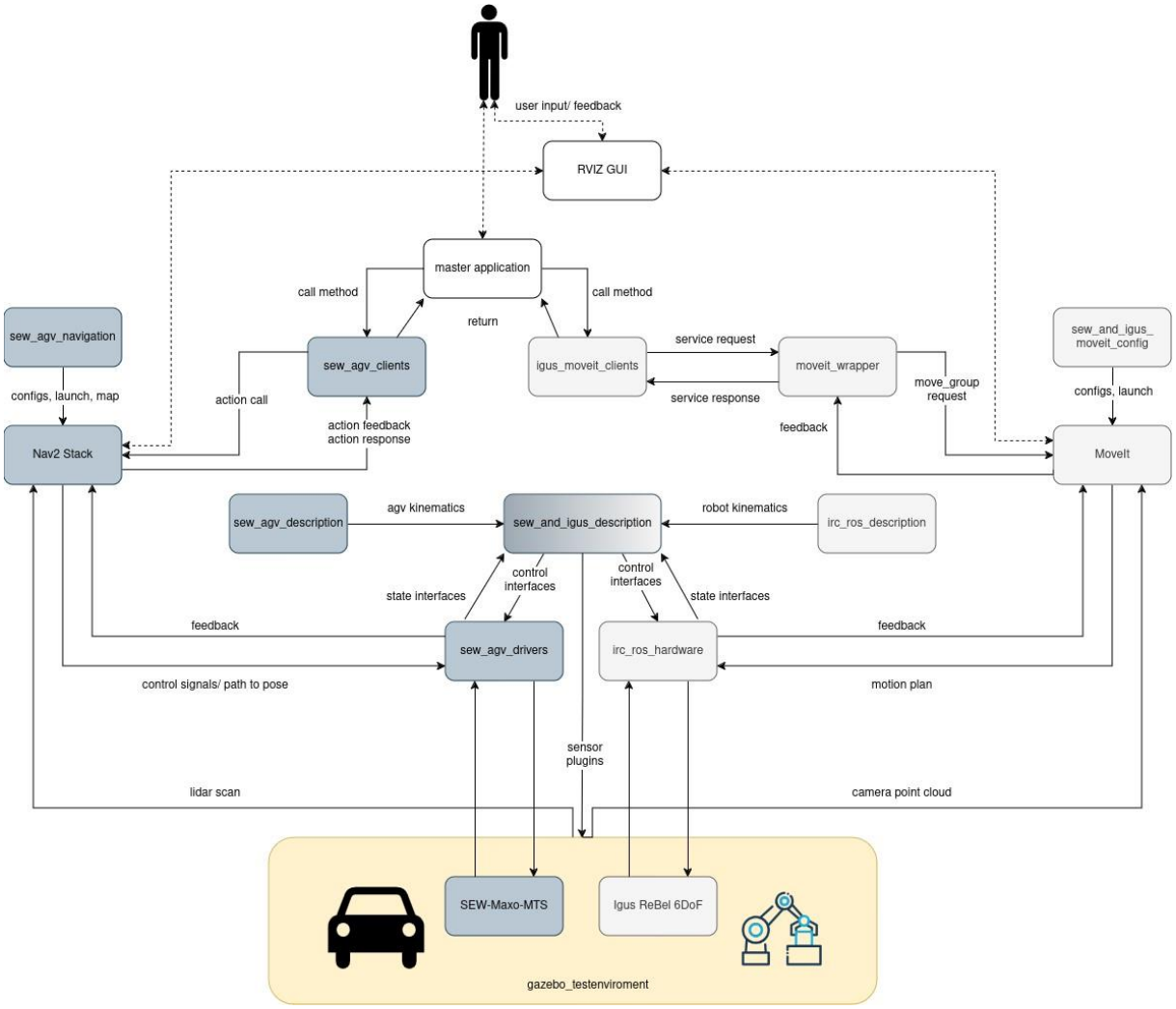

Abbildung 5-1 Systemübersicht simuliertes Gesamtsystem

In den folgenden Abschnitten werden die einzelnen Packages jeweils genauer erläutert und deren Bedeutung für das Gesamtsystem herausgestellt.

5.2 Design der Testumgebung

Gazebo und dessen Physics Engine stellt für das zu entwickelnde Robotersystem die Umgebung, in der es agiert, dar.

Wie bereits erwähnt, soll diese ein möglichst realistisches Logistikumfeld abbilden, in dem verschiedene Benchmark-Aufgaben mit dem System durchgeführt werden können.

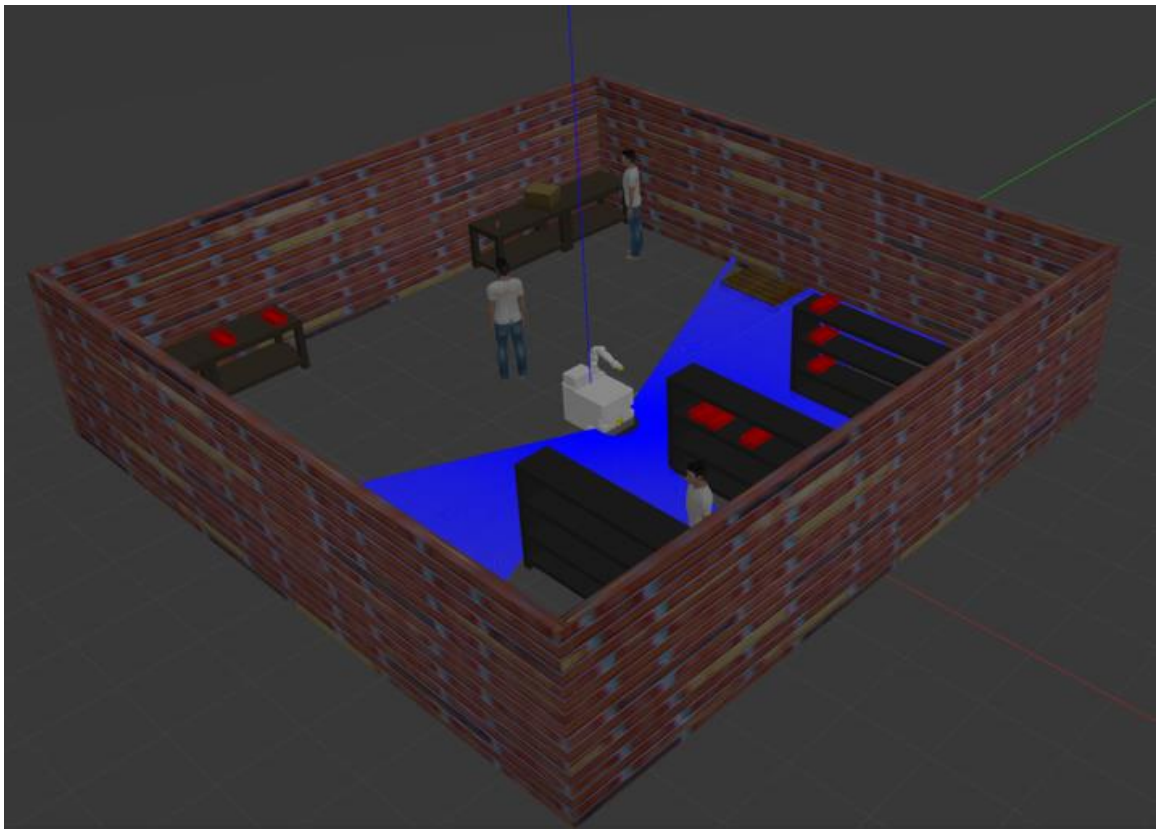

Abbildung 5-2 Benchmark Umgebung in Gazebo Simulation mit Robotermodell

Die gewählte Benchmark Umgebung besteht aus einem ca. 10x10m großem Raum mit zwei Tischen und drei Regalen, die in Korridoren angeordnet sind. Die Korridore weisen eine Breite von ca. 2 m auf und sind damit vergleichsweise eng. Dies ist allerdings bewusst gewählt, um möglichst anspruchsvolle Benchmarks durchführen zu können.

Dazu sind diverse typische Hindernisse, wie Menschen, Paletten und Kleinladungsträger (KLT) integriert, die spezielle Roboteraufgaben wie die Navigation oder Kollisionsvermeidung herausfordern sollen.

Dabei wurde initial ein Modell der Welt erstellt, indem diverse Gegenstände aus der Gazebo-Objektbibliothek (Mensch, KLT, Palette, Wände) und eigens konstruierte (Tisch, Regal) in der

gewünschten Anordnung positioniert wurden. Diese Welt wird von nun an bei jedem Start der Simulation geladen.

Das Robotermodell (in Abbildung 5-2 in der Mitte zu sehen) wird direkt aus der Definition der kinematischen Kette abgeleitet und mittels Spawn-Entity Node in Gazebo-Datenformate überführt.

Die kinematische Kette ihrerseits ist in einem URDF-Modell (unified robot description format) angelegt und stellt die Grundlage für ein Robotersytem in ROS2 dar.

In Gelb dargestellt ist die simulierte Sensorik des Robotermodells. Zum einen befindet sich ein 2D-Lidar Scanner in der Front des AGV. Dessen Aufgabe ist es, ein möglichst genaues Bild der Umgebung aufzunehmen, mit dem der mobile Manipulator in der Umgebung lokalisiert werden kann. Zum anderen befindet sich ein Tiefenbildkamera am TCP des Roboterarms, welche aktuell eine Punktewolke aufnimmt, die zur lokalen Kollisionsvermeidung während der Bahnplanung verwendet wird. Denkbar ist hier auch eine kamerabasierte Zielposen-Lokalisierung.

Mit der aufsetzten Simulationsumgebung kann nun also ein reales Robotersystem für Entwicklungszwecke nachgebildet werden. Erstens können Sensorsignale (inklusive Messrauschen!) generiert werden, zweitens kann ich der Roboter gemäß physikalischen Gesetzen durch seine Antriebe in der Umgebung bewegen.

5.3 Kinematische Kette und URDF-Modelle

Die Grundlage einer jeden ROS2 Applikation stellt das URDF-Modell des Robotersystems dar.

Das URDF-Modell spezifiziert die kinematische Kette (Gelenke und Körper) eines Robotermodells sowie alle kinematischen und geometrischen Beziehungen und Randbedingungen. Außerdem sind dabei jedem Körper jeweils ein CAD-Modell für die zu visualisierende Geometrie und ein vereinfachtes CAD-Modell, welches aus Rechenzeitgründen bei der Kollisionsprüfung verwendet wird, zugewiesen.

Für das SEW-AGV existierten vor dem Projekt keine CAD-Daten, daher wurde eine Konstruktion, die die äußeren Abmaße repräsentiert, angefertigt.

Das URDF-Modell wird in xml verfasst und kann durch den Einsatz des „xacro-Packages“ in weitere Unterbaugruppen strukturiert werden. [5]

Das folgende Diagramm in Abbildung 5-3 zeigt einen Überblick über die einzelnen Komponenten der kinematischen Kette des gesamten Robotersystems. Außerdem ist der Inhalt der jeweiligen Macros stichpunktartig erläutert.

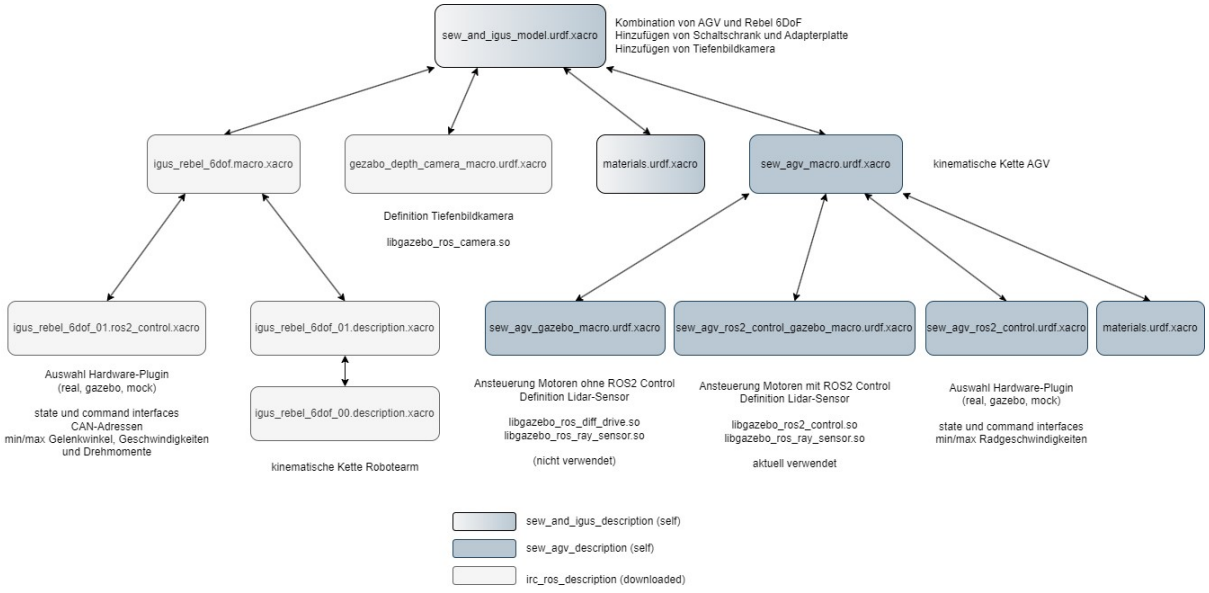

Abbildung 5-3 Struktur URDF-Modell des gesamten Robotersystems

Es wird deutlich, dass der Roboterarm und das AGV zuerst separat betrachtet und anschließend zusammengeführt wurden. Dabei wurde das Package „irc_ros_description“ von dem vom Hersteller zur Verfügung gestellten Repo übernommen. Das Package „sew_agv_description“ welches die kinematische Kette des AGV beschreibt und das

kombinierte Package „sew_and_igus_description“ wurde dem gegenüber selbst implementiert.

Zusätzlich muss jedem Gelenk ein entsprechendes Interface für den Hardwaretreiber und dessen Plugin zugewiesen werden. Dazu wurden Macros implementiert, die diese Definitionen für das ROS2-Control Framework bereitstellen.

Da wie bereits erläutert auch Sensoren mit Gazebo simuliert werden sollen, müssen sogenannte Sensorplugins dem zugehörigen Körper aus der kinematischen Kette zugewiesen werden. So wird das Sensorkoordinatensystem und diverse Eigenschaften des Sensors wie Auflösung, Messrauschen, Reichweite, etc. festgelegt.

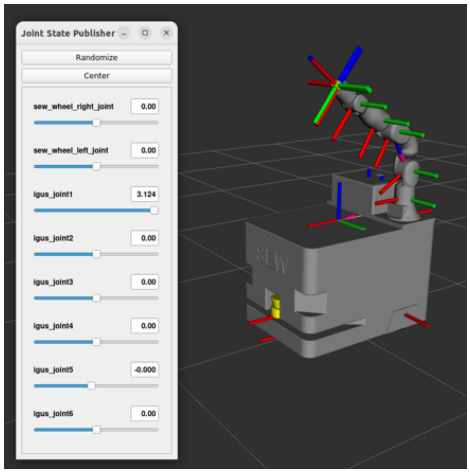

Abbildung 5-4 kombiniertes URDF-Modell aus AGV und Igus ReBeL 6 DoF

5.4 Navigation des SEW-AGV über Nav2-Stack

Der Nav2 Stack [6] ist ein in ROS2 integriertes Framework für mobile Robotik. Dieser ermöglicht es mobilen Robotern, sich durch komplexe Umgebungen zu navigieren, um benutzerdefinierte Aufgaben zu erledigen, unabhängig von der Klasse der Roboterkinematik. Nav2 kann ein Umweltmodell aus Sensor- und semantischen Daten erstellen, dynamisch Wege planen, Geschwindigkeiten für Motoren berechnen und Hindernisse vermeiden.

Die Nav2 interne Steuerung der einzelnen unabhängigen Server wird durch einen Behavior-Tree gesteuert. Der Vorteil eines Behavior Tree Frameworks ist, dass dieser dynamisch auf Veränderungen in der Anforderung eingehen und entsprechend reagieren kann.

Die folgende Grafik stellt die wesentlichen Bestandteile des Nav2 Frameworks, also dessen Server und wie diese miteinander agieren, dar.

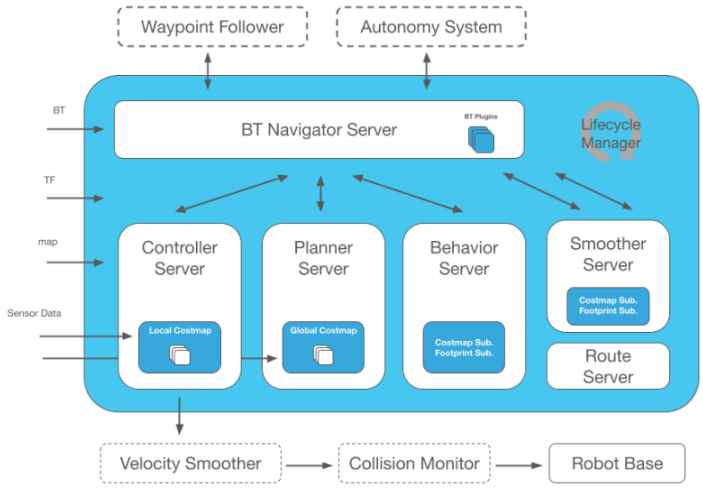

Abbildung 5-5 Systemübersicht Nav2 Stack [6]

Für dieses Entwicklungsprojekt wird Nav2 verwendet, um alle Tasks, die mir der Navigation des mobilen Manipulators in Verbindung gebracht werden können, zu bewältigen.

Darunter zählt:

- Aufnehmen einer Karte der Umgebung mittels SLAM-Algorithmus (Simultaniuos Localization and Mapping)
- Lokalisierung über 2D Lidarscan in der Karte mittels AMCL-Algorithmus (Adaptive Monte-Carlo Localization)
- Planen von kollisionsfreien Pfaden mittels A*-Algorithmus und Global- sowie Local Costmap

Ist das System einmalig aufgesetzt und die entsprechenden Nodes gelauncht, können sämtliche Service- und Actionserver des Nav2 Stacks wie z.B. „navigate_to_pose“ über entsprechende Clients angesprochen werden.

Die Konfiguration des Nav2 Stacks und dessen Algorithmen sind in dem Package „sew_agv_navigation“ zu finden. Die entsprechenden Clients zur Kommunikation mit Nav2 sind im Package „sew_agv_clients“ definiert.

Dieser Node stellt eine eigene Python Klasse zur Verfügung, die in einem Master-Control Skript instanziiert werden kann und damit ein stark abstrahiertes User- Interface bereitstellt.

5.4.1 Mapping

Grundlage für eine funktionierende Navigation ist eine Repräsentation der Umwelt, in der sich das Robotersystem lokalisieren und orientieren kann.

Da in dem hier entwickelten System ein 2D-Lidarscanner als Sensor verwendet wird, ist die Umwelt als 2D-Karte repräsentiert. Diese muss einmalig initial aufgenommen werden, indem das Robotersystem im Mapping Modus gestartet wird und manuell, über einen Xbox Controller, in der Umgebung bewegt wird.

Wichtig dabei ist, dass alle Hindernisse, die sich nicht auf Höhe des Lidarscanners befinden, nicht in der Karte berücksichtigt werden. In der Benchmark Umgebung wurden daher Regale und Tische so konstruiert, dass sie vom Lidarscanner erfasst werden können.

Beim Mapping selbst wird ein SLAM-Algorithmus (Simultaniuos Localization and Mapping) verwendet, um aus den Daten des Lidarscanners die Repräsentation der Umgebung zusammenzusetzen. [7]

Abschließend muss die Karte über entsprechende Terminal-Befehle gespeichert werden.

Abbildung 5-6 zeigt die von der Benchmark Umgebung aufgenommene Karte.

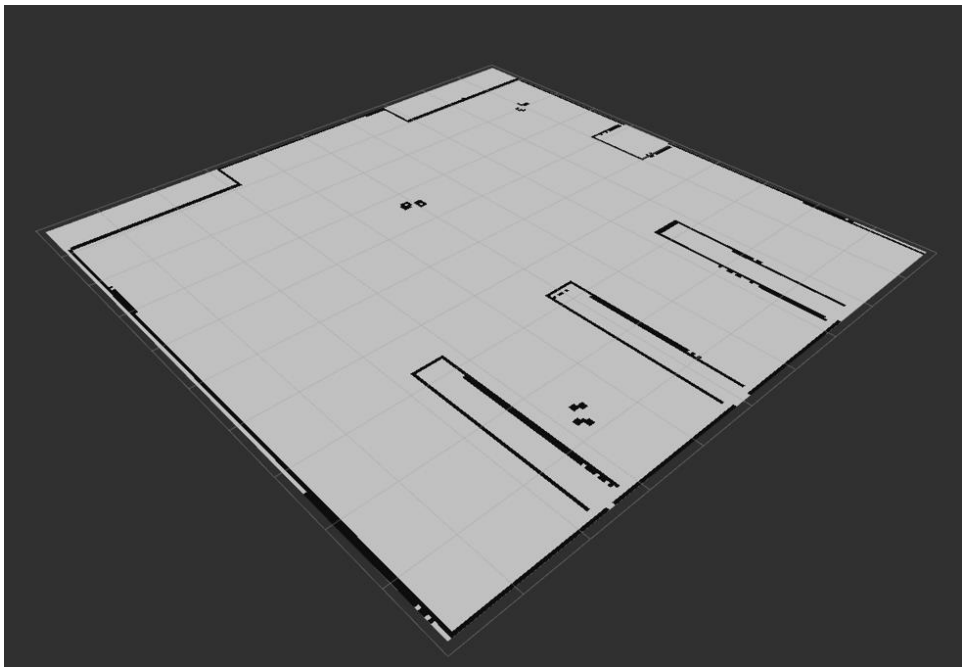

Abbildung 5-6 aufgenommene Repräsentation der Benchmark Umgebung (2D-Karte)

5.4.2 Navigation

Die Navigation umfasst alle Nodes, die zur Planung und Ausführung einer Bewegung des AGVs benötigt werden. Diese werden, wie bereits erwähnt, durch einen Behavior Tree gesteuert.

Als Grundlage benötigt die Navigation eine Repräsentation der Umwelt, im Falle des hier entwickelten Systems die Karte.

Auf Basis der Karte und der aktuellen Sensordaten kann das Robotersystem sich mittels einer adaptiven Monte-Carlo Lokalisation innerhalb der Karte lokalisieren. Dabei wird die Transformation zwischen dem Odometrie-Koordinatensystem und der Karte berechnet und so die Transformationskette zwischen Kartenursprung und Roboter geschlossen. Dies ist die Grundlage für die Berechnung von Bewegungsbahnen/ Trajektorien zu einer gegebenen Zielpose.

Folgende Grafik zeigt den Zusammenhang zwischen der internen Odometrie, welche aus Raddrehzahlen bestimmt wird und der Lokalisierung über den Lidar-Scanner.

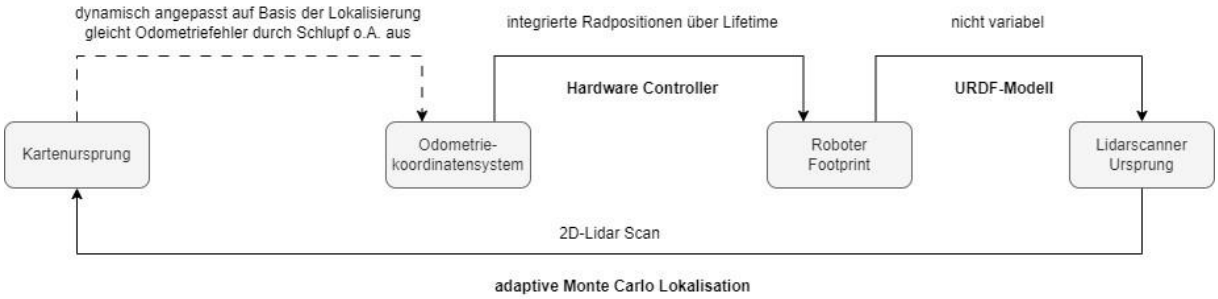

Abbildung 5-7 Prinzip der Lokalisierung mittels Odometrie und Lidarscanner

Als Planungsalgorithmus kommt ein kostenbasierter A*-Algorithmus zum Einsatz. Um den Algorithmus um Hindernisse herumzuleiten, werden parallel sogenannte Local und Global Costmaps erzeugt. Die Global Costmap basiert auf den Hindernissen, die aus der Karte hervorgehen. Desto näher der Pfad an Hindernisse kommt, desto höhere Kosten hat es zur Folge. Für lokale, veränderliche Hindernisse, die nicht in der initialen Karte vorhanden sind, kann das Robotersystem ebenfalls reagieren. Durch die Local-Costmap werden zusätzliche Kosten für Punkte, die vom Lidar Scanner erkannt werden, hinzugefügt.

Als Kollisionsgeometrie des Roboters muss ein sogenannter Footprint definiert werden, der der Maximalfläche des Robotermodells projiziert auf den Boden entspricht. Wichtig hierbei zu erwähnen ist, dass der Footprint nicht variabel ist. Dies bedeutet, dass sich der Roboterarm des mobilen Manipulators bei einer Bewegung des AGV immer innerhalb des Footprint befindet. Sonst kann eine Kollision des Arms mit der Umgebung während der Navigation nicht ausgeschlossen werden.

Die folgende Grafik zeigt Lidarscans und Costmaps innerhalb der Karte. Der mobile Manipulator befindet sich hier in seiner Home Position. Der blassere Farbverlauf stellt die Global Costmap dar, der kräftigere die Local Costmap. Die Sensordaten des Lidarscanners sind als rote Punkte dargestellt.

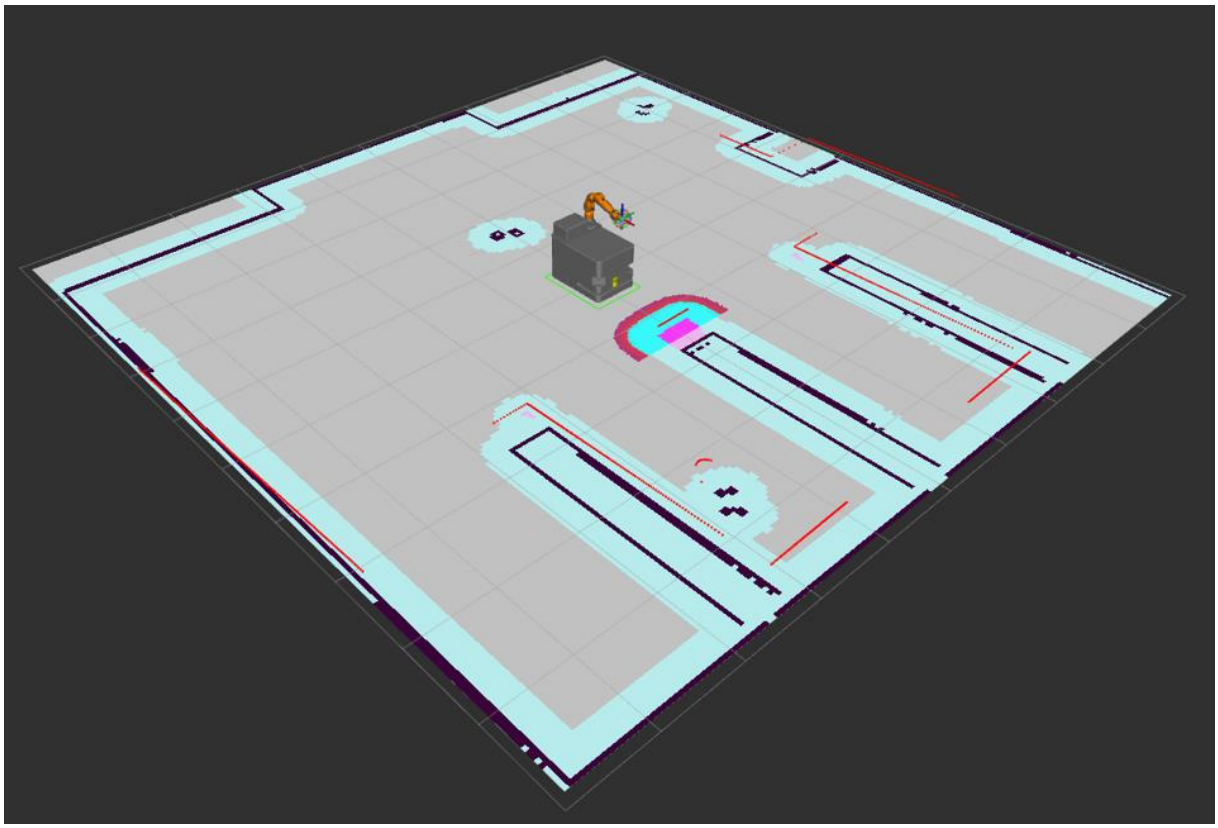

Abbildung 5-8 mobiler Manipulator in Benchmarkumgebung mit aktiver Navigation

Damit kann das Robotersystem sich nun autonom in der Umgebung bewegen, auf dynamische Hindernisse reagieren und sogar auf diverse Fallback-Strategien zurückgreifen, sollte ein Fehler bei der Navigation auftreten.

5.5 Ansteuerung des Roboterarms und Kollisionsvermeidung

Die Trajektoriengenerierung für den Roboterarm wird über das MoveIt Framework ebenfalls in ROS2 umgesetzt.

Hierzu dient vor Allem das implementierte Package „sew_and_igus_moveit_config“.

Das Ziel der Implementierung ist, den Roboter ohne Kollisionen mit sich selbst oder der Umwelt zu einer gegebenen Zielpose zu bewegen. Diese Zielpose kann entweder im Joint Space (Gelenkwinkel) oder im World Space (kartesische Koordinaten) angegeben werden. Die Bewegung soll entweder als PTP-Befehl, welcher aktiv Kollisionen vermeidet und schnellstmöglich zur Zielpose führt oder als LIN-Befehl ausgeführt werden. Der LIN-Befehl ist zwar nicht in der Lage aktiv Kollisionen zu umfahren, allerdings ermöglicht er genaue und nachvollziehbare lineare Pfade im Weltkoordinatensystem.

Das MoveIt Framework bietet eine Vielzahl von Methoden, mit welchen solche Funktionalitäten ermöglicht werden können, doch dafür muss es zunächst eingerichtet werden.

Die Basis für die Trajektorienplanung stellt einmal mehr das zuvor definierte URDF-Modell des gesamten mobilen Manipulators dar. Dieses definiert die kinematische Kette inklusive der Kollisionsgeometrien der einzelnen Körper. Diese sind für die Bahnplanung besonders wichtig.

5.5.1 Semantische Beschreibung der kinematischen Kette im SRDF-Modell

Im URDF-Modell sind lediglich geometrische Beziehungen und Gelenkvariablen innerhalb der Kinematischen Kette definiert. Damit die kinematische Kette nun auch zur Trajektorienplanung verwendet werden kann, müssen zusätzlich semantische Zusammenhänge definiert werden:

- Welche Körper sind benachbart und dürfen daher an Kontaktflächen miteinander kollidieren?
- Welche Körper können auf Grund der kinematischen Struktur nie kollidieren und können daher bei der Kollisionsprüfung vernachlässigt werden?
- Welcher Körper stellt die Basis der kinematischen Kette dar, welcher das Ende bzw. den Tool Center Point (TCP)?

Da das gegebene Robotersystem durch eine serielle kinematische Kette repräsentiert wird, können Kollisionen an allen Kontaktflächen deaktiviert werden.

Wichtig für die Kombination der einzelnen Robotersysteme AGV und Roboterarm ist an dieser Stelle, dass die Basis für die Trajektorienplanung in das Koordinatensystem der Basis des Roboterarms gelegt wird. So wird nur die serielle Kette des Arms bei der Trajektorienplanung

berücksichtigt. Es werden nun simultane Trajektorien für die sechs Gelenke erzeugt, welche an einen entsprechenden Hardwarecontroller (Joint Trajectory Controller) weitergeleitet werden können, um diese auszuführen. Die restlichen Geometrien des AGV und des Aufbaus werden nur zur Kollisionsprüfung hinzugezogen. [8]

5.5.2 Bahnplanungsalgorithmen

Bei der Trajektoriengenerierung muss das MoveIt Framework auf einen Bahnplanungsalgorithmus zurückgreifen, welcher im ersten Schritt kollisionsfreie Roboterbahnen berechnet bzw. sucht.

Dafür unterstützt MoveIt diverse Libraries. In diesem Projekt wurde die Bahnplanungslibrary OPML (Open Motion Planning Library) verwendet.

In OMPL sind eine Reihe an probabilistischen Bahnplanern wie z.B. PRM (Probabilistic Roadmaps) und RRT (Rapidly Exploring Random Tree), welche wiederum in den unterschiedlichsten Distributionen angeboten werden, vorhanden. Sogar optimierende Planer wie RRT-Star sind integriert.

Diese können in einer eigenen Config-Datei parametriert werden. Als Standardbahnplaner wird im Zuge dieses Projektes der RRTkConfigDefault Planer eingesetzt. Dieser liefert optimale Roboterbahnen und weist eine äußerst schnelle Berechnung auf. [9]

Die Bahnplanung findet für gewöhnlich ausschließlich im Joint bzw. Configuration Space des Roboters statt, der in diesem Falle sechs Dimensionen besitzt (sechs Gelenke). Die Kollisionsprüfung der einzelnen berechneten Wegpunkte der Bewegungsbahn muss allerdings im World Space vollzogen werden, da es auf Grund der Berechnungskomplexität keinen Sinn machen würde, die Hindernisse in den Configuration Space zu transformieren.

Um die Umrechnung zwischen Configuration Space und World Space zu ermöglichen wird einerseits die Vorwärtskinematik, welche über das URDF-Modell definiert ist, andererseits auch die weitaus komplexere Inverskinematik benötigt.

5.5.3 Inverskinematik-Solver

Die Berechnung der Inverskinematik ist ein komplexes Problem der Robotik, da diese je nach aktueller Roboterpose entweder eine Lösung, mehrere Lösungen oder keine Lösung hat.

Aus dem Grund wird die Inverskinematik (IK) innerhalb einer ROS2 Applikation für gewöhnlich mit numerischen, konvergierenden Optimierungsalgorithmen gelöst, welche als vorgefertigtes Plugin in MoveIt eingebunden werden können.

Als Standard IK-Slover kommt meist der sogenannte KDL-Solver von Orocos (Kinematics and Dynamics Libary) zum Einsatz, welcher die Lösung der IK über eine Integration der invertierten Jacobimatrix berechnet. Das numerische Verfahren basiert auf dem Newton Verfahren in der Optimierung und löst nichtlineare Gleichungssysteme (der Matrizengleichungen) mit einem Approximationsalgorithmus. [10]

In vergangenen Projekten wurden allerdings schlechte Erfahrungen mit diesem IK-Solver gemacht, da dieser vor Allem in der Nähe der Bewegungsgrenzen keine optimale Lösung für die IK findet. Dies führt im einfachsten Falle zu willkürlichen, nicht nachvollziehbaren Roboterbahnen bei PTP-Bewegungen, im schlimmsten Falle funktioniert die Kollisionsvermeidung nicht mehr zuverlässig und es kommt zu Crashs.

Daher wurde innerhalb dieses Projektes der Trac-IK Solver [11] von TRAC-Labs verwendet, welcher eine deutlich höhere Zuverlässigkeit aufweist.

Trac-IK ist eine Weiterentwicklung von KDL und führt neben einem angepassten Newton-Verfahren, welches lokale Minima in der Lösung der IK findet, auch ein nichtlineareren Optimierungsansatz (Sequential Quadratic Programming) durch.

In der Konfigurationsdatei kann aus mehrere Optimierungsobjektiven (Distance, Manipulability, Speed) gewählt werden, welche aus den vielen möglichen IK-Lösungen die jeweils am besten geeignete bestimmt. [12]

Innerhalb dieses Projektes erwies sich das Optimierungsobjektiv „Distance“ als am besten geeignet.

5.5.4 3D Perzeption zur Lokalen Kollisionsvermeidung

Nun ist das Robotersystem (speziell der Arm des mobilen Manipulators) in der Lage kollisionsfreie Bahnen zu planen, diese in Trajektorien für die einzelnen Gelenke zu überführen und schlussendlich auszuführen.

Zum jetzigen Stand werden ausschließlich Kollisionen mit dem Robotermodell geprüft und dementsprechende Trajektorien ausgeführt. Das gesamte Robotersystem kann sich allerdings durch die mobile Plattform durch die Umgebung bewegen und ist damit immer veränderten Umgebungsbedingungen/ Kollisionsgeometrien ausgesetzt.

Da das Mapping in einem 2D-Ansatz implementiert wurde, hat das System keine Information über die 3D Geometrien der Umwelt im aktuellen Standpunkt. Diese müssen für eine sichere, kollisionsfreie Roboterbewegung zuerst rekonstruiert werden.

Dafür wurde eine weitere Erweiterung zum MoveIt Framework implementiert. Mittels der sogenannten Octomap können dreidimensionale Strukturen in der Umgebung des Roboters durch eine aufgenommene Punktewolke rekonstruiert werden und als zusätzliche Kollisionsgeometrie übernommen werden.

Genau zu diesem Zweck wurde eine Tiefenbildkamera am Roboter-TCP montiert.

Octomap basiert auf einem 3D Occupancy Grid Ansatz, das bedeutet die Umgebung wird durch ein Grid von Würfeln approximiert, die entweder frei oder durch Geometrien belegt (occupied) sind. Darüber hinaus kann die so erzeugte Octomap ständig erweitert oder upgedatet werden, das heißt, sie entwickelt sich zunehmend mit der Roboterbewegung.

Die so aufgenommene Octomap wird relativ zur Basis des Roboterarms definiert und ist somit nicht mehr gültig, sobald sich die mobile Plattform in der Umgebung bewegt. Ist dies der Fall, muss die vorhandene Octomap zurückgesetzt und neu aufgenommen werden. [13]

Abbildung 5-9 zeigt eine aufgenommene Octomap von einer Regalpartie im Vergleich zu dessen Originaler Umgebung in der Simulation.

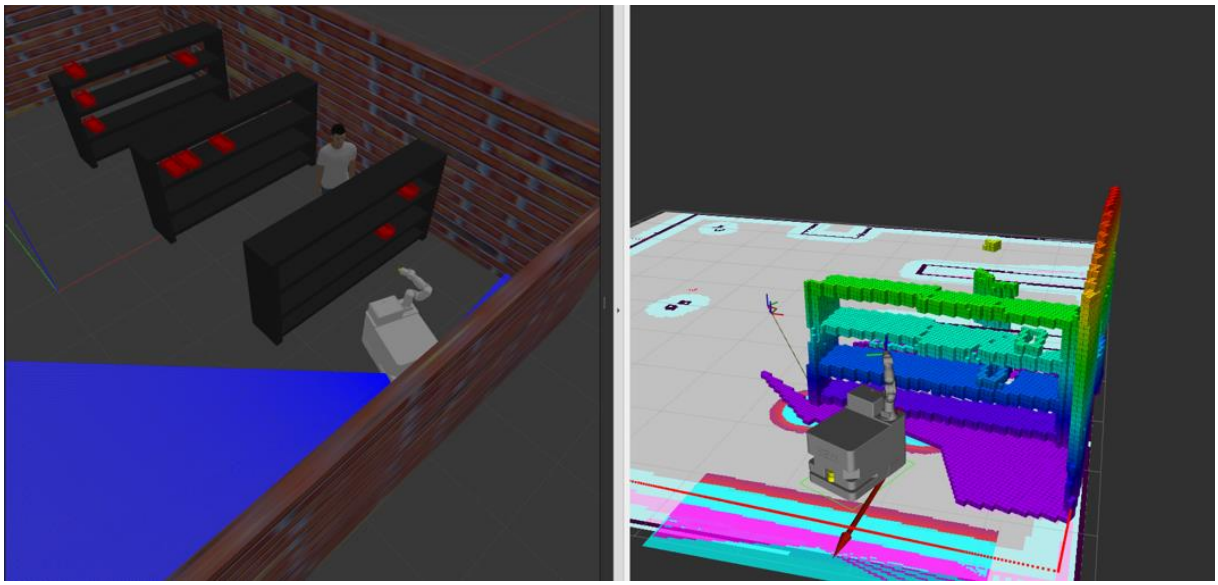

Abbildung 5-9 lokale dreidimensionale Nachbildung der Simulationsumgebung durch Octomap

5.5.5 Zugriff auf Funktionalitäten von MoveIt mittels Wrapper Package

Nun sind alle Funktionalitäten, die das zu entwickelnde Robotersystem aufweisen muss, um in der Benchmarkumgebung autonom Logistikaufgaben auszuführen, implementiert.

Diese können über MoveIt, das zentrale Framework zur Steuerung des Roboterarms nun abgerufen werden. Hierzu wird das Move-Group Interface in C++ verwendet, da es als einzige MoveIt API den vollen, benötigten Funktionsumfang bietet.

Leider ist die Nutzung dieser API nicht trivial und in C++ zudem noch nicht wirklich userfreundlich. Für Nutzer, die sich nicht ausgiebig mit Robotik und dem MoveIt Framework auseinandergesetzt haben, ist eine Nutzung dieser API faktisch nicht möglich. [14]

Daher wurde innerhalb dieses Projektes ein Wrapper-Package (moveit_wrapper) in C++ implementiert, welches Service-Server zur Verfügung stellt, die wiederum häufig verwendete Motion-Tasks der MoveIt API durchführen. So kann der User, über die im folgenden Kapitel erläuterten Clients, ganz einfach aus einer Python Datei die Server über einen ROS2-Service aufrufen, ohne sich explizit mir der konkreten Umsetzung der Motion-Applikation auseinanderzusetzen.

5.6 User Interface

Dem User soll wie bereits erläutert eine möglichst abstrahierte Bibliothek von Methoden zur Verfügung gestellt werden, mit denen er die genannten Robotersysteme ansteuern kann. Außerdem sollen die Methoden so implementiert werden, dass der User seine eigene Logistik-Applikation, unabhängig von dem innerhalb dieses Projektes weiter verfolgten Ansatzes, selbstständig programmieren kann.

Dazu wurden insgesamt zwei Packages aufgesetzt, die jeweils eine Klasse beinhalten, die dem User spezifische Methoden für den jeweiligen Systemteil zur Verfügung stellt.

1) „sew_agv_clients“ beinhaltet Methoden, die jeweils als Client für den Aufruf von ROS2 Services und Actions des Nav2 Stacks fungieren
2) „igus_moveit_clients“ beinhaltet erstens Methoden, die die angesprochenen Server aus dem „moveit_wrapper“ Package aufrufen, um den Roboterarm zu bewegen.

Die beiden Klassen sind jeweils innerhalb eines separaten ROS2 Nodes organisiert, der bei der Instanzierung der Klasse gelauncht und so lange aktiv ist, bis dieser aktiv zerstört wird.

Dem ganzen übergeordnet wird das Applikationspackage „master_application“. Hier werden die abstrahierten, übergeordneten „High-Level“ Programme vom User verfasst. Diverse Beispiel und Demoprogramme sind bereits zur Orientierung implementiert.

5.6.1 SEW AGV Methoden

Dieses Package enthält alle Methoden, die der User in Verbindung mit der Navigation des AGVs in der Benchmarkumgebung nutzen kann. Es handelt sich bei den Methoden der Klasse um Aufrufe von Service und Action Servern aus dem Nav2 Stack. Entsprechende Klassenvariablen und Methoden sind nachfolgend dargestellt.

AGV-Control:

```
# class variables
self.home_position = [[0.0,0.0,0.0], [0.0,0.0,0.0,1.0]]     # [position, quaternion]

# class methods
def check_nav_goal(self, frameID, pose):
    """
    string frameID: frame where the pose is given in e.g. 'map'
    list pose [x,y,w]: position and quarternion angle (rad) of the goal

    Returns
    -------
    bool goal_ok

    """
def move_to_nav_goal(self, frameID, pose):
    """
    string frameID: frame where the pose is given in e.g. 'map'
    list pose [x,y,w]: position and quarternion angle (rad) of the goal

    Returns
    -------
    int status (code which encodes status with navigation has terminated)

    """
```

Abbildung 5-10 Abstrahierte User-Methoden zur Steuerung des AGV

5.6.2 Igus Roboterarm Methoden

Dieses Package enthält alle Methoden, die der User in Verbindung mit der Bewegung des Igus Roboterarms in der Benchmarkumgebung nutzen kann. Es handelt sich bei den Methoden der Klasse erstens um Aufrufe von Service Servern aus dem MoveIt Wrapper Package, welches wiederum eine abstrahierte Schnittstelle zu den Funktionalitäten des MoveIt Bahnplanungsframeworks bereitstellt. Zweitens kann durch einen entsprechenden Service Aufruf ein Löschen der aktuellen Octomap veranlasst werden.

Entsprechende Klassenvariablen und Methoden sowie die Datentypen, die denen übergeben und zurückgegeben werden, sind nachfolgend dargestellt.

Robot-Arm-Control:

```
# class variables
self.home_position = [0.0,0.0,0.0,0.0,0.0,0.0]  # [joint1, joint2, joint3, joint4, joint5, joint6]

# class methods
def get_transform(self, from_frame_rel, to_frame_rel, affine=True):
    """
    string from_frame_rel: name of the source frame frame to transform from (child)
    string to_frame_rel: name of the target frame to transform into (parent)

    Returns:
    --------
    Affine: transformation matrix from robot base to target position and orientation (4x4 numpy array, can directly passed in m
    or
    geometry_msgs/TransformStamped: transformation between the two frames if affine=False

    """
def reset_planning_group(self, planning_group):
    """
    string planning_group: name of the planning group of the arm (default: igus_6dof)

    Returns
    -------
    bool success

    """
def setVelocity(self, fraction):
    """
    float: velocity caling coeffinet relative to joint speed limits (0.01 ... 0.5 recommendet)

    Returns
    -------
    bool success

    """
```

Abbildung 5-11 Abstrahierte User-Methoden zur Steuerung des Roboterarms Teil1

```
def home(self):
    """

    Returns
    -------
    bool success

    """
def ptp(self, pose: Affine):
    """
    cartesian goal pose: affine transformation martix format (can be converted from x,y,z, r,p,y or x,y,z and quaternion)

    Returns
    -------
    bool success

    """
def ptp_joint(self, joint_positions: List[float]):
    """
    joint space goal pose: list of goal joint states (rad)

    Returns
    -------
    bool success

    """
def lin(self, pose: Affine):
    """
    cartesian goal pose: affine transformation matrix format (can be converted from x,y,z, r,p,y or x,y,z and quaternion)

    Returns
    -------
    bool success

    """
def clear_octomap(self):
    """
    Returns
    -------
    None
    """
```

Abbildung 5-12 Abstrahierte User-Methoden zur Steuerung des Roboterarms Teil2

5.6.3 Master-Applikation

Das Package „master_applikation“ stellt das zentrale Interface der gesamten entwickelten ROS2 Applikation dar. Innerhalb dieses Packages kann der User seine eigenen Programme aus den vorgefertigten, abstrahierten Methoden erstellen und diese durch alle gängigen Python-Libaries wie z.B. OpenCV oder SK-Learn erweitern.

Dazu muss lediglich eine neue Python-Datei im Ordner *„src/master_application/master_application“* erzeugt werden und diese zur setup.py Datei hinzugefügt werden, um sie anschließend aus der Terminal-Befehlszeile aufzurufen.

Darüber hinaus wurde eine Klasse von Methoden implementiert, die für die Bearbeitung von Logistikaufgaben hilfreich sein könnten. Einige dieser Methoden wurden in dem innerhalb dieses Projektes gewählten Koordinationskonzeptes verwendet. Diese Klasse wird bei Initialisierung aus dem übergeordneten Programmskript als ein Node gelauncht und steht so lange zur Verfügung, bis dieser aktiv zerstört wird.

Die Methoden beziehen sich vor Allem auf das Handling der Koordination zwischen AGV und Roboterarm wie z.B. das Berechnen einer geeigneten Parkposition, um mit dem Roboterarm eine Kiste aus dem Regal zu greifen. Auch diverse Koordinatentransformationen, Datentypkonvertierungen und Methoden zur Visualisierung sind hier implementiert.

Die folgenden Methoden sind dabei in der Klasse „StorageClient“ verfügbar:

Storage-Handling:

```
# class variables
self.armRange = 0.66    # maximum range of the robots workspace (radius)
self.agvOffsetX = 0.4     # offset between the first detected position not in collision with the enviroment and the park positi
self.parkClearanceX = 0.15      # offset between the first detected position not in collision with the enviroment and the park
self.agvOffsetY = -0.19     # offset between the first detected position not in collision with the enviroment and the park posi
self.stepSize = 0.1     # step size in X-direction of the search algorithm searching positions not in collision with the enviro
self.joint1Heigth = 0.86    # heigth of the robots joint 1 axis above the ground

# class methods
def publish_affine_tf(self, parent_frame, child_frame, affine):
    """
    string parent_frame: name of the parent frame
    string child_frame: name of the child frame
    Affine() affine: Affine transformation of the tf to publish

    Returns:
    --------
    None
    """

def publish_target_tf(self, target_name):
    """
    string target_name: name of the target to publish its tf

    Returns:
    --------
    None
    """

def get_approach_name(self, target_name):
    """
    string target_name: name of the target to reach

    Returns:
    --------
    string approach_name: name of the approach pose for the given target
    """
```

Abbildung 5-13 Abstrahierte User-Methoden im Bezug zum Koordinationskonzept zwischen AGV und Roboterarm in Logistikumgebungen Teil1

```
def publish_approach_tf(self, target_name):
    """
    string target_name: name of the target to publish its tf

    Returns:
    --------
    None
    """

def clear_tf(self, tf_name):
    """
    string tf_name: name of the target tf to clear

    Returns:
    --------
    None

    """
def get_transform(self, from_frame_rel, to_frame_rel, affine=True):
    """
    string from_frame_rel: name of the source frame frame to transform from (child)
    string to_frame_rel: name of the target frame to transform into (parent)

    Returns:
    --------
    Affine: transformation matrix from robot base to target position and orientation (4x4 numpy array, can directly passed in m
    or
    geometry_msgs/TransformStamped: transformation between the two frames if affine=False

    """

def get_park_poseCmd(self, target_name, numTestpoints, armRange, offset, visualize):
    """
    string target_name: name of the target to reach

    Returns:
    --------
    NavigateToPose.Goal() goal message to navigate agv to the nearest park pose of the given target_tf, if not reachable return

    """
```

Abbildung 5-14 Abstrahierte User-Methoden im Bezug zum Koordinationskonzept zwischen AGV und Roboterarm in Logistikumgebungen Teil2

6. Koordination AGV und Roboter

Innerhalb dieses Kapitels wird zuerst auf diverse Koordinationskonzepte zwischen dem AGV und dem Roboterarm von mobilen Manipulatoren, so wie sie in der Literatur zu finden sind, eingegangen. Danach wird das konkret innerhalb dieses Projektes implementierte Koordinationskonzept Schritt für Schritt erläutert.

6.1 Recherche zu AGV-Roboter Koordination

Durch die Kopplung einer mobilen Plattform mit einem Manipulator (Roboterarm) lässt sich der Arbeitsraum des Roboters um ein Vielfaches erweitern. Neben der möglichen Problematik der Positioniergenauigkeit der mobilen Plattform, stellt sich vor allem die Frage wie das Erreichen einer Zielpose durch den am Roboterarm befestigten Aktor (z.B. Greifer) auf Ebene der Steuerung stattfinden soll. Folgende drei Varianten wären dabei für die steuerungstechnische Kopplung der beiden Systeme denkbar [15]:

6.1.1 Lose Kopplung

Hierbei werden mobile Plattform und Manipulator als vollständig getrennte Systeme betrachtet. Die jeweiligen Bewegungen erfolgen sequenziell, d.h. die mobile Plattform bewegt sich zunächst zu einer Parkposition, welche den Manipulator in eine für das Erreichen der Zielpose günstige Stellung bringt. Daraufhin stoppt die Bewegung der Plattform und der Manipulator führt seine Arbeitsaufgabe durch. Die beiden Systeme befinden sich niemals gleichzeitig in Bewegung.

6.1.2 Vollständige Kopplung

Hierbei werden mobile Plattform und Manipulator als ein einziges System mit neun Freiheitsgraden betrachtet. Durch die Kombination ergeben sich drei redundante Freiheitsgrade. Diese Lösung bildet hinsichtlich der Kollisionsvermeidung die größtmögliche Flexibilität, da Manipulator und mobile Plattform komplexen Hindernissen zeitgleich ausweichen können. Durch die gleichzeitige Bewegung von Manipulator und mobiler Plattform ist auch eine kürzere Bearbeitungsdauer zu erwarten. Allerdings gestaltet sich die Anwendung von Bahnplanungsalgorithmen auf eine solche Struktur als äußerst aufwändig, da alle zusätzlichen Freiheitsgrade berücksichtigt werden müssen. Gleichzeitig kann eine Bewegung der Plattform bei hoher geforderter Positioniergenauigkeit kritisch sein, da diese im Allgemeinen nicht mit der des Manipulators vergleichbar ist.

6.1.3 Bedarfsorientierte Kopplung

Bei der bedarfsorientierten Kopplung werden mobile Plattform und Manipulator grundsätzlich als getrennte Systeme betrachtet, die aber „nach Bedarf“ auch gleichzeitige Bewegungen als Gesamtsystem durchführen können. Dabei werden zum Feststellen einer zulässigen Kopplung zusätzliche Randbedingungen (z.B. Hindernisse) und Sensordaten mit einbezogen. Dem Problem der vielen Lösungen durch die kinematische Redundanz soll hierbei durch die Einführung aufgabenabhängiger Kriterien begegnet werden. Die gekoppelte Bewegung soll jedoch nur dann eingesetzt werden, wenn der Nutzen dem planungsaufwand gerecht wird. So wäre es zum Beispiel bei einer hinsichtlich der TCP-Genauigkeit unkritischen Transferbewegung des Manipulators ggf. möglich die mobile Plattform schon in Bewegung zu setzen, um eine kürzere Bearbeitungszeit zu erzielen.

6.2 Umsetzung des Koordinationskonzepts

Für die gegebene Aufgabenstellung sind die Vorteile einer steuerungstechnischen Kopplung der beiden Systeme wenig relevant, da die Aufgabe nicht unter hohem Zeitdruck zu bewältigen ist und auch keine hohe Flexibilität zum Erreichen der Zielposition nötig ist. Deshalb wird das Konzept *„lose Kopplung“* gewählt. Dabei werden AGV und Roboterarm zwar als eine gemeinsame mechanische Einheit betrachtet, sind steuerungstechnisch und logisch jedoch getrennt.

Dies wird dadurch erreicht, dass jedes Gelenk (das inkludiert auch die Räder des AGV) mit seinen definierten Interfaces von dem jeweiligen Controller angesprochen wird.

Für die Räder des AGV ist dies der „Differentialdrive Controller“ für die Gelenke des Arms ist dies der „Joint Trajectory Controller“. Dabei bekommt der Differentialdrive Controller seine Sollwerttrajektorien von der Navigation aus dem Nav2 Stack, der Joint Trajectory Controller von der Bahnplanung aus MoveIt zugespielt.

So können Tasks der mobilen Robotik und stationären Robotik mit jeweils aufgabenbezogen optimierten Frameworks optimal absolviert werden.

Das Zusammenspiel der beiden Robotersysteme muss durch eine übergeordnete Logik erfolgen, die der User je nach Aufgabenstellung anpassen kann. Dazu stehen alle zuvor erläuterten Methoden zur Verfügung.

6.2.1 Zustandsdiagramm für Beispielablauf

Die Benchmarkaufgabe, welche mit dem Robotersystem absolviert werden soll, wird wie folgt definiert und deckt damit den kompletten zurzeit verfügbaren Funktionsumfang ab.

„Der mobile Manipulator soll ausgehend von einer Home-Position (z.B. Ladestation) zu einer Parkposition navigieren und sich mit dem Heck zum Regal ausrichten. Daraufhin soll der Arm eine Pose, welche sich zwischen zwei Regalböden befindet, anfahren. Abschließend soll der Arm sich wieder in eine sichere Home-Pose bewegen, bevor der mobile Manipulator zurück zu seiner Home Position navigiert.“

Zur Verdeutlichung der Komplexität des Ablaufes ist nachfolgend ein Zustandsdiagramm an die UML-Notation angelehnt dargestellt, welches die einzelnen benötigten Funktionalitäten in Bezug setzt.

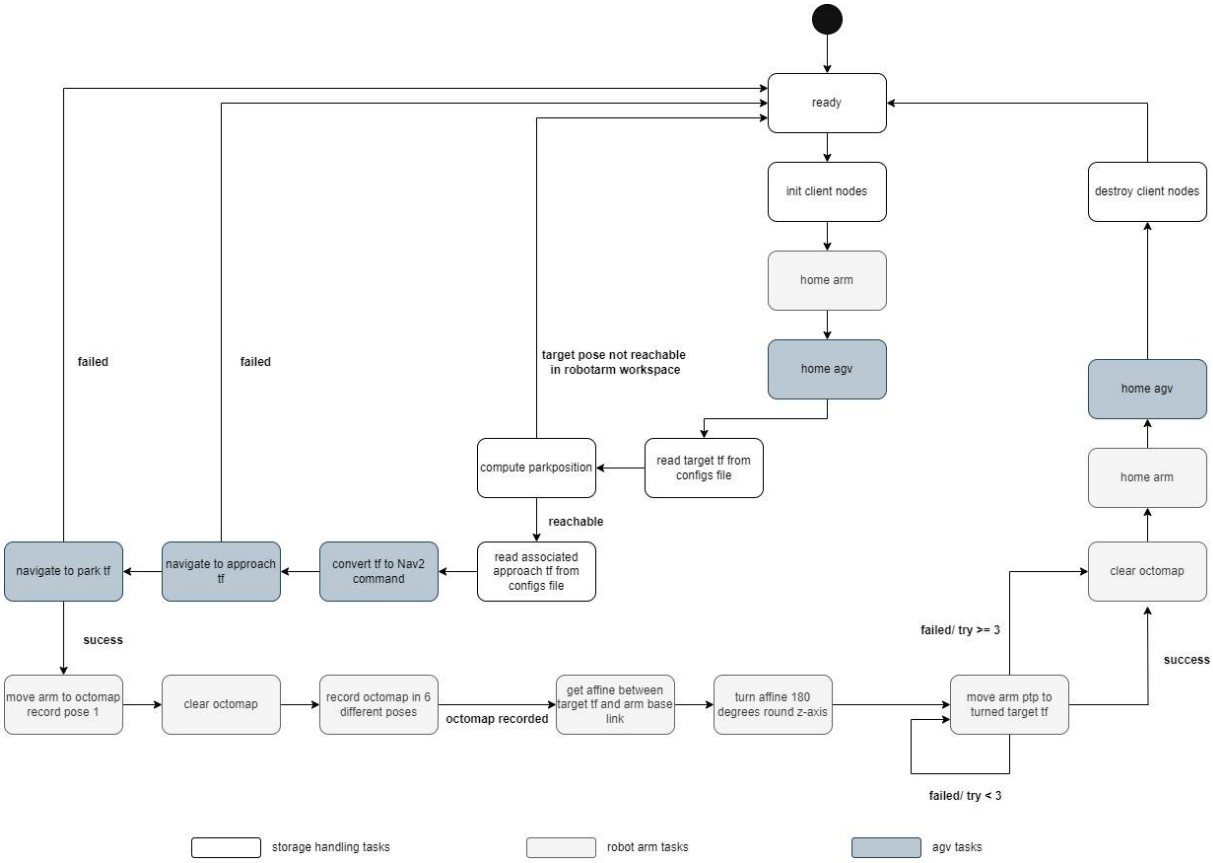

Abbildung 6-1 Zustandsdiagramm übergeordnetes Kontrollprogramm für Benchmarkaufgabe

6.2.2 Anlegen der Lagerpositionen

Grundlage für das Handling von Lagerpositionen stellt eine eindeutige Konvention dar, wie diese im System vom User einmalig angelegt werden müssen.

Grundsätzlich wurde entschieden, die Lagerpositionen in einer yaml-Konfigurationsdatei unter *„src/master_application/config“* anzulegen. Die gewählte Struktur ist nachfolgend exemplarisch für die Lagerposition „shelf_test“ gezeigt.

```
shelf_test:
  approach: approach_corridor_2
  position:
    x: 2.5
    y: 0.0
    z: 0.75
  orientation:
    x: 0.0
    y: 0.0
    z: 1.0
    w: 1.0
```

Abbildung 6-2 Struktur beim Anlegen einer Lagerposition

Der User muss eine Position und Orientierung (Quaternion) relativ zum Ursprungskoordinatensystem der Karte der Umgebung anlegen. Zusätzlich muss zu jeder Lagerposition festgelegt werden, welcher Approachposition diese zugeordnet ist. Auf die Notwendigkeit einer Approachposition bei der Navigation wird im Folgenden noch weiter eingegangen.

Die Orientierung der Lagerposition ist dabei in der festgelegten Konvention unbedingt einzuhalten. Das AGV wird sich immer aus der positiven X-Richtung an die Lagerposition annähern, daher muss die X-Achse der Lagerposition zwingend Richtung Gang/ Korridor zwischen den Regalen zeigen. Gleiches gilt für Lagerpositionen, die auf den Tischen definiert sind.

Die Abbildung 6-4 zeigt unter anderem die oben definierte Lagerposition „shelf_test“ innerhalb der Karte.

6.3 Berechnung der Parkposition und Navigation dorthin

Durch das gewählte Koordinationskonzept zwischen AGV und Roboterarm muss basierend auf der gegebenen Karte und der nach der festgelegten Konvention angelegten Lagerposition eine bestmögliche Parkposition für das AGV berechnet werden. Diese berechnete Parkposition muss den folgenden Ansprüchen genügen:

1) Die Parkposition darf nicht mit Hindernissen in der Umgebung kollidieren.
2) Das AGV muss auf Grund seines Scanner-Schutzfeldes und der Montageposition mit dem Heck Richtung Regal oder Tisch stehen.
3) Die Basis des Roboterarms soll möglichst nahe an der zu erreichenden Lagerposition positioniert werden.
4) Kann der Roboterarm eine gegebene Lagerposition von keiner Parkposition aus erreichen, soll die Navigation nicht gestartet werden, sondern ein Fehler gemeldet und der Ablauf abgebrochen werden.

Nachfolgend wird zuerst der Algorithmus zur Berechnung einer validen, optimalen Parkposition erläutert, bevor anschließend auf die aufgetretenen Probleme bei der Navigation zur Parkposition eingegangen wird.

6.3.1 Suchalgorithmus zur Berechnung der optimalen Parkposition

Basis der Berechnung einer Parkposition stellt einerseits die nach der dargelegten Konvention, andererseits die aufgezeichnete Karte der Umgebung dar.

Folgendes Zustandsdiagramm zeigt den Berechnungsalgorithmus schematisch auf.

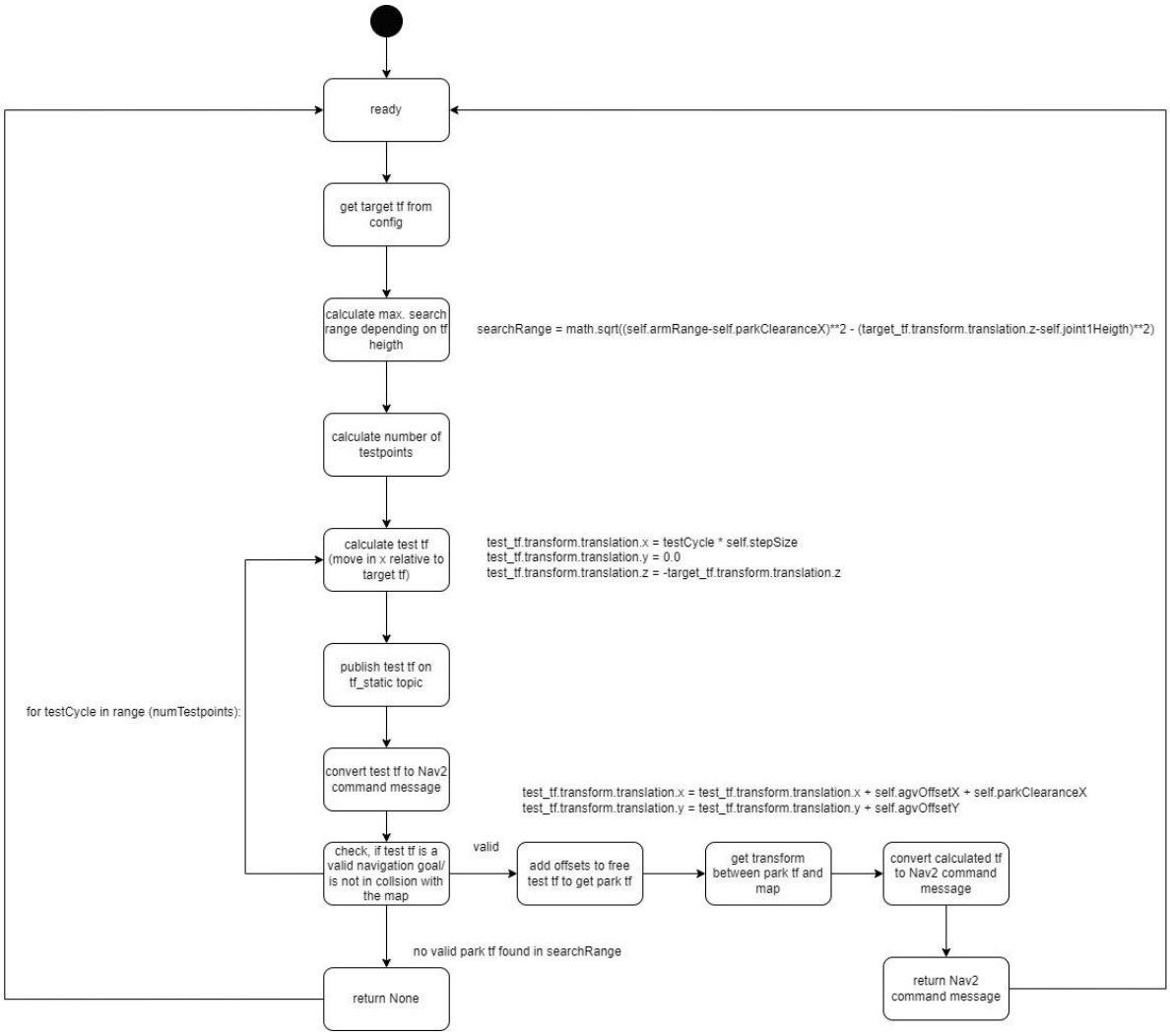

Abbildung 6-3 schematischer Ablauf der Berechnung einer optimalen Parkposition

Dabei gilt zu beachten, dass sofern keine Parkposition innerhalb einer bestimmten Distanz, die der Roboterarm noch bewältigen kann, gefunden wird, der Algorithmus „None“ zurückgibt. Dieser Rückgabewert kann in dem übergeordneten Steuerungsprogramm zum Fehlerhandling geprüft werden.

Darüber hinaus muss an dieser Stelle betont werden, dass der Algorithmus keine Garantie dafür gibt, dass der Roboterarm von der gegebenen Parkposition tatsächlich die Lagerposition erreichen kann.

Der Algorithmus geht bei der Berechnung der maximalen Suchrange von einer idealen Umgebung aus. Dies bedeutet es, es werden weder 3D Kollisionsgeometrien (Octomap) noch Positionierfehler bei der Navigation berücksichtigt. Er dient lediglich als erste Abschätzung, ob es lohnend ist, die berechnete Parkposition überhaupt anzufahren.

Folgender Bildschirmausschnitt in Abbildung 6-4 zeigt die visualisierten Zwischenschritte bei der Berechnung der Parkposition für das AGV auf Basis der Lagerposition. Wichtig dabei zu beachten sind die Pfeile, welche die jeweiligen Transformationen verbinden. Diese stellen dar, welche Transformation relativ zu welcher definiert ist.

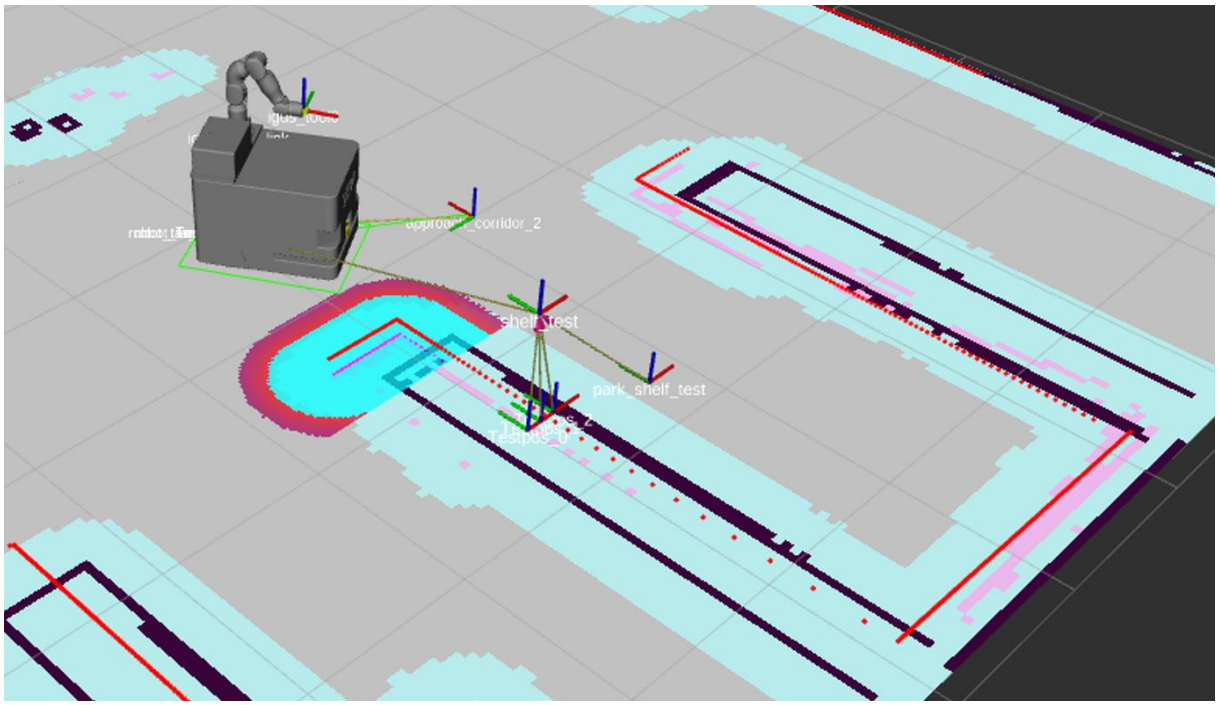

Abbildung 6-4 benötigte Transformationen für die Benchmarkaufgabe

6.3.2 Navigation zur Parkposition

Nachdem eine valide Parkposition berechnet wurde, von welcher aus der Roboterarm die gewünschte Lagerposition erreichen kann, startet die Navigation zur Parkposition automatisch.

Zu Beginn wurde direkt von der Homeposition zur Parkposition navigiert, was sich allerdings als nicht prozesssicher herausstellte.

Durch das reale Schutzfeld des Lidarscanners und der Montageposition des Roboterarms auf dem AGV war es eine bereits erläuterte Anforderung, dass das AGV mit dem Heck Richtung Regal/ Tisch steht. Dafür muss es nach einer Vorwärtsfahrt bis kurz vor die Parkposition eine 180° Drehung vollziehen, um daraufhin rückwärts auf die Zielposition zurückzusetzen.

Diese Bewegung konnte gerade innerhalb der schmalen Regalkorridore nicht zuverlässig durchgeführt werden. Die folgenden Gründe konnten dafür ermittelt werden:

1) Durch die Differential Drive Kinematik kann das AGV sich nicht holonom bewegen. Dies bedeutet, eine seitliche Positionskorrektur ist nur durch erneutes vor- und zurücksetzen kombiniert mit einem Lenkbefehl möglich. Da diese Operation allerdings dem Prinzip eines heuristikgetriebenen A* Algorithmus widerspricht, kann sie nicht ausgeführt werden. Zusammengefasst kann die Parkposition nicht genau genug angefahren werden, da Positionskorrekturen und eine Rotation auf der Stelle durch die eingeschränkten Bewegungsfreiheitsgrade des AGV nicht umsetzbar sind.
2) Durch die engen Regalkorridore und durch die Costmaps steigenden Kosten, desto näher das AGV an Hindernisse heranfährt, wurde die Berechnung eines geeigneten Pfades zusätzlich erschwert. Speziell die 180° Drehung innerhalb der Korridore war dadurch unzuverlässig.
3) Durch die Montageposition des Lidarscanners in der Front fehlen dem AGV direkte Abstandsinformationen am Heck, diese müssen durch weiter entfernte Konturen in der Front des AGV antizipiert werden. Dies führte zu einer unzuverlässigen Lokalisierung. Damit stimmte die theoretische Position des AGV nicht mehr mit der tatsächlichen überein.

Die 180° Rotation kurz vor der Parkposition, welche sich in der Regel nahe einem Hindernis befindet, konnte als Hauptursache für fehlende Prozesssicherheit identifiziert werden.

Der Ansatz dies zu lösen ist, eine zusätzliche Approachposition für jeden Regalkorridor/ Tisch zu definieren. Diese wird vom User einmalig in der Konfigurationsdatei, äquivalent zu den Lagerpositionen, angelegt. Darüber hinaus wird jeder Lagerposition die entsprechende Approachposition zugewiesen (siehe Abbildung 6-2).

Die Approachposition ist bereits so orientiert, dass das AGV mit dem Heck zum Regalkorridor/ Tisch steht. Aus dieser Ausgangslage erkennt der A* Algorithmus automatisch, dass eine Rückwärtsfahrt zur Parkposition am schnellsten zum Ziel führt und damit die geringsten Kosten zur Folge hat. Ein optimaler Pfad, ohne die Notwendigkeit einer nachträglichen Positionskorrektur auf engstem Raum nach einer 180° Drehung wird generiert und zuverlässig abgefahren.

Damit sind eine ausreichende Positioniergenauigkeit und Zuverlässigkeit gegeben. Folgende Darstellung zeigt das AGV bei der Navigation zur Approachposition (links) und anschließend zur Parkposition (rechts). In grün zu sehen ist der jeweilige Pfad.

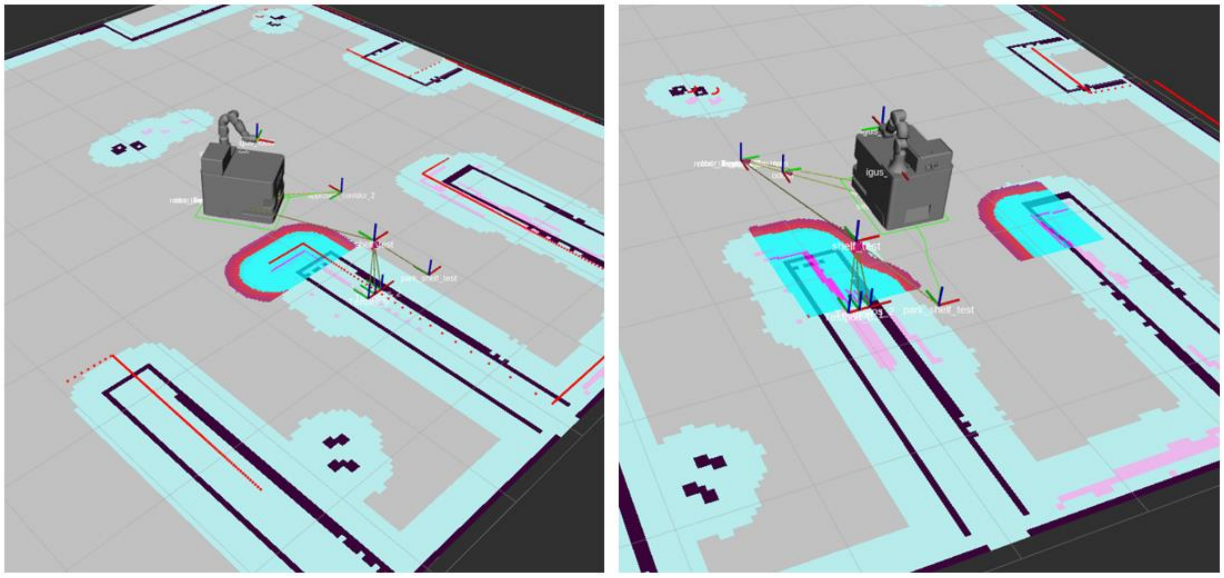

Abbildung 6-5 Navigationsablauf des AGV von Homeposition zu berechneter Parkposition

6.4 Aufnahme der Octomap und Bewegung des Roboterarms zur Lagerposition

Nun befindet sich er mobile Manipulator an einer validen Parkposition, von welcher aus der Roboterarm theoretisch die geforderte Lagerposition erreichen kann.

Bei der Beurteilung dieser Aussage wurden, allerdings wie in Kapitel 6.3.1 erläutert, keine Kollisionsgeometrien berücksichtigt. Diese müssen nun im ersten Schritt rekonstruiert werden, sodass im zweiten Schritt eine kollisionsfreie Robotertrajektorie von der sicheren Homepose des Arms zur geforderten Lagerposition berechnet und ausgeführt werden kann.

6.4.1 Aufnahme der Octomap

In Kapitel 5.5.4 wurde bereits das Konzept der Octomap zur Kollisionsvermeidung bei Bewegungen des Roboterarms erläutert.

An dieser Stelle wird wiederholt darauf hingewiesen, dass die Octomap eine sehr genaue Repräsentation der lokalen Kollisionsgeometrien im Arbeitsbereich des Roboterarms sein muss. Der hinterlegte Bahnplaner in MoveIt benötigt die Octomap als Grundlage und plant die Robotertrajektorien daraufhin auch sehr nahe an bekannten Kollisionsgeometrien vorbei.

Die Octomap kann nur im Stillstand des AGV aufgenommen werden und muss nach jeder AGV-Bewegung verworfen werden, denn die Octomap wird relativ zum Basiskoordinatensystem des Igus Roboterarms definiert. AGV-Bewegungen können nicht berücksichtigt werden, womit ein Abgleich zwischen der Navigationskarte und der Octomap nicht möglich ist.

Wie im Zustandsdiagramm (Abbildung 6-1) dargestellt, wird die Octomap im implementierten Ablauf aus sechs verschiedenen Roboterposen aufgenommen und anschließend konkateniert. Dies ist notwendig, da die am TCP montierte Kamera den Bewegungsraum des Roboters aus einer Pose nicht vollständig erfassen kann.

Die genaue Anzahl der Aufnahmeposen könnte bei der zeitlichen Optimierung des Ablaufes auf zwei reduziert werden, wird hier allerdings zu Anschauungszwecken mit sechs beibehalten. Die nachfolgende Grafik zeigt die aufgenommene Octomap in der Parkposition des AGV im durchgeführten Beispielablauf.

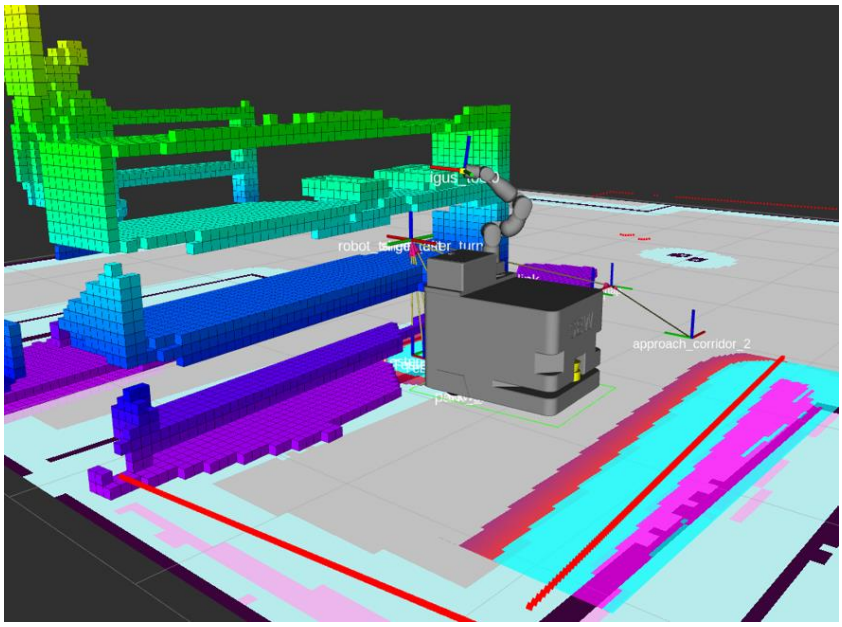

Abbildung 6-6 aufgenommene Octomap als bei Bahnplanung zu berücksichtigende Kollisionsgeometrien

6.4.2 Bewegung Roboterarm zur Lagerposition

Nun stehen dem MoveIt Framework alle Informationen zur Verfügung, die bei der Bahnplanung zur geforderten Lagerposition benötigt werden. Ziel an dieser Stelle ist es, das TCP-Koordinatensystem des Roboterarms exakt auf dem der gewünschten Lagerposition zu positionieren. Zu beachten ist hier, dass die zu demonstrationszwecken verwendeten Lagerpositionen nicht den Positionen der KLTs in der Simulation entsprechen. Die KLTs dienen hier nur der Anschauung.

Dafür muss die nach der definierten Konvention definierten Lagerposition allerdings noch 180° um ihre Z-Achse gedreht werden, da die Z-Achse des TCP Richtung Regal/ Tisch zeigt und nicht davon weg.

Da innerhalb der Applikation mit Transformationen (homogene Transformationsmatrizen/ Affine) gerechnet wird, ist die Drehung der Zielpose durch eine einfache Matrizenmultiplikation umzusetzen. Wichtig dabei ist die Multiplikationsreihenfolge einzuhalten, um zuerst die lokale Transformation von Lagerpose zu Zielpose durchzuführen und anschließend die Zielpose in das Roboterbasissystem zu transformieren. Die so berechnete Gesamttransformation beschreibt die geforderte Roboterzielpose im Basiskoordinatensystem und kann von MoveIt ausgeführt werden. Dabei wird die Transformation zwischen Lagerpose und Roboterbasis über die Lokalisierung des AGV, Odometriedaten und Teile der Vorwärtskinematik ermittelt.

$$^{Zielpose}T_{Roboterbasis} = {}^{Roboterbasis}T_{Lagerpose} * {}^{Lagerpose}T_{Zielpose}$$

Als Bewegungsart wird hier ein PTP-Befehl verwendet, da hierbei bei der Bahnplanung aktiv Kollisionen umplant werden können. Bei einem LIN-Befehl wäre dies nicht möglich.

Folgende Abbildung zeigt den Roboterarm in seiner Zielpose zwischen den Regalböden. Die Berechnung einer Kollisionsfreien Trajektorie dorthin kann als komplex bewertet werden, wodurch hier ein funktionierendes System in einem herausfordernden Benchmark nachgewiesen werden kann.

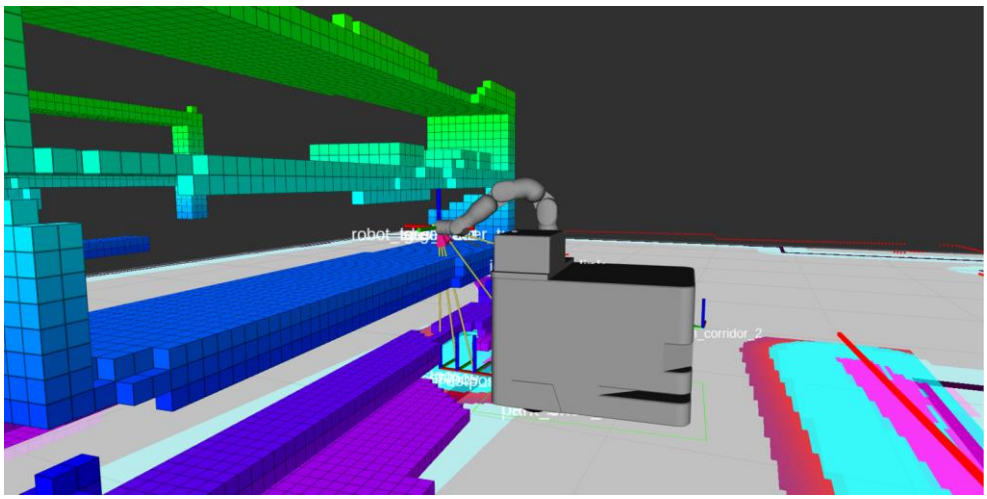

Abbildung 6-7 Roboterarm an der Zielpose zwischen den Regalböden (Benchmarkaufgabe)

6.4.3 Problem Lokalisierungs- und Positionierunsicherheit des AGV

Prinzipiell wurde die geforderte Benchmarkaufgabe damit erfolgreich absolviert. Die Funktionalität des Robotersystems ist nachgewiesen und kann auf beliebige Lagerpositionen adaptiert werden. Allerdings muss an dieser Stelle auf ein typisches Problem in der Koordination zwischen AGV und Roboterarm bei mobilen Manipulatoren hingewiesen werden, mit welchem sich innerhalb dieses Projektes nicht auseinandergesetzt wurde.

Wie in diversen Abbildungen der Octomap (siehe z.B. Abbildung 6-6) zu erkennen ist, stimmt die aufgenommene Octomap nicht exakt mir der zuvor aufgenommenen 2D Karte der Benchmarkumgebung überein. Dies liegt an der Lokalisierung des Robotersystems über Lidarscanner innerhalb der 2D Karte. Die Lokalisierung durch die Lidarscanner und darauf basierende Navigation weist nur eine prinzipbedingte Positioniergenauigkeit von ca. +/- 0.1m auf und ist damit für Navigationsaufgaben ausreichend.

Die Octomap wiederum stellt dabei die Umgebung im Bezug zum Roboter mit der Messungenauigkeit der Kamera (<1 mm) dar. Daher sind Kollisionen bei einer Roboterarmbewegung trotz mangelhafter Lokalisierung und Positionierung ausgeschlossen.

Die Lagerposition ist allerdings innerhalb des Referenzkoordinatensystems der 2D Karte definiert. Dadurch wird der Positionierfehler des AGV bei der Berechnung der Transformation zwischen Lagerpose und Roboterarmbasis berücksichtigt, der Lokalisierungsfehler allerdings nicht. Dies resultiert wiederum in einer relativen Abweichung zwischen angefahrener Lagerpose und der eigentlich angestrebten, welche gerade dem Lokalisierungsfehler entspricht.

Um dieses Problem zu lösen ist es ein gängiger Ansatz, die Lagerposition nach der abgeschlossenen Navigation durch eine Kamera z.B. durch Arucomarker zu detektieren und relativ zum Robotersystem anzupassen. Damit wird der Lokalisierungsfehler ausgeglichen.

7. Hardwareaufbau

Dieses Kapitel befasst sich mit der Montage des Igus ReBeL auf dem AGV sowie der der elektrischen Verschaltung aller Komponenten.

7.1 Mechanische Montage

Um den Roboterarm mechanisch auf dem AGV zu montieren, muss eine geeignete Position dafür festgelegt werden. Da der Roboterarm mit 0,66 m aber selbst auf gleicher Höhe nur eine geringe maximale Reichweite besitzt, sollte beim Parken des AGV ein möglichst geringer Abstand von Roboterarmbasis zur Zielpose angestrebt werden.

Der von SEW einprogrammierte Schutzradius des Lidarscanners hat sich dabei als ein limitierender Faktor herausgestellt. Dieser verhindert, dass das AGV vorwärts um mehr als ca. 0,5 m an Hindernisse heranfährt. Werden Objekte im Schutzfeld detektiert, führt das AGV einen Not-Halt aus. Um das AGV wieder in den betriebsbereiten Zustand zu versetzen, muss dieses maunell über die Bedieneinheit am Heck aus dem Gefahrenbereich manövriert werden. Deshalb soll das AGV Rückwärts an das Regal fahren. Auf dieser Seite ist lediglich eine mit einem Sensor versehene Gummilippe angebracht, welche eine Kollision erfasst. Hier wird ein Not-Halt nur bei Berührung der Lippe ausgelöst, somit kann das AGV rückwärts nahezu unlimitiert nah an Hindernisse heranfahren, ohne dass ein manuelles Eingreifen nötig ist.

Der Roboterarm wird demnach auf der Rückseite des AGV montiert. (Abbildung 7-1).

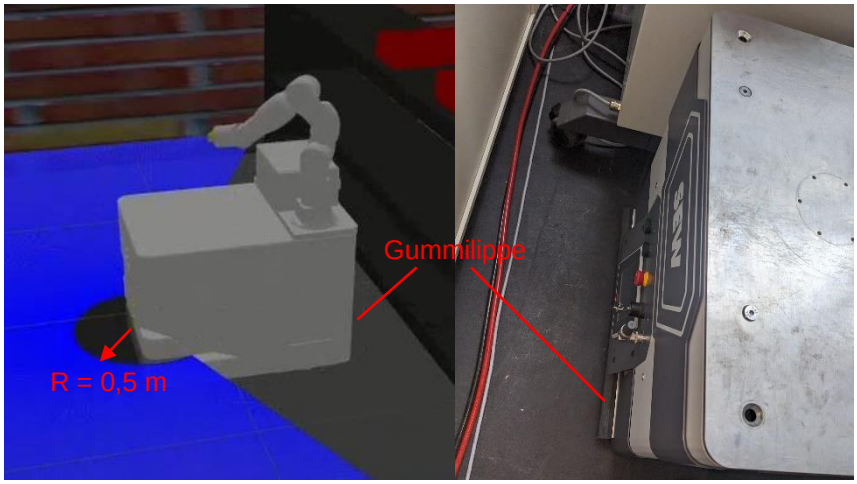

Abbildung 7-1: Schutzradius LIDAR und sensorische Gummilippe

Um die Stromversorgung zu gewährleisten, muss zusätzlich ein Schaltschrank angebracht werden. Um den Aufbau in Zukunft einfach rückgängig machen zu können, oder die Anordnung der Komponenten zu ändern, werden diese nicht direkt auf die Montageplatte des

AGV geschraubt. Stattdessen wird eine Holzplatte aufgelegt und durch die Montagebohrungen der Metallplatte fixiert, sodass keine zusätzlichen Bohrungen in der Metallplatte nötig sind. Abbildung 7-2 zeigt den so entstandenen Aufbau.

Das erste Gelenk des Roboters ermöglicht eine Drehung um 360° um die vertikale Achse. Dabei ist jedoch keine kontinuierliche Drehung möglich, an der Endstellung muss die Bewegung gestoppt werden. Da die für das Greifen von Objekten relevante Bewegungen des Roboters hauptsächlich nach hinten erfolgt, wurde der „Umkehrpunkt“ durch die entsprechende Montage der Roboterbasis nach vorne gelegt. So kann der Freiheitsgrad bei der Bewegung nach hinten ungestört ausgenutzt werden.

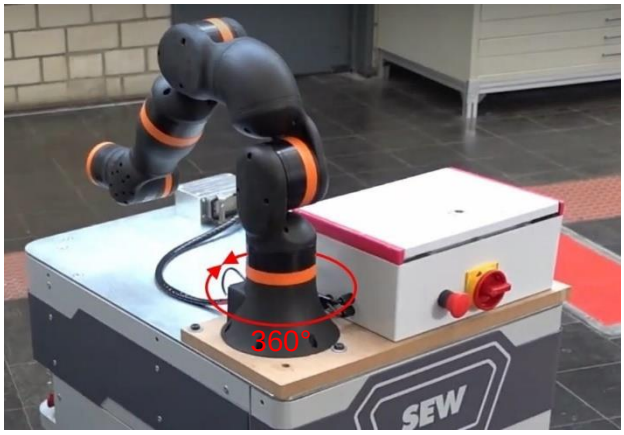

Abbildung 7-2: Hardware mit Holzplatte auf AGV montiert, Bewegungsradius Roboterarm

Im Fuß der Open-Source-Variante des Igus ReBeL befindet sich ein Hohlraum, in dem bei der Plug-and-Play Variante ein Raspberry Pi zur Steuerung verbaut ist. Diesem Beispiel wurde nachgegangen und eine Abdeckung konstruiert, welche ebenfalls den verwendeten Raspberry Pi im Innern des Hohlraumes fixiert. Zusätzlich ist hinter der Abdeckung noch der verwendete USB-to-CAN-Konverter verstaut. Der Raspberry Pi befindet sich in einem Gehäuse mit eingebautem Lüfter. Um eine ausreichende Luftzufuhr dafür zu gewährleisten, sind im oberen Bereich der Abdeckung Lüftungsschlitze vorgesehen. Neben dem Raspberry Pi ist noch ein

Abbildung 7-3: Montage des Raspberry Pi in Abdeckung (links) und Einbau in Fuß des Igus ReBeL (rechts)

Taster angeordnet, mithilfe dessen ein Neustart des Raspberry PI's nach Betätigung des Not-Aus-Tasters initiiert werden kann.

7.2 Elektrischer Anschluss

Um den Roboter mit Energie zu versorgen, muss die benötigte Spannung von den Akkus des AGV abgegriffen werden. Im Schaltplan des AGV wurde dabei ein gesicherter Lastausgang ausfindig gemacht (Klemme 3X9), welcher die Akkuspannung von 48 V bereitstellt [16]. Die Antriebe des Igus ReBeL benötigen eine Spannung von 24V, wozu ein passender Spannungswandler verbaut werden muss. Für die Versorgung des Raspberry Pi's sowie der Versorgung der Motortreiber des Roboterarms wird ein Spannungsniveau von 5V benötigt. Auch dafür ist ein separater Spannungswandler nötig. Zusätzlich sollen Hauptschalter, Not-Aus-Schalter und eine Sicherung in den Stromkreis integriert werden. Um alle Komponenten sicher anbringen zu können, wird ein Schaltschrank benötigt. Dieser darf nicht zu groß dimensioniert werden, um den Platz auf dem AGV für eventuelle spätere Aufbauten nicht zu verschwenden und die Bewegung des Roboterarms möglichst wenig einzuschränken. Besonders wichtig für Zwieteres, ist dass der die Höhe des Schaltschrankes geringer ist, als die der 2. Achse des Roboters (Schultergelenk).

Abbildung 7-4 zeigt den Schaltplan. Die 48V Versorgungsspannung wird zunächst über eine Sicherung F1 zum Schutz aller Komponenten zum Hauptschalter geführt. T1.1 stellt einen DC/DC Wandler dar, welcher die 24V-Versorgungsspannung für die Antriebe des Igus ReBeL bereitstellt. Dieser kann durch Betätigen des Not-Aus Schalters S2 vom Stromkreis getrennt werden. Dies entspricht einem ordnungsgemäßen Abschalten des Roboters im Sinne der Betriebsanleitung, da die Steuerspannung aufrechterhalten wird. Die 5V Steuerspannung wird zur Versorgung der Motortreiber des Igus ReBeL sowie zum Betrieb des Raspberry Pi benötigt. Sie wird durch den zweiten DC/DC Wandler T1.2 bereitgestellt.

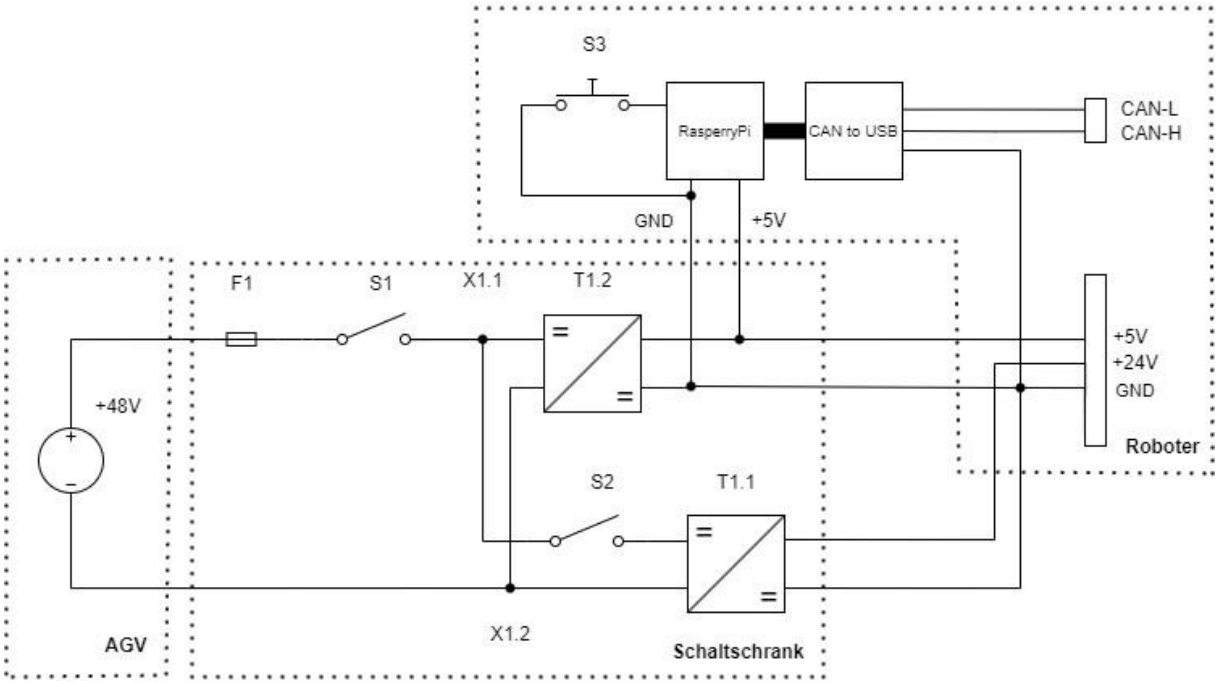

Abbildung 7-4: Schaltplan

Abbildung 7-5 zeigt den Schaltschrank mit den entsprechenden Elementen aus dem Schaltplan.

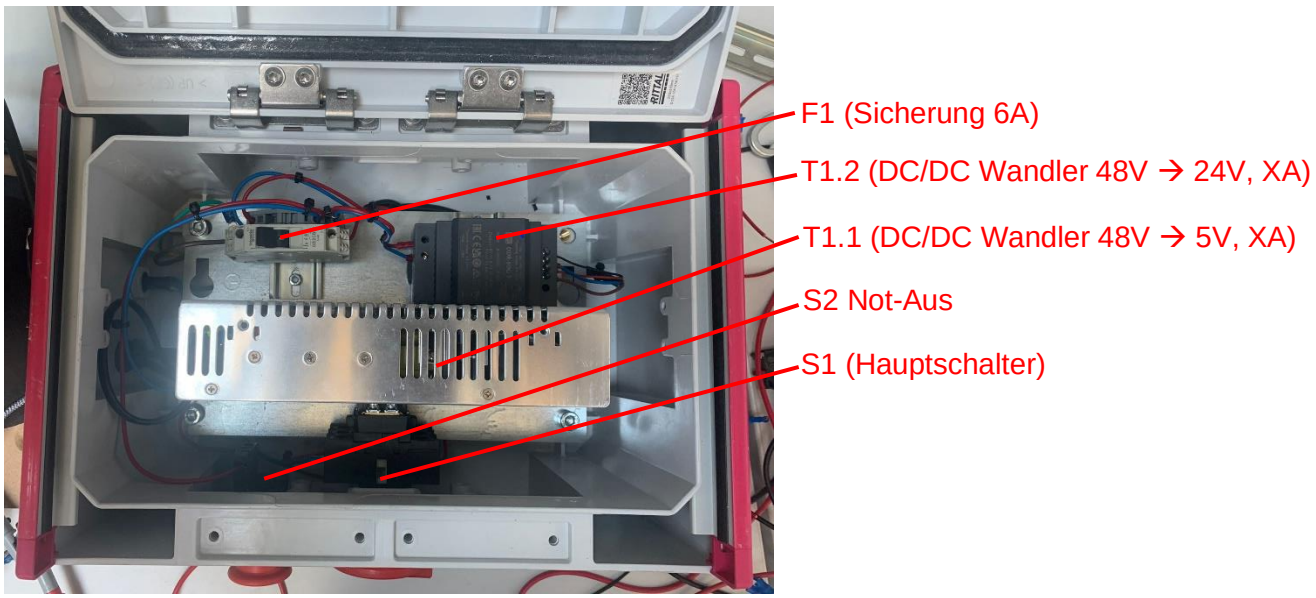

Abbildung 7-5: Schaltschrank

Abbildung 7-6 zeigt das vollständige Hardware-Setup. Die Stromversorgung der Payload sowie die Ethernet-Verbindung zum AGV sind in einem Kabelbündel zusammengefasst und werden über den vorgesehenen Anschluss ins Innere des AGV geführt. Zwischen Schaltschrank und AGV verlaufen zusätzlich die benötigten Leitungen für die Stromversorgung des Roboterarms sowie des Raspberry Pi und Motortreiber.

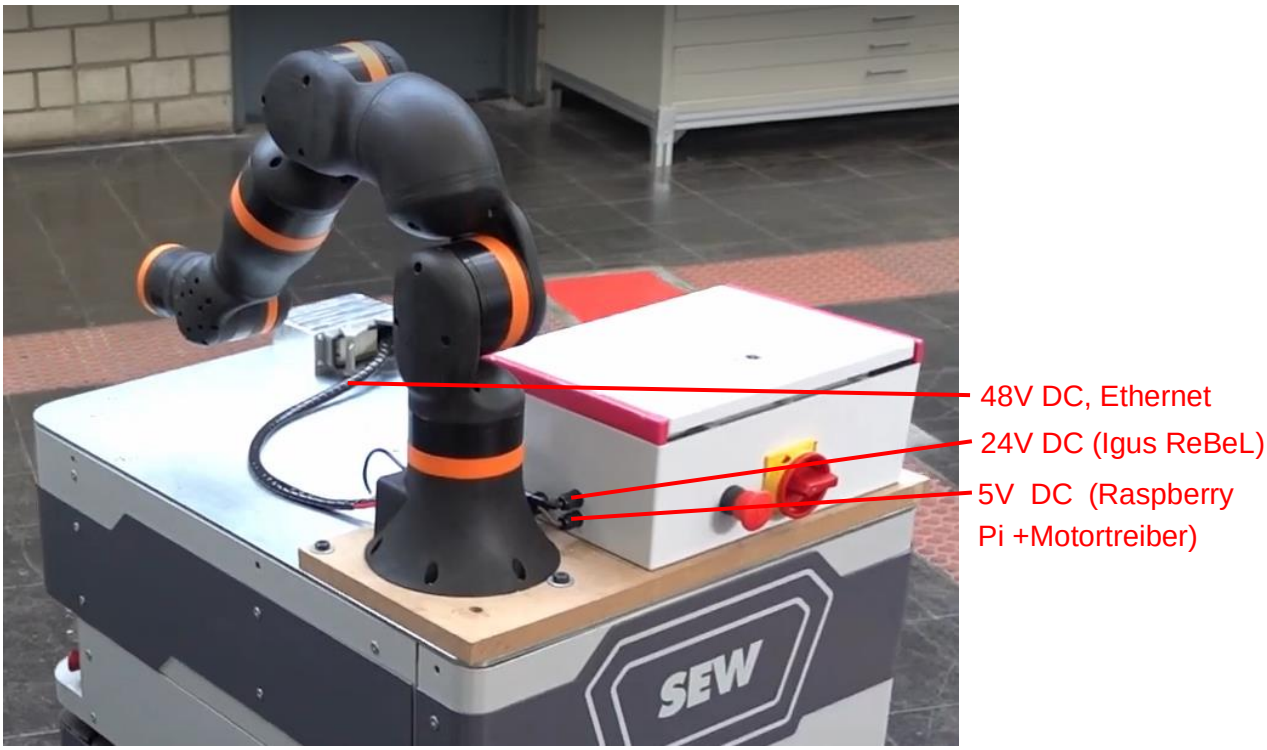

Abbildung 7-6: vollständiges Hardware-Setup

7.3 Ansatz Informationsverarbeitung

Das Konzept für die Informationsverarbeitung ist in Abbildung 7-7 dargestellt. Über einen User-PC wird der Roboter und das AGV ferngesteuert.

An den PC ist ein Xbox Controller über ein USB-Kabel verbunden mit dem das AGV verfahren werden kann. Der Raspberry Pi 5 befindet sich im Fuß des Roboters und hostet ein Wifi Netzwerk, mit dem sich der User-PC verbindet. Der Raspberry Pi ist über einen USB-CAN-Adapter mit dem Roboter, und über ein EtherCAT Kabel mit dem AGV verbunden und kommuniziert so mit diesen.

Auf dem PC und auf dem Raspberry Pi laufen die gleichen Docker Container mit den ROS2 Implementierungen. Da die *ROS_DOMAIN_ID*'s auf dem PC und Raspberry Pi identisch konfiguriert sind und sie sich im selben Wifi Netzwerk befinden, können beide Geräte über mehrere Docker Container hinweg über ROS2-Topics, Services und Actions kommunizieren.

Dazu muss die Variable „ROS_DOMAIN_ID" in der .env-Datei eines jeden Docker Containers gesetzt und beim starten des Containers in diesen gemountet werden.

Die Docker Container, mit welchen der Roboterarm gesteuert wird, haben eine ROS_DOMAIN_ID von 1, die für das AGV von 2. So können die Systeme unabhängig ohne Interferenzerscheinungen im gleichen Netzwerk betrieben werden.

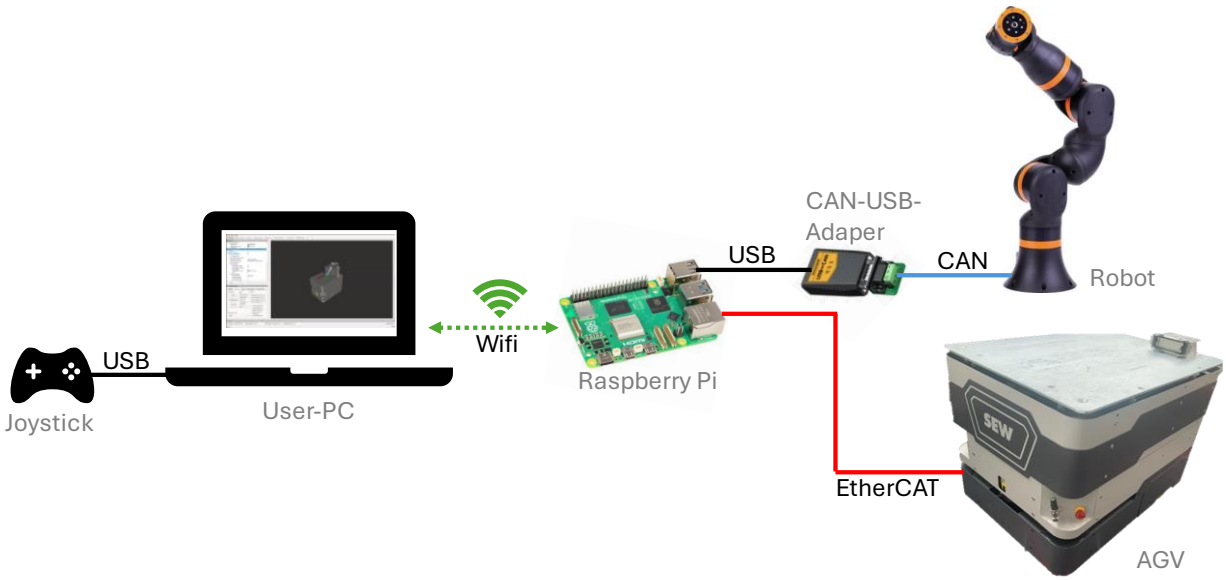

Abbildung 7-7: Informationsverarbeitung Ansatz

8. Inbetriebnahme Igus ReBeL Roboter

Im Rahmen dieser Projektarbeit wurde der Roboter „ReBeL 6 DOF“ der Firma Igus [17] verwendet. Mit einem Preis von rund 5000 € ist der ReBeL im Vergleich zu herkömmlichen Industrierobotern günstig und daher gut für diesen Demoaufbau geeignet. Er verfügt über sechs Achsen, eine maximale Nutzlast von 2 kg und hat laut Hersteller eine Wiederholgenauigkeit von ±1 mm. Mit einer maximalen Reichweite von 664 mm und einer Nennreichweite von 400 mm kann der ReBeL vielfältige Aufgaben in der Montage, Qualitätsprüfung sowie im Pick & Place-Bereich übernehmen. Er kann sowohl in einer Plug-and-Play-Version mit integrierter Steuerung als auch als Open-Source-Version gekauft werden. Die Open-Source-Version ist mit rund 4200 € etwas günstiger als die Plug-and-Play-Version [17].

Abbildung 8-1: Igus ReBeL 6 DOF Roboter [17]

In dieser Projektarbeit wurde die Open-Source-Version verwendet. Bei dieser Version befindet sich keine Steuerung im Fuß des Roboters. Lediglich ein JST-Stecker kommt aus dem inneren des Roboters (siehe Abbildung 8-3). Über diesen kann der Roboter mit Spannung versorgt werden und die Motoren der Achsen über CAN angesprochen werden. Mit dem passenden Gegenstück (JST PAP-07V-S [18]) zum Stecker wurden die Kabel verlängert und wie in Kapitel 7.2 beschrieben, angeschlossen.

Für die CAN-Kommunikation wurde hier der USB-CAN-Konverter von Innomaker [19] verwendet. Über den USB-Stecker kann der Adapter dann an einen PC oder den Raspberry Pi angeschlossen werden. Damit die Kommunikation mit dem CAN-Adapter richtig funktioniert muss die Verbindung eingerichtet werden. Dies kann über das Terminal mit folgendem Befehl gemacht werden:

sudo ip link set can0 up type can bitrate 500000 restart-ms 1000

Auf dem Rasperry Pi 5, der in dieser Projektarbeit aufgesetzt und in den Roboterfuß integriert wurde, wird dies beim Booten automatisch konfiguriert. Wie dies eingerichtet werden kann ist im Readme der zugehörigen GitHub Repositories nachzulesen [20].

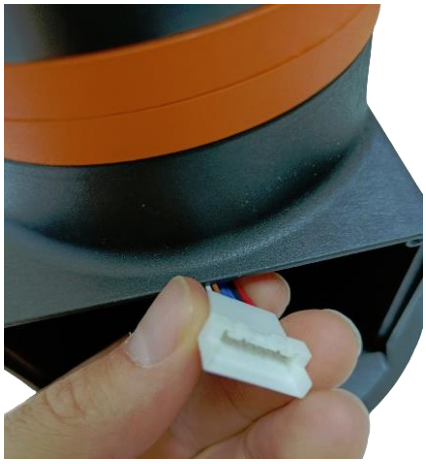

Abbildung 8-3: Stecker Igus ReBeL Open-Source Version

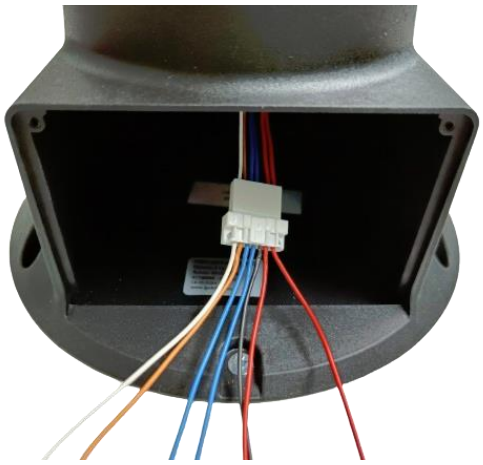

Abbildung 8-2: Stecker Igus ReBeL Open-Source Version - Stecker angeschlossen (rot 24V, blau GND, schwarz 5V, weiß CAN-L, braun CAN-H)

8.1 Softwareaufbau Igus ReBeL

Als Basis unserer ROS2 Implementierung haben wir das offizielle GitHub Repository des Igus ReBeL verwendet [21]. Die Implementierungen wurden von uns angepasst und in einen Docker Container integriert. Der Code dafür wurde von uns auf GitHub veröffentlicht und dokumentiert [20], [22].

Das Gesamtkonzept unserer Softwarearchitektur mit dem ROS2 Softwareframework wurde bereits in Kapitel 5 dargelegt. Große Teile der Software für die Hardware-Inbetriebnahme des Igus ReBeL Roboters als auch für das AGV (siehe Kapitel 9) können dank des ROS2 Frameworks aus dem Gesamtkonzept unserer Gazebo Simulation übernommen werden.

Ein großer Vorteil von ROS2 ist, dass sämtliche Hardware durch geeignete Hardware-Interfaces abstrahiert wird. Dies bedeutet, dass die komplette Software, welche mit simulierter

Hardware entwickelt wird, ohne Anpassungen auch für das reale Robotersystem funktioniert. Es müssen lediglich die Hardwareinterfaces getauscht werden.

Außerdem kann unsere Software wegen der Verwendung von Docker problemlos auf unterschiedlicher Hardware migriert und in Betrieb genommen werden.

Während der Entwicklungsphase kann der Docker Container direkt vom PC aus gestartet und bedient werden. Dafür muss der CAN-Adapter mit dem PC verbunden und wie oben beschrieben eingerichtet werden.

Für den regulären Betrieb wurde ein Raspberry Pi 5 mit Ubuntu 24.04 Betriebssystem zusammen mit dem CAN-Adapter in den Roboterfuß integriert. Auf diesem Raspberry Pi läuft sowohl der Docker Container für den Roboter als auch der Container für das AGV (siehe Kapitel 9).

Außerdem stellt der Raspberry Pi ein Wifi Netzwerk bereit mit dem man sich mit dem PC verbinden und den Roboter somit fernsteuern kann. Dafür muss auf dem PC der gleiche Docker Container mit gleicher *ROS_DOMAIN_ID* aktiv sein. Die *ROS_DOMAIN_ID* wird in der *.env-File* definiert.

Der Roboter kann von diesem PC dann entweder über RViz (siehe Abbildung 8-4) oder über ein Python Skript mit Bewegungsbefehlen gesteuert werden. Dies wurde bereits in Kapitel 5.5.5 näher erklärt.

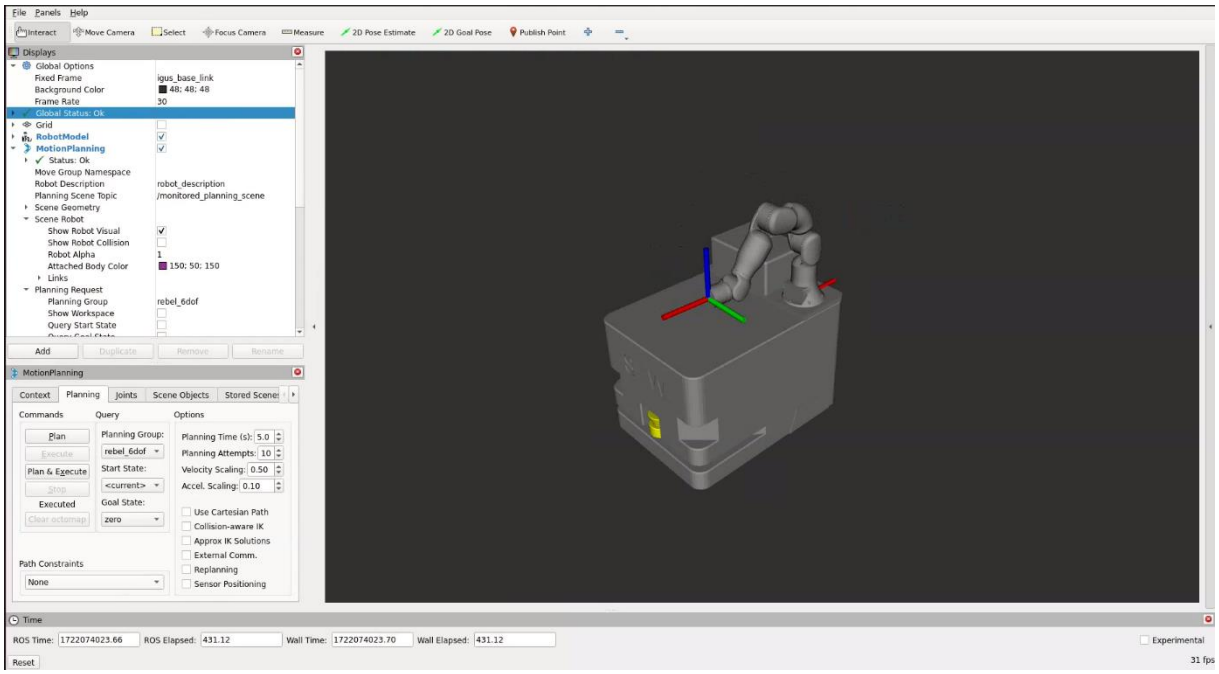

Abbildung 8-4: Roboter in RViz

Wie der Roboter auf dem SEW AGV einzuschalten ist und wie die Terminal-Befehle zur Bedienung des Roboters sind, ist in den Readmes des zugehörigen GitHub Repos ausführlich dokumentiert [20], [22].

8.1.1 Überblick ROS2 Packages

Um einen Überblick über die hier vorliegende Implementierung zu geben, werden im Folgenden die einzelnen ROS2-Packages beschrieben. Die iRC Packages stammen hierbei aus dem offiziellen Igus ReBeL GitHub Repository von CommonplaceRobotics [21] und wurden von uns modifiziert. In Kapitel 5.1 wurde die grundsätzliche ROS2 Architektur unserer Implementierung bereits erörtert.

8.1.1.1 Description Packages

Wie bereits in Kapitel 5.3 beschrieben liefern die Description Packages die vollständige kinematische Definition und die CAD-Daten unseres Robotersystems. Hierbei gibt es zunächst zwei getrennte URDFs für den Roboter und das AGV, die dann anschließend zusammengeführt werden. In diesem Teil dient das AGV nur als statisches Modell zur Kollisionsvermeidung bei der Bahnplanung des Roboters. Im Vergleich zu Kapitel 5 wurden hierbei über launch-Argumente die Hardware-Interfaces für Gazebo mit denen für die reale Hardware ausgetauscht und die Gazebo Sensor-Plugins nicht geladen.

irc_ros_description:
Dieses Package enthält die kinematische Beschreibung und die ROS2-Control-Definitionen des Igus ReBeL Arms.

sew_agv_description:
Hier wird die kinematische Beschreibung des an der Hochschule Karlsruhe vorhandene sew-maxo-mts AGV bereitgestellt. Dieses AGV ist Teil des Robotersystems, wird jedoch in einem separaten Container verwaltet und wird hier nur als statisches Modell zur Kollisionsvermeidung verwendet.

sew_and_igus_description:
Dieses Package kombiniert die kinematische Beschreibung des Arms und des AGV zu einem vereinten Roboter mit mehr Freiheitsgraden (DoF).

8.1.1.2 Bringup and Control Packages:

Diese Packages bieten zusätzliche Funktionalitäten für die Hardware-Kommunikation.

irc_ros_bringup:
Dieses Package kümmert sich um das korrekte Bringup aller benötigten ROS2-Knoten. Es muss vom Benutzer gestartet werden, um das System zu starten. Der Code wurde aus dem Repository von CommonplaceRobotics [21] geklont, jedoch stark modifiziert.

Die von CommonplaceRobotics bereitgestellte Implementierung ist für die Plug-And-Play Version des Igus ReBeL Roboters mit integrierter Steuerung vorgesehen. Diese hat wie in Abbildung 8-5 zu sehen zusätzliche DIO's und einen Anschluss für einen Not-Halt, der ebenfalls zusätzlich über einen DIO überwacht wird. Durch das Wegfallen dieser Steuerungseinheit bei der Open-Source Variante musste dies in der Implementierung berücksichtigt werden.

1	Spannungsversorgung
2	Not-Aus
3	Soft-An/Aus-Schalter
4	Digitale Ein- und Ausgänge Basis
5	Ethernet
6-9	USB-Anschlüsse

Abbildung 8-5: Igus ReBeL Steuerung Plug-And-Play Version [30, p. 24]

irc_ros_controllers:

Hier werden die DIO-Controller definiert, um die DIO-Ports am Igus-Roboterarm zu nutzen. Diese sind derzeit nicht verfügbar, da in diesem Projekt die Open-Source-Version des Roboters verwendet wird, bei der keine Hardware-DIO-Ports verfügbar sind.

irc_ros_hardware:

Dieses Package definiert den Controller mit einer Hardware-Schnittstelle für die CAN-Kommunikation zwischen dem Container und den verschiedenen Achsenmodulen.

irc_ros_msgs:

Dieses Package definiert mehrere benutzerdefinierte Nachrichten- und Servicetypen, die für interne Kommunikationszwecke erforderlich sind, wenn das Robotersystem aktiv ist.

8.1.1.3 Motion Planning and Application Packages:

Diese Packages bieten alle Funktionalitäten bezüglich der Bewegungsplanung des Roboters und der Benutzeroberfläche.

trac-ik:

Dieses Package enthält den Quellcode und die Plugin-Definition für einen IK-Solver, der in unserer Konfiguration von MoveIt verwendet wird. Der Code wurde aus [11] geklont.

sew_and_igus_moveit_config:

Hier wird die Konfiguration der Bewegungsplanung mit MoveIt2 in ROS2 bereitgestellt und die benötigten ROS Nodes verwaltet, um Trajektorien zu planen und auszuführen.

moveit_wrapper:

Dieses Package stellt die ROS service servers bereit, die die Bewegungsplanung über die C++ move_group-Schnittstelle steuern können.

igus_moveit_clients:

Dieses Package stellt eine Python-Klasse zur Verfügung, die Clients verwaltet, die sich mit den Servern des moveit_wrapper-Package verbinden. Der Benutzer kann die Methoden der Klasse aus einer übergeordneten Python-Datei aufrufen, um eine einfache Nutzung der MoveIt2-Bewegungsplanungen zu ermöglichen.

robot_application:

Dieses Package bietet eine einfache programmierbare Schnittstelle für das ROS2-Ökosystem. Der Benutzer kann seine eigene Steuerungslogik in Python-Code implementieren, um das Robotersystem in Simulation und Realität zu bewegen, indem er die Methoden der Client-Klassen aufruft.

8.2 Not-Halt

Wie bereits in Kapitel 7 beschrieben wurde von uns auch ein Not-Halt vorgesehen. Dieser trennt die 24 V Versorgung der Motoren und bewirkt somit einen sofortigen Stillstand des Roboters. Anders als bei der Plug-And-Play Version des Igus ReBeLs wird der Not-Halt jedoch nicht über einen DIO überwacht (siehe Kapitel 8.1.1.2). Die Software erkennt somit einen Fehler und der Roboter kann nach Beendigung des Not-Halts nicht selbstständig wieder anlaufen. Dafür wurde ein Reboot Taster integriert, der den Raspberry Pi im Roboterfuß neu startet und der Roboterbetrieb anschließend fortgesetzt werden kann. Not-Halt und Reboot Taste des Roboters sind in Abbildung 8-6 zu sehen.

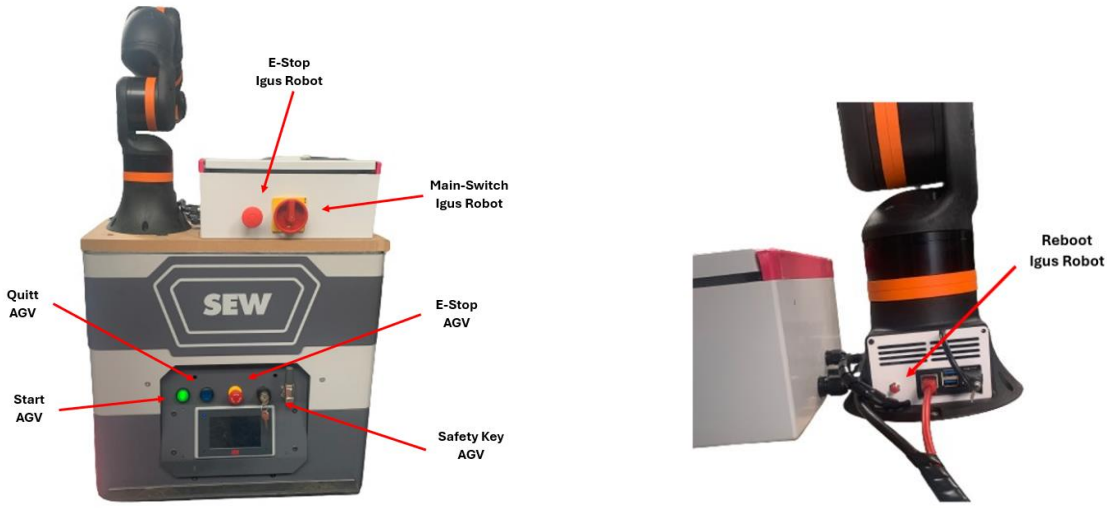

Abbildung 8-6:Schalter und Taster am AGV und Roboter

9. Inbetriebnahme SEW-AGV

Im Rahmen dieser Projektarbeit wurde neben der Inbetriebnahme des Igus ReBeL Roboters auch das an der Hochschule Karlsruhe verfügbare AGV der Firma SEW mit dem Softwareframework ROS2 in Betrieb genommen.

Abbildung 9-1: SEW MAXO MTS AGV

In einem Vorgängerprojekt [23] wurde bereits eine einfache ROS1 Python Implementierung umgesetzt, die es ermöglicht das AGV über eine in Abbildung 9-2 dargestellte Webseite zu steuern.

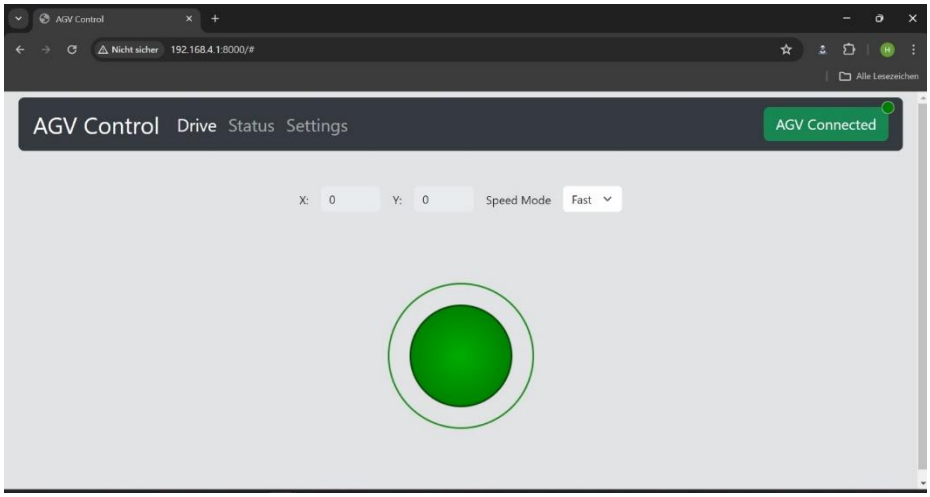

Abbildung 9-2: Website zur Steuerung des AGV mit ROS1 Implementierung aus Vorgängerprojekt

Um die Steuerung des AGVs in unser Gesamtkonzept zu integrieren, musste eine neue ROS2 Implementierung in C++ für ROS2-Control aufgesetzt werden. Hierbei konnte auf das Kommunikationskonzept zwischen PC und AGV der Vorarbeit zurückgegriffen werden.

Ein wesentlicher Vorteil von ROS2-Control liegt in seiner hohen Modularität und Flexibilität. Durch die modulare Architektur können verschiedene Steuerungskomponenten unabhängig voneinander entwickelt und integriert werden. Zusätzlich abstrahiert ROS2-Control die Hardware, indem es eine einheitliche Schnittstelle zur Steuerung unterschiedlicher Hardwarekomponenten bietet. Dies erleichtert die Integration neuer Hardware und die Wiederverwendbarkeit von Code, was die Entwicklung effizienter und weniger fehleranfällig macht [24].

Für die aktuelle Hardwarefunktionalität des AGVs ist die ROS2-Control-Funktionalität jedoch nicht zwingend erforderlich. Wie im Vorgängerprojekt gezeigt wurde, ist ein einfaches manuelles Steuern des AGVs mit deutlich geringerem Implementierungsaufwand möglich. Für das Zusammenspiel aller Komponenten, wie in unserer Gazebo-Simulation (siehe Kapitel 5 und 6) demonstriert, ist ROS2-Control jedoch unerlässlich. In Nachfolgearbeiten soll diese Funktionalität auch auf den Hardwareaufbau übertragen werden. Mit der von uns bereitgestellten Implementierung wurden somit alle notwendigen Grundlagen für die nächsten Schritte geschaffen.

Um eine einfache Anwendbarkeit des Codes zu gewährleisten, wurde hier ebenfalls ein Docker Container verwendet. Der Code ist in dem verlinkten GitHub Repository [25] zu finden.

Wie auch für den Roboter in Kapitel 8 beschrieben, kann der Docker Container während der Entwicklungsphase direkt auf einem PC ausgeführt werden. Der PC muss dafür über ein Netzwerkkabel mit dem AGV verbunden sein. Wo das Netzwerkkabel am AGV einzustecken ist, ist in Abbildung 9-3 zu sehen.

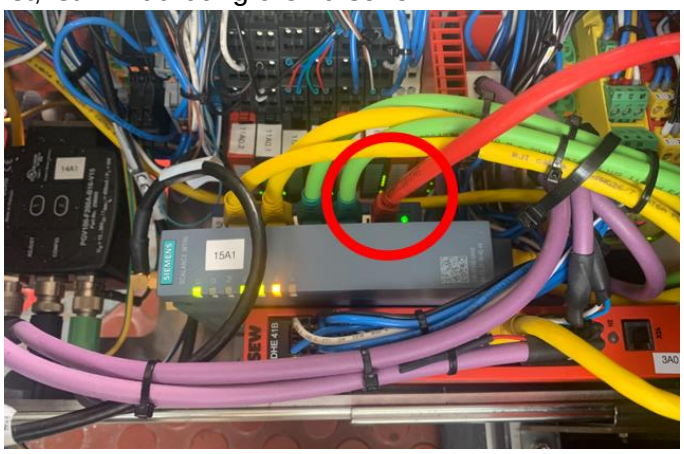

Abbildung 9-3: AGV Netzwerkkabel

Für den regulären Betrieb ist wie bereits erwähnt ein Raspberry Pi 5 in den Fuß des Roboters integriert. Auf diesem Raspberry Pi läuft sowohl der Docker Container für den Roboter (siehe Kapitel 8) als auch der Container für das AGV.

Außerdem stellt der Raspberry Pi ein Wifi Netzwerk zur Verfügung mit dem man sich mit dem PC verbinden und das AGV somit fernsteuern kann. Dafür muss auf dem PC der gleiche Docker Container mit gleicher *ROS_DOMAIN_ID* aktiv sein. Die *ROS_DOMAIN_ID* wird im *.env-File* definiert.

Das AGV kann von diesem PC dann entweder über die Tastatur oder über einen angeschlossenen Xbox-Controller gesteuert werden. Dafür muss die Taste „X“ gedrückt gehalten werden, während das AGV über den linken Joystick verfahren werden kann. Eine detaillierte Anleitung mit Terminal-Befehlen ist in den Readmes der verlinkten GitHub Repositories [22] und [25] zu finden.

9.1 Softwareaufbau SEW AGV

Unsere Software für das AGV mit ausführlicher Dokumentation ist in unserem GitHub Repository [25] zu finden und beinhaltet die folgenden drei ROS2 Packages:

sew_agv_description

Das Package sew_agv_description bietet die vollständige kinematische Definition und CAD-Daten des SEW-MAXO-MTS AGV. In ROS2 wird die Kinematik des Roboters/AGV in einem URDF-Modell definiert. Das URDF-Modell kann mithilfe des xacro-package strukturiert werden, indem sub-macros definiert werden, die alle im Haupt-URDF zusammengeführt werden. Darüber hinaus sind einige Tags bezüglich der Hardware-Kommunikation mit ROS2-Control und einige Tags bezüglich der Gazebo-Simulation spezifiziert, um Sensoren wie Lidar und Tiefenkamera zu simulieren. Die Gazebo-Komponenten werden in diesem Repository nicht verwendet und über definierte Launch-Argumente aus dem Bringup bequem deaktiviert. Es wird jedoch das identische Package wie in der Gazebo-Simulation verwendet. [26] (siehe Kapitel 5).

sew_agv_drivers

Dieses Treiber Package ist verantwortlich für die Hardware-Kommunikation mit dem AGV. Es sendet Bewegungsdaten, bestehend aus der x- und y-Richtung, Geschwindigkeit und Geschwindigkeitsmodus, über ein Netzwerkkabel mittels UDP-Protokolls an das AGV. Zusätzlich liest das Package Statusdaten vom AGV aus, um dessen aktuellen Zustand zu überwachen. Für die Bewegungsbefehle wird hier der ROS2 *diff_drive_controller* [27] verwendet.

Der *diff_drive_controller* steuert Differentialantriebe, indem er die Geschwindigkeit und Richtung der linken und rechten Räder reguliert. Dies ermöglicht dem Roboter, präzise Manöver wie Drehungen auf der Stelle oder Kurvenfahrten auszuführen [27]. Zusätzlich berechnet der Controller automatisch die Odometrie, was für diese Projektarbeit besonders wichtig ist, da das AGV keine eigene Position zurückmeldet.

Da die Motoren des AGV nicht direkt gesteuert werden, sondern das AGV Bewegungsdaten in den x- (Rotation) und y- (Vorwärtsbewegung) Richtungen erhalten muss, wurde ein Workaround implementiert. Hierbei werden die Radgeschwindigkeiten, die vom Hardware-Interface berechnet werden in Geschwindigkeiten in x- und y-Richtungen zurück gerechnet und anschließend an das AGV übertragen.

sew_agv_navigation

Dieses Package ermöglicht die Verbindung zu einem Xbox Controller oder der Tastatur des PCs, die zur manuellen Steuerung des AGV verwendet werden können, und stellt alle notwendigen Nodes zur Verfügung, um das AGV autonom in einer aufgezeichneten Karte mit dem ROS2 Nav2 Stack zu navigieren. Funktionen für die 2D-SLAM-Kartierung sind ebenfalls enthalten. Da zum derzeitigen Stand der Laserscanner des AGV nicht ausgelesen werden kann, ist die autonome Navigation nur in der Gazebo Simulation möglich (siehe Kapitel 5.4). In einem Nachfolgeprojekt soll das AGV mit einem Kamerasystem erweitert werden und somit auch die autonome Navigation mit der Hardware ermöglicht werden.

9.2 Systematischer Softwareablauf

Der systematische Softwareablauf ist in Abbildung 9-4 dargestellt. Die Steuerungsbefehle für das AGV kommen entweder von einem Joystick/ einer Tastatur oder aus der autonomen Navigation. Da zum derzeitigen Stand der Laserscanner des AGV nicht ausgelesen werden kann, ist die autonome Navigation nur in der Gazebo Simulation möglich (siehe Kapitel 5.4). Über den *twist_mux* Multiplexer [28] wird sichergestellt, dass immer nur ein Kanal der Bewegungsbefehle weitergegeben wird. Der Joystick hat hierbei eine höhere Priorität als die autonome Navigation. Die Navigationsbefehle werden also mit den Befehlen des Joysticks überschrieben, wenn dieser betätigt wird.

Die Bewegungsbefehle aus dem Multiplexer werden dann über ein Topic an den *diff_drive_controller* übergeben. Dieser berechnet daraus die Bewegungsbefehle für das AGV und übergibt diese über das *command interface* des Hardware-Interfaces an die Hardware. Über das *state interface* des Hardware-Interfaces kann der Status des AGVs an den *diff_drive_controller* zurückgegeben werden. Da das AGV allerdings keine Information über seine Position zurück gibt wird der *diff_drive_controller* hier im Open-Loop Modus betrieben. Dies bedeutet, dass der *diff_drive_controller* die Orometrie anhand der gesendeten Bewegungsbefehle berechnet.

Das Hardware-Interface ist für die Kommunikation mit dem AGV verantwortlich. Es baut über ein Netzwerkkabel eine Verbindung mit dem AGV auf und sendet die Bewegungsbefehle an das AGV. Außerdem liest es die Statusnachrichten des AGV aus.

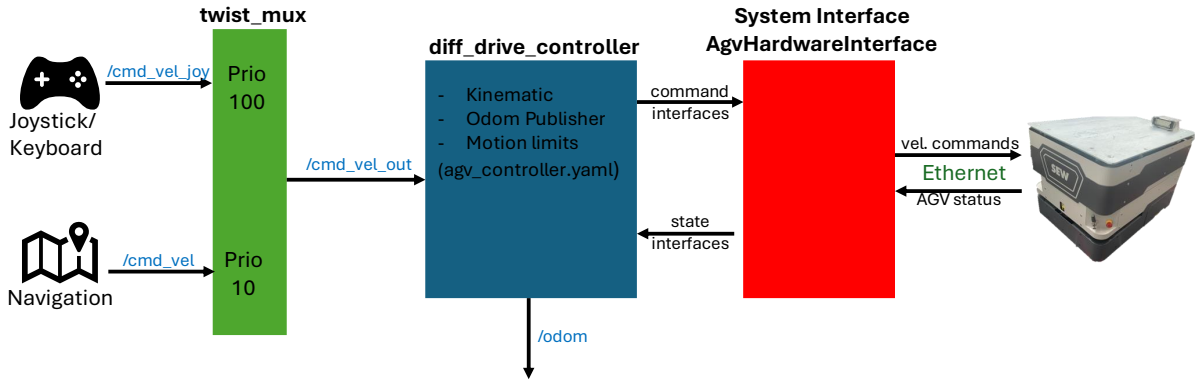

Abbildung 9-4: Systemarchitektur AGV Steuerung

10. Zusammenfassung und Ausblick

In der vorgestellten Projektarbeit ist es erfolgreich gelungen, ein Koordinationskonzept zur Steuerung eines Verbundsystems aus einem Roboterarm und einem Automated Guided Vehicle (AGV) zu entwickeln, um die autonome Navigation des AGV und Positionierung des Aktors in einem Hochregallager zu ermöglichen. Dabei wurden wesentliche Funktionen in einer Simulationsumgebung getestet, da die reale Hardware noch nicht vollständig implementiert ist. Die Simulation ermöglichte es, die Kollisionsvermeidung und die Bewegungsplanung des Roboterarms sowie das SLAM-Verfahren des AGV zu entwickeln und zu evaluieren.

Falls es in Zukunft gelingen würde Zugriff auf die vom Lidarscanner aufgenommenen Umgebungsdaten zu erlangen, könnte das in dieser Arbeit implementierte SLAM-Framework leicht von der Simulation auf die Realität übertragen werden. Hierzu gibt es bereits fertige ROS2 Nodes, welche das RS422 Protokoll entschlüsseln und nahtlos in das bestehende Framework integriert werden könnten. Mit der Implementierung einer weiteren Tiefenkamera am TCP des Roboterarms könnte nun, wie in der Simulation erfolgreich validiert, die Lagerposition nach der Navigation neu lokalisiert werden. Diese Anpassung könnte dabei helfen, den Einfluss der Lokalisierungs- und Positionierungsungenauigkeiten des AGV zu verringern, die in der realen Anwendung auftreten könnten.

Durch Montage eines geeigneten Greifers und Anwendung des momentan in der Entwicklung befindlichen Verfahrens zur Greifpositionserkennung von Lagergegenständen, könnte das in der Simulation getestete Gesamtsystem so schon bald Anwendung in der Realität finden, um das Hochregallager des Logistiklabors autonom zu Be- und Entladen.

11. Quick Start Guide

Dieses Kapitel dient nur dem schnellen Überblick. Nähere Informationen und genauere Anleitungen befinden sich in dem Main-Repository [22] und den Unter-Repositories [20], [25], [26].

Um das AGV mit dem Roboter zu verwenden, müssen das AGV und der Roboter in der angegebenen Reihenfolge eingeschaltet werden. Der Raspberry Pi, der sich im Sockel des Roboters befindet, richtet alles automatisch ein und startet die ROS-Packages. Sobald der Raspberry Pi vollständig hochgefahren ist, lässt sich ein anderer PC, auf dem Ubuntu läuft, mit dem vom Raspberry Pi erstellten WLAN verbinden. Auf dem PC lassen sich dann die beiden Docker-Container erstellen und starten. Eine Internetverbindung ist während des Build-Prozesses erforderlich. Daher sollte man sich erst nach dem Build mit dem WLAN des Raspberry Pi verbinden. Im *igus_rebel_ros2_docker* Docker-Container muss RViz gestartet werden, und im *sew_maxo_mts_ros2* Docker-Container muss die Launch-Datei für den Joystick gestartet werden. Zum Ausschalten des Roboters und des AGV muss erneut die

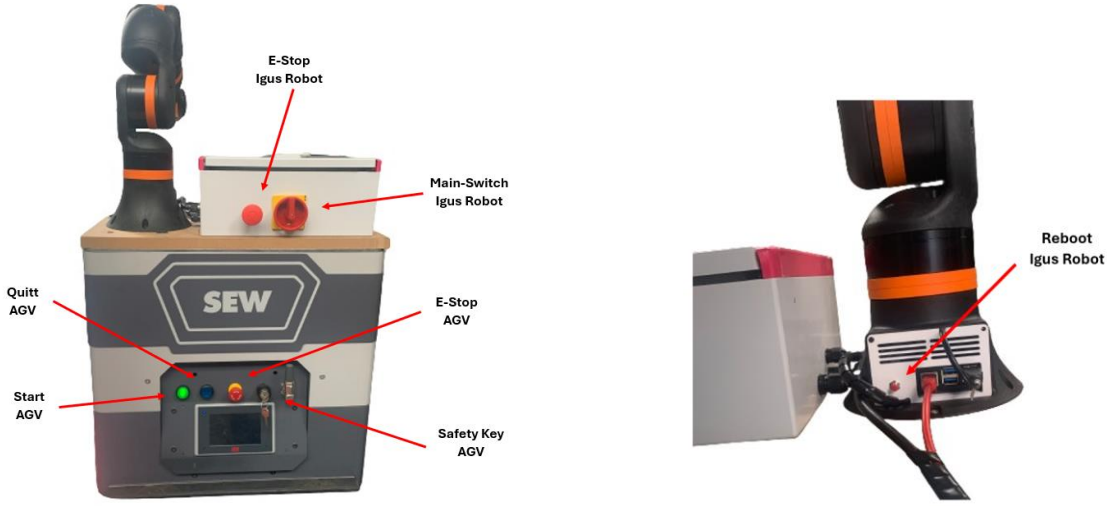

Abbildung 11-1:Schalter und Taster am AGV und Roboter

angegebene Reihenfolge eingehalten werden.

11.1 Einschalten des AGV und des Roboters

1. Stelle sicher, dass der Hauptschalter des Roboters ausgeschaltet ist (Hinweis: Die Aus-Position ist die vertikale Schalterposition (|) und die Ein-Position ist die horizontale Schalterposition (–)).
2. Not-Aus-Schalter des AGV ziehen.
3. AGV einschalten, indem die grüne und blaue Taste einige Sekunden lang gedrückt wird.
4. Das Display schaltet sich nun ein und die grüne Taste leuchtet.

5. Not-Aus-Schalter des Roboters ziehen, falls er gedrückt ist.
6. Hauptschalter des Roboters einschalten.
7. Sobald der Raspberry Pi vollständig hochgefahren ist, wird das AGV-WLAN sichtbar, die grüne Taste leuchtet nicht mehr und das Display zeigt „RC“ in einem blauen Feld unten rechts an. Der Roboter gibt ein leises Summen von sich und aktiviert die einzelnen Achsen, was als Klicken zu hören ist.
8. Bei Fehlern kann der Raspberry Pi über die Reboot-Taste im Sockel des Roboters neu gestartet werden. Ein Neustart ist auch erforderlich, wenn der Not-Aus-Schalter des Roboters aktiviert wurde.

11.2 Bauen und Starten der Docker-Container

1. Beide Repositories [20] und [25] klonen:

   git clone https://github.com/mathias31415/igus_rebel_ros2_docker.git

   git clone https://github.com/RobinWolf/sew_maxo_mts_ros2.git
2. Sicherstellen, dass man sich auf dem richtigen Branch befindet

   git checkout <branchname>
3. Docker Container bauen und starten. Dazu muss man in das jeweilige Verzeichnis navigieren (mit *cd*) und dort die Skripte zum Bauen und Starten ausführen. Dafür muss eine Internetverbindung vorhanden sein.

   ./build_docker.sh

   ./start_docker.sh
4. Docker-Container durch Drücken von CTRL + C verlassen. Der Docker-Container wird weiterhin im Hintergrund ausgeführt.

11.3 Mit dem WLAN des Raspberry Pi verbinden

Mit dem PC mit dem vom Raspberry Pi aufgemachten WLAN-Netztwerk verbinden. Das WLAN-Netzwerk hat den Namen „AGV“ und das Passwort „agv12345“

11.4 RViz im *igus_rebel_ros2_docker* Docker-Container starten

1. Erneut mit dem *igus_rebel_ros2_docker* Docker-Container verbinden

   docker exec -it igusrebel bash
2. Sourcen

   source install/setup.bash
3. RViz starten

   ros2 launch irc_ros_bringup rviz.launch.py

Jetzt lässt sich RViz verwenden, um den Roboterarm zu bewegen.

11.5 Joystick im *sew_maxo_mts_ros2* Docker container verwenden um das AGV zu verfahren

1. Erneut mit dem *igus_rebel_ros2_docker* Docker-Container verbinden (in einem zweiten Terminal)

   docker exec -it sew_navigation bash

2. Sourcen

   source install/setup.bash

3. Joystick Launch-Datei ausführen

   ros2 launch sew_agv_navigation joystick.launch.py Docker-Container

11.6 Ausschalten des AGV und des Roboters

1. Not-Aus-Schalter des Roboters drüchen, um die 24V-Stromversorgung des Roboters zu unterbrechen (die Logikspannung bleibt erhalten).
2. Hauptschalter des Roboters ausschalten.
3. Not-Aus-Schalter des AGV drücken.
4. AGV ausschalten, indem die grüne und blaue Taste einige Sekunden lang gedrückt wird.
5. Das Display schaltet sich aus und die Tasten leuchten nicht mehr.

12. Literaturverzeichnis

[1] Open Robotics, „Gazebosim," 21 07 2024. [Online]. Available: https://gazebosim.org/home.

[2] SEW Eurodrive, „sew eurodrive," [Online]. Available: https://www.sew-eurodrive.de/automatisierung/anlagenautomatisierung/mobile-systeme/mobile-transportsysteme/mobile-transportsysteme.html. [Zugriff am 21 07 2024].

[3] Igus, „igus motion plastics," [Online]. Available: https://www.igus.de/robolink/rebel-cobot. [Zugriff am 21 07 2024].

[4] Open Robotics, „ros2 humble," [Online]. Available: https://docs.ros.org/en/humble/index.html. [Zugriff am 21 07 2024].

[5] Open Robotics, „Ros2 URDF," [Online]. Available: https://docs.ros.org/en/humble/Tutorials/Intermediate/URDF/URDF-Main.html. [Zugriff am 21 07 2024].

[6] Open Navigation, „Nav2," [Online]. Available: https://docs.nav2.org/. [Zugriff am 21 07 2024].

[7] „navigation2 with slam," [Online]. Available: https://docs.nav2.org/tutorials/docs/navigation2_with_slam.html. [Zugriff am 21 07 2024].

[8] PickNik Robotics, „Moveit2 Humble," [Online]. Available: https://moveit.picknik.ai/humble/doc/examples/urdf_srdf/urdf_srdf_tutorial.html. [Zugriff am 21 07 2024].

[9] Kavraki Lab Department of Computer Science Rice University, [Online]. Available: https://ompl.kavrakilab.org/. [Zugriff am 21 07 2024].

[10] Orocos, „Orocos Kinematics and Dynamics," [Online]. Available: https://www.orocos.org/wiki/Kinematic_and_Dynamic_Solvers.html. [Zugriff am 21 07 2024].

[11] „trac-ik," [Online]. Available: https://bitbucket.org/traclabs/trac_ik/src/rolling-devel/. [Zugriff am 29 07 2024].

[12] TracLabs, „Trac-IK,“ [Online]. Available: https://traclabs.com/projects/trac-ik/. [Zugriff am 21 07 2024].

[13] University of Freiburg, „OctoMap An Efficient Probabilistic 3D Mapping Framework Based on Octrees,“ [Online]. Available: https://octomap.github.io/. [Zugriff am 21 07 2024].

[14] PickNik Robotics, „Github Moveit2 MoveGroup Interface,“ [Online]. Available: https://github.com/moveit/moveit2/blob/main/moveit_ros/planning_interface/move_group_interface/include/moveit/move_group_interface/move_group_interface.h. [Zugriff am 21 07 2024].

[15] M. D. T. L. U. Nassal, „Technische Universität München,“ 1994. [Online]. Available: chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://www.mec.ed.tum.de/fileadmin/w00cbp/mimed/Publications/1994_WCRR_Nassal.pdf. [Zugriff am 07 08 2024].

[16] SEW-EURODRIVE, „https://www.manualslib.com/,“ 12 2017. [Online]. Available: https://www.manualslib.com/products/Sew-Eurodrive-Maxolution-Maxo-Mts-T005-P-00-B04-10732763.html. [Zugriff am 08 08 2024].

[17] igus motion plastics, „ReBeL Cobot 6 DOF,“ [Online]. Available: https://www.igus.de/product/21465?artNr=REBEL-6DOF-OS. [Zugriff am 29 07 2024].

[18] Distrelec, „PAP-07V-S - Crimpgehäuse Buchse / Steckbuchse 7 Positionen 2mm, JST,“ [Online]. Available: https://www.distrelec.de/de/crimpgehaeuse-buchse-steckbuchse-positionen-2mm-jst-pap-07v/p/30021707?itemList=cart. [Zugriff am 29 07 2024].

[19] rs-online, „Innomaker USB-CAN-Konverter Kommunikation,“ [Online]. Available: https://de.rs-online.com/web/p/raspberry-pi-hats-und-add-ons/2526692?gb=s. [Zugriff am 29 07 2024].

[20] M. Fuhrer, „igus_rebel_ros2_docker,“ [Online]. Available: https://github.com/mathias31415/igus_rebel_ros2_docker. [Zugriff am 29 07 2024].

[21] CommonplaceRobotics , „iRC_ROS,“ [Online]. Available: https://github.com/CommonplaceRobotics/iRC_ROS. [Zugriff am 29 07 2024].

[22] M. Fuhrer, „igus_rebel_on_sew_agv,“ [Online]. Available: https://github.com/mathias31415/igus_rebel_on_sew_agv/tree/main. [Zugriff am 29 07 2024].

[23] T. Schneider, „ros_sew_agv,“ [Online]. Available: https://github.com/elekTom/ros_sew_agv . [Zugriff am 29 07 2024].

[24] „ros2_control,“ [Online]. Available: https://control.ros.org/humble/doc/getting_started/getting_started.html. [Zugriff am 08 01 20204].

[25] R. Wolf, „sew_maxo_mts_ros2,“ [Online]. Available: https://github.com/RobinWolf/sew_maxo_mts_ros2/tree/dev. [Zugriff am 29 07 2024].

[26] R. Wolf, „sew_maxo_mts_and_igus_rebel,“ [Online]. Available: https://github.com/RobinWolf/sew_maxo_mts_and_igus_rebel. [Zugriff am 29 07 2024].

[27] „diff_drive_controller,“ [Online]. Available: https://control.ros.org/master/doc/ros2_controllers/diff_drive_controller/doc/userdoc.html. [Zugriff am 29 07 2024].

[28] „twist_mux,“ [Online]. Available: https://wiki.ros.org/twist_mux. [Zugriff am 29 07 2024].

[29] „URDF,“ [Online]. Available: https://docs.ros.org/en/humble/Tutorials/Intermediate/URDF/URDF-Main.html. [Zugriff am 29 07 2024].

[30] igus, igus ReBeL Dokumentation Roboterarm mit integrierter Steuerung.